11.11.2024

Сортировки (продолжение). Медианы и порядковые статистики. Растущий массив. Очередь.

Филиппов Михаил Витальевич

m.filippov@g.nsu.ru

89232283872

Императивное программирование, 2024-2025

Новосибирский государственный университет

*НАСТОЯЩАЯ НАУКА

Давайте познакомимся

Филиппов Михаил Витальевич

- Окончил магистратуру ФФ НГУ
- Окончил аспирантуру ИТ СО РАН
- Являюсь м.н.с. ИТ СО РАН
- 7+ лет опыт в программировании C/C++

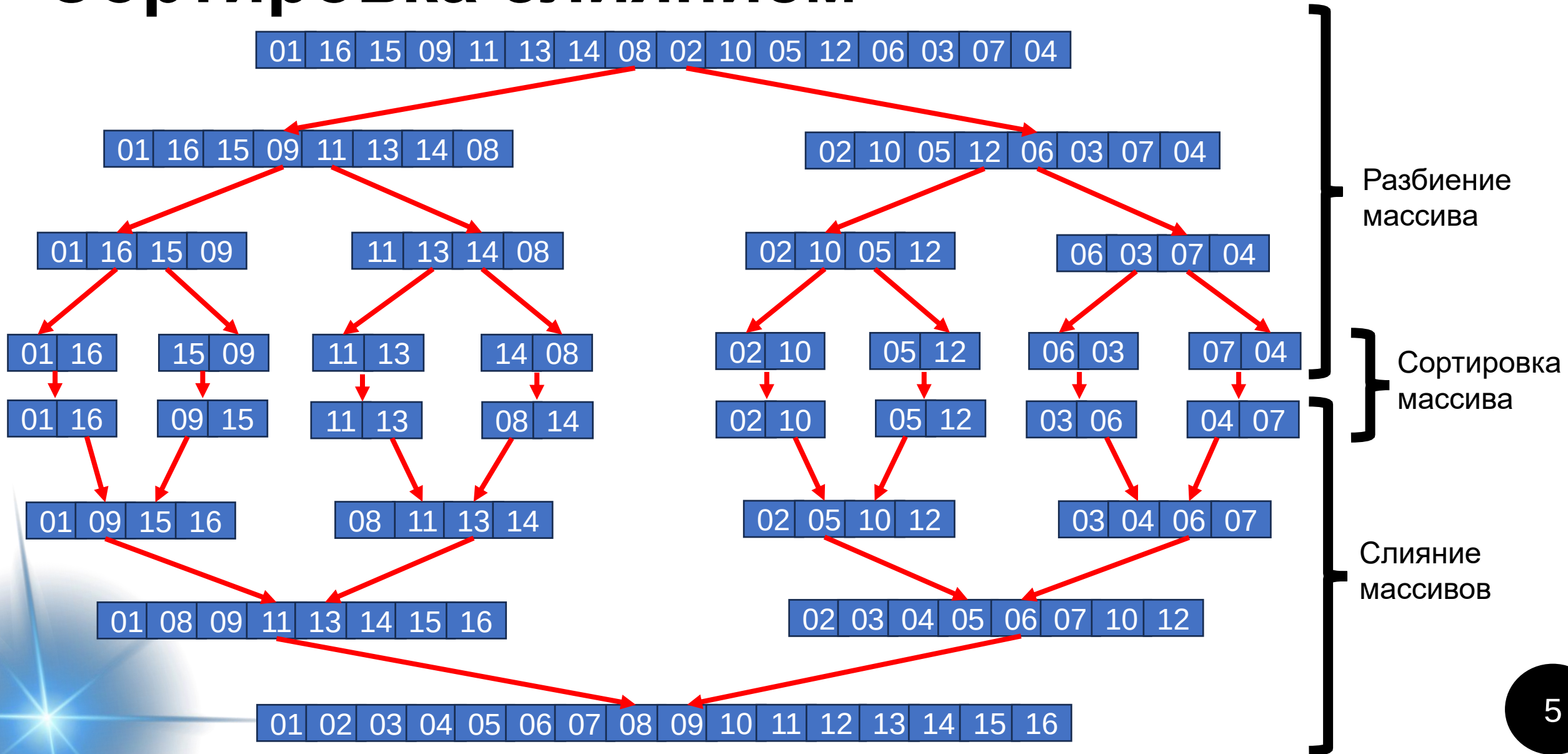
План лекции

Сортировки (продолжение)	Медианы и порядковые статистики	Растущий массив	Очередь
35 минут	15 минут	15 минут	25 минут

План лекции

Сортировки (продолжение)	Медианы и порядковые статистики	Растущий массив	Очередь
25 минут	15 минут	25 минут	25 минут

Сортировка слиянием



Сортировка слиянием

```
void mergesort(void **copy_ar, void **ar, int
(*cmp)(const void *, const void *), int left, int
right)
{
    if (right - left < 2)
        return;
    if (right - left == 2)
    {
        if (copy_ar[left] > copy_ar[left + 1])
        {
            void *temp = copy_ar[left];
            copy_ar[left] = copy_ar[left + 1];
            copy_ar[left + 1] = temp;
        }
        return;
    }
```

```
    int mid = (right + left) / 2;
    mergesort_array(copy_ar, ar, cmp, left, mid);
    mergesort_array(copy_ar, ar, cmp, mid, right);
    // слияние левой и правой сторон Ar
    int i = left, j = mid, idx = left;
    while (idx < right)
    {
        if (j >= right || (i < mid && ar[i] < ar[j]))
        {
            copy_ar[idx] = ar[i];
            i += 1;
        }
        else
        {
            copy_ar[idx] = ar[j];
            j += i;
        }
        idx += 1
    }
}
```

Скорость во всех случаях

$W(N) = N\log_2 N - N + 1 = O(N\log_2 N)$.

Как наиболее эффективно посчитать контрольную сумму (летящие биты по Wi-Fi)

Самое простое, что приходит

```
uint64_t CountBitsSlow(uint64_t x) {  // O(n)
  uint64_t count = 0;
  while (x) {
    count += x & 1u;
    x >>= 1;
  }
  return count;
}
```

Теперь поинтереснее

```
uint64_t CountBitsFast(uint64_t x) {  // O(n_1)
  uint64_t count = 0;
  while (x) {
    ++count;
    x &= (x - 1);
  }
  return count;
}
```

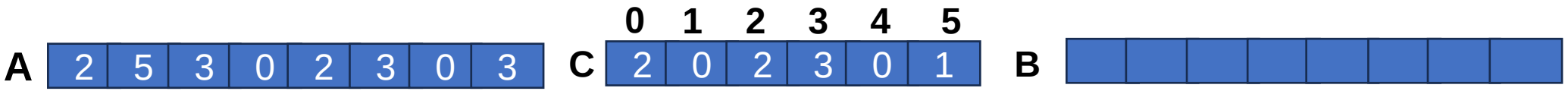
Ну как же без треша

```
uint64_t CountBitsFaster(uint64_t x) {
  x = (x & 0x5555555555555555) + ((x >> 1) & 0x5555555555555555);
  x = (x & 0x3333333333333333) + ((x >> 2) & 0x3333333333333333);
  x = (x & 0x0f0f0f0f0f0f0f0f) + ((x >> 4) & 0x0f0f0f0f0f0f0f0f);
  x = (x & 0x00ff00ff00ff00ff) + ((x >> 8) & 0x00ff00ff00ff00ff);
  x = (x & 0x0000ffff0000ffff) + ((x >> 16) & 0x0000ffff0000ffff);
  x = (x & 0x00000000ffffffff) + ((x >> 32) & 0x00000000ffffffff);
  return x;
}
```

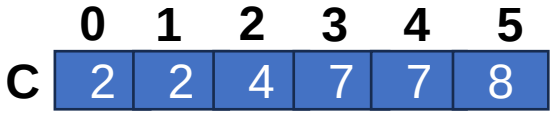
Можно ли любой сортировкой сравнениями быстрее nlog(n)?

Сортировка подсчетом

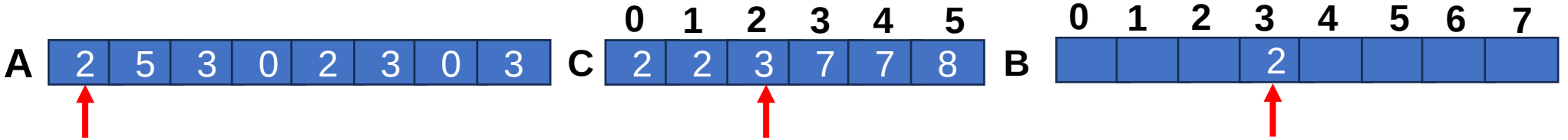
Шаг 1 – определяем сколько цифр 1, 2, 3,….

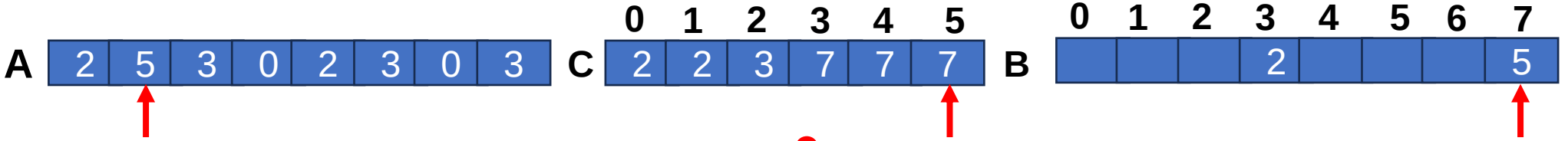


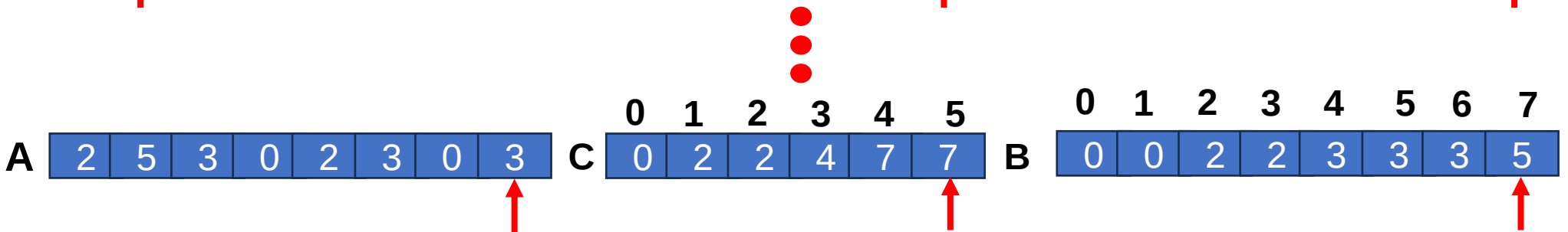
Шаг 2 – определяем сколько раз встречается цифра меньше k



Шаг 3 – сортировка





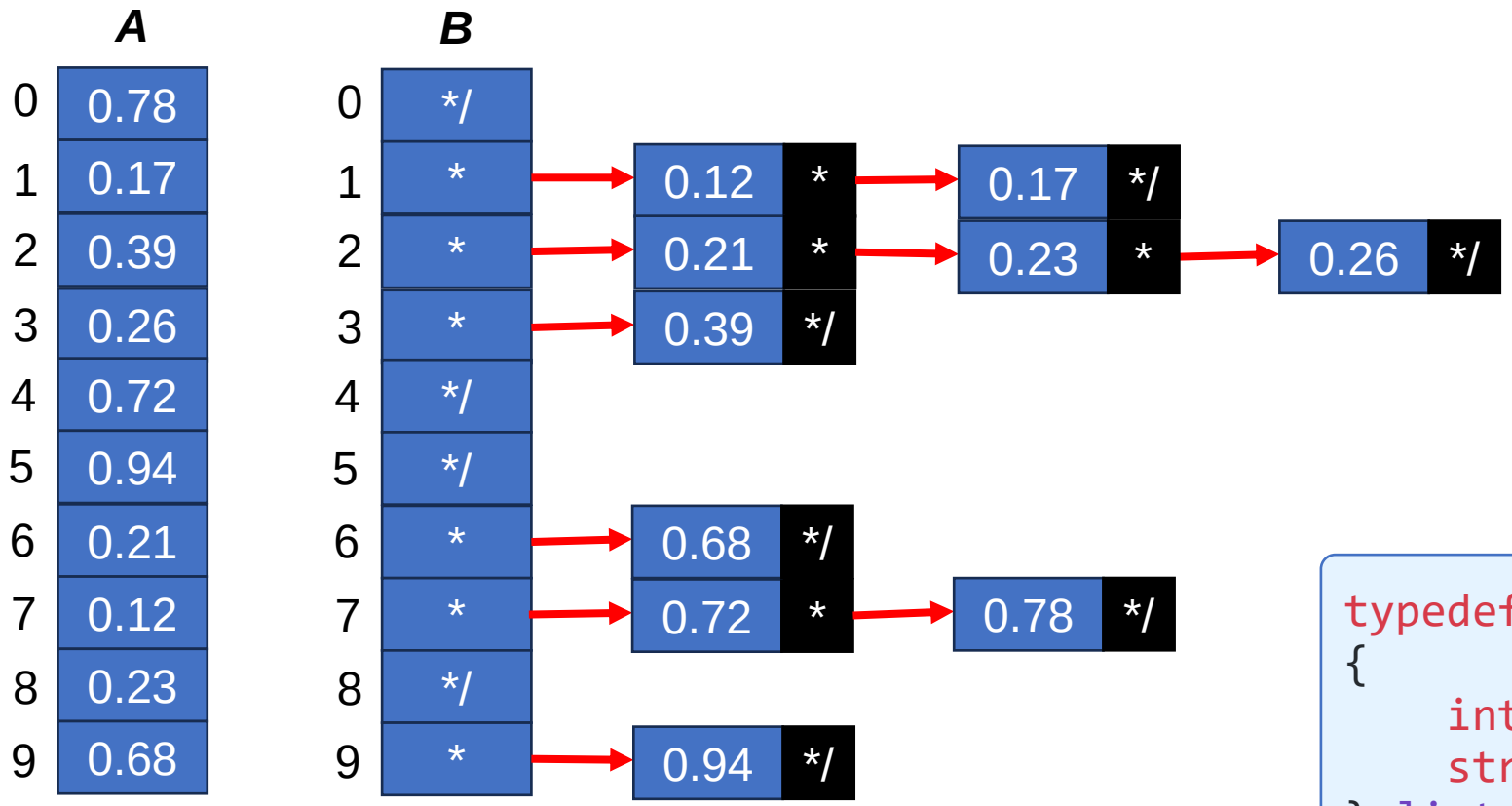


Сортировка подсчетом

```
// Counting-Sort(A, B, k)
void counting_sort(int *A, int *B, int k)
{
    int *C = (int *)calloc(k, sizeof(int)); // Пусть C[0…k] - новый массив, C[i] = 0
    for (size_t j = 0; j < sizeof(A) / sizeof(int); j++)
    {
        C[A[j]] = C[A[j]] + 1; // Сейчас C[i] содержит количество элементов, равных i.
    }
    for (size_t i = 1; i < k; i++)
    {
        C[i] = C[i] + C[i - 1]; // Сейчас C[i] содержит количество элементов, не превышающих i.
    }
    for (size_t j = 0; j < sizeof(A) / sizeof(int); j++)
    {
        B[--C[A[j]]] = A[j];
    }
}
```

Таким образом, общее время составляет $\Theta(k + n)$. На практике обычно сортировка подсчетом применяется, когда $k = O(n)$, и в этом случае общее время работы равно $\Theta(n)$.

Блочная (карманная) сортировка




```
typedef struct node
{
    int data;
    struct node *next;
} list;
list *start = NULL;
list *insert_beg(list *, int);
list **delete_list(list **);
list *sort_list(list *);
```

Блочная (карманная) сортировка

```
// Bucket-Sort(A)
list *merge(list *, int *, int) void bucket_sort(int *A) // A 0 - 1
{
    size_t n = sizeof A / sizeof(int);
    list **B = (list **)malloc(n * sizeof(list *)); // B[0…n - 1] - новый массив
    for (int i = 0; i < n; ++i)                     // Сделать B[i] пустым списком
        B[i] = NULL;
    for (int i = 0; i < n; ++i) // Вставить A[i] в список B[[nA[i]]]
        insert_beg(roundl(B[n * A[i]]), A[i]);
    for (int i = 0; i < n; ++i)
    { // Отсортировать список B[i]
        sort(B[i]);
    }
    for (int count = 0, i = 0; i < n; ++i)
    {
        merge(B[i], A, &count);
    } // Соединить списки B[0], B[1], …, B[n - 1] в указанном порядке
}
```

Блочная (карманная) сортировка

Чтобы оценить стоимость этих вызовов, введем случайную величину n_i, обозначающую количество элементов, попавших в карман B[i].

$$T(n) = \theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

E – математическое ожидание

$$E(T(N)) = E\left(\theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right) = \theta(n) + E\left(\sum_{i=0}^{n-1} O(n_i^2)\right) = \theta(n) + \sum_{i=0}^{n-1} O\left(E(n_i^2)\right)$$

Каждому i карману соответствует одна и таже величина $E(n_i^2)$, поэтому $n_i = \sum_{j=0}^{n} X_{ij}$

$$E(n_i^2) = E\left(\left(\sum_{j=0}^{n} X_{ij}\right)^2\right) = E\left(\sum_{j=0}^{n}\sum_{k=0}^{n} X_{ij}\, X_{ik}\right) = \sum_{j=0}^{n} E(X_{ij}^2) + \sum_{1<j<n}\ \sum_{\substack{1<k<n \\ k\neq j}} E(X_{ij})E(X_{ik}) = \sum_{j=0}^{n}\frac{1}{n} + \sum_{1<j<n}\ \sum_{\substack{1<k<n \\ k\neq j}} \frac{1}{n}\times\frac{1}{n}$$

$$= 2 - \frac{1}{n}$$

$$T(n) = \theta(n) + n * O\left(2 - \frac{1}{n}\right) = \theta(n)$$

Резюме

Сортировка	Время работы	Метод	Область использования
Вставкой	$O(N^2)$	Вставка	Очень малые массивы
Выбором	$O(N^2)$	Двусторонние прохождения, ограничения рассматриваемых пределов	Очень малые массивы
Пузырьковая	$O(N^2)$	Кучи, хранение полных деревьев в массиве	Крупные массивы с неизвестным распределением
Пирамидальная	$O(NlogN)$	Кучи, хранение полных деревьев в массиве	Крупные массивы с неизвестным распределением
Быстрая	$O(NlogN)$ ожидаемое, $O(N^2)$ в худшем случае	«Разделяй и властвуй», перемещение элементов в позицию, рандомизация во избежание худшего случая	Крупные массивы без большого количества дубликатов, параллельная сортировка
Слиянием	$O(NlogN)$	«Разделяй и властвуй», объединение, внешняя сортировка	Крупные массивы с неизвестным распределением, большие объемы данных, параллельная сортировка
Подсчетом	$O(N+M)$	Счет	Крупные массивы с достаточно единообразным распределением значений

План лекции

Сортировки (продолжение)	**Медианы и порядковые статистики**	Растущий массив	Очередь
25 минут	15 минут	25 минут	25 минут

Основные понятия

Будем называть ***i-й порядковой статистикой*** (order statistic) множества, состоящего из n элементов, i-й элемент в порядке возрастания. Например, *минимум такого множества* — это первая порядковая статистика (i = 1), а его *максимум* — это n-я порядковая статистика (i = n). *Медиана (median)* неформально обозначает середину множества. Если n нечетное, то медиана единственная, и ее индекс равен i = (n + 1)/2; если же n четное, то медианы две, и их индексы равны i = n/2 и i = n/2 + 1. Таким образом, независимо от четности n, медианы располагаются при i = [(n + 1)/2] (нижняя медиана (lower median)) и i = [(n + 1)/2] (верхняя медиана (upper median)). Однако для простоты в этой термин "медиана" относится к нижней медиане.

Мы посвятим себя задаче выбора i-й порядковой статистики в множестве, состоящем из n чисел. Формально задачу выбора (selection problem) можно определить следующим образом.

Вход. Множество A, состоящее из n (различных) чисел, и число $1 \le i \le n$.

Выход. Элемент $x \in A$, превышающий по величине ровно i - 1 других элементов множества A.

Основные понятия

Сколько сравнений необходимо для того, чтобы найти минимальный элемент в n-элементном множестве? Для этой величины легко найти верхнюю границу, равную n - 1 сравнениям: мы по очереди проверяем каждый элемент множества и следим за тем, какой из них является минимальным на данный момент.

```
int min(int *A)
{
    size_t n = sizeof A / sizeof(int);
    int min = A[0];
    for (size_t i = 1; i < n; ++i)
        if (A[i] < min)
            A[i] = min;
    return min;
}
```

Является ли представленный выше алгоритм оптимальным?
Да!

А если ищем максимум и минимум одновременно? Сколько сравнений потребуется?

Выбор в течение линейного ожидаемого времени

Общая задача выбора оказывается более сложной, чем простая задача поиска минимума. Однако, как это ни удивительно, время решения обеих задач в асимптотическом пределе ведет себя одинаково - как Θ(n). Рассмотрим алгоритм типа Randomized-Select, предназначенный для решения задачи выбора. Этот алгоритм разработан по аналогии с алгоритмом быстрой сортировки. Как и в алгоритме быстрой сортировки, в алгоритме Randomized-Select используется идея рекурсивного разбиения входного массива. Однако в отличие от алгоритма быстрой сортировки, в котором рекурсивно обрабатываются обе части разбиения, алгоритм Randomized-Select работает лишь с одной частью. Это различие проявляется в результатах анализа обоих алгоритмов: если математическое ожидание времени работы алгоритма быстрой сортировки равно O(nlgn), то ожидаемое время работы алгоритма Randomized-Select равно Θ(n), в предположении, что все элементы входного множества различны.

В алгоритме Randomized-Select используется процедура Randomized-Partition. Таким образом, подобно процедуре Randomized-Quicksort, Randomized-Select — это рандомизированный алгоритм, поскольку его поведение частично определяется выводом генератора случайных чисел. Приведенный ниже код процедуры Randomized-Select возвращает i-й в порядке возрастания элемент массива A[p…r].

Быстрая сортировка

01	16	15	09	11	13	14	08	02	10	05	12	06	03	07	04

8 – опорный элемент

Проверка

<8 левая

>8 правая

01	04	15	09	11	13	14	08	02	10	05	12	06	03	07	16

01	04	07	09	11	13	14	08	02	10	05	12	06	03	15	16

01	04	07	03	11	13	14	08	02	10	05	12	06	09	15	16

01	04	07	03	06	13	14	08	02	10	05	12	11	09	15	16

01	04	07	03	06	05	14	08	02	10	13	12	11	09	15	16

01	04	07	03	06	05	02	08	14	10	13	12	11	09	15	16

Наихудший случай $\frac{(N-1)N}{2} = O(N^2)$

Быстрая сортировка

```
int partition(void **ar, int (*cmp)(const void *, const void *),
              int left, int right, int pivotIndex) {
    int idx,
        store;
    void *pivot = ar[pivotIndex]; /* Перемещаем pivot в конец массива */
    void *tmp = ar[right];
    ar[right] = ar[pivotIndex];
    ar[pivotIndex] = tmp; /* Все значения, не превышающие pivot, перемещаются в начало
     * массива, и pivot вставляется сразу после них. */
    store = left;
    for (idx = left; idx < right; idx++) {
        if (cmp(ar[idx], pivot) <= 0) {
            tmp = ar[idx];
            ar[idx] = ar[store];
            ar[store] = tmp;
            store++;
        }
    }
    tmp = ar[right];
    ar[right] = ar[store];
    ar[store] = tmp;
    return store;
}
```

Выбор в течение линейного ожидаемого времени

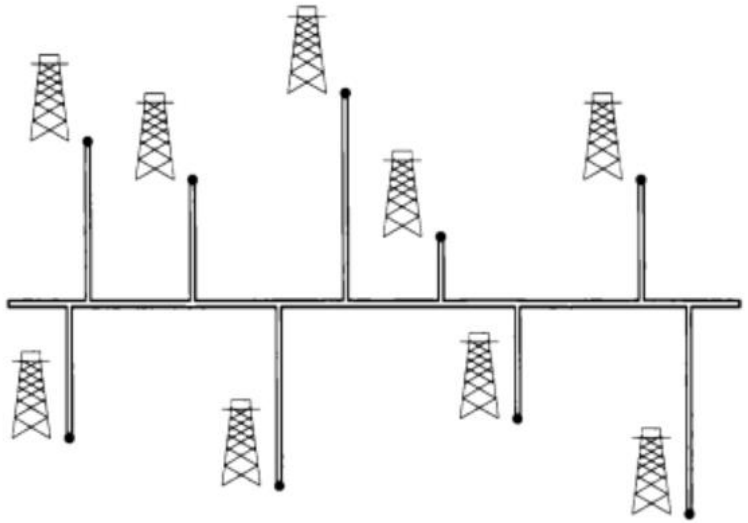
```c
// Randomized-partition(A, cmp, p,r)
//  Randomized-Select(A,p,r, i)
// int (*cmp)(const void *, const void *)

int randomized_partition(void **ar, int (*cmp)(const void *, const void *),
                         int left, int right, int pivotIndex)
{
    srand(time(NULL));
    int result = (rand() % (right - left)) + right;
    void *temp = ar[left];
    ar[left] = ar[result];
    ar[result] = temp;
    return partition(ar, cmp, left, right, pivotIndex);
}
int randomized_select(void **ar, int (*cmp)(const void *, const void *),
                      int left, int right, int pivotIndex, int i)
{
    if (left == right)
        return ar[left];
    int q = randomized_partition(ar, cmp, left, right, pivotIndex);
    int k = q - left - 1;
    if (i == k)
        return ar[right];
    else if (i < k)
        return randomized_select(ar, cmp, left, q - 1, pivotIndex, i);
    return randomized_select(ar, cmp, q + 1, right, pivotIndex, i);
}
```

Алгоритм выбора с линейным временем работы в наихудшем случае

1. Все n элементов входного массива разбиваются на [n/5] групп по 5 элементов и одну группу, содержащую оставшиеся n mod 5 элементов (впрочем, эта группа может оказаться пустой).
2. Сначала методом сортировки вставкой сортируется каждая из [n/5] групп (содержащих не более 5 элементов), а затем в каждом отсортированном списке, состоящем из элементов групп, выбирается медиана.
3. Путем рекурсивного использования процедуры SELECT определяется медиана x множества из [n/5] медиан, найденных на шаге 2. (Если этих медиан окажется четное количество, то, согласно принятому соглашению, переменной x будет присвоено значение нижней медианы.)
4. С помощью модифицированной версии процедуры PARTITION входной массив делится относительно медианы медиан x. Пусть число k на единицу превышает количество элементов, попавших в нижнюю часть разбиения. Тогда x - k-й в порядке возрастания элемент, и в верхнюю часть разбиения попадает n – k элементов.
5. Если i = k, то возвращается значение x. В противном случае процедура SELECT вызывается рекурсивно, и с ее помощью выполняется поиск i-го в порядке возрастания элемента в нижней части, если i < k, или (i - k)-го в порядке возрастания элемента в верхней части, если i > k.

Пример задачи



Профессор работает консультантом в нефтяной компании, которая планирует провести магистральный трубопровод от восточного до западного края нефтяного месторождения с n скважинами. От каждой скважины к магистральному трубопроводу кратчайшим путем, в направлении на север или на юг, проведены рукава. Каким образом профессор может выбрать оптимальное расположение трубопровода (т.е. такое, при котором общая длина всех рукавов была бы минимальной) по заданным координатам скважин (x, y)? Покажите, что это можно сделать в течение времени, линейно зависящего от количества скважин.

План лекции

Сортировки (продолжение)	Медианы и порядковые статистики	Растущий массив	Очередь
25 минут	15 минут	25 минут	25 минут

Основные понятия

arr

1	2	4			

size = 3
capacity = 6
Если size становится больше capacity (массив переполняется), то capacity = capacity * 2;
vector->arr = (int*)calloc(sizeof(int)), capacity);

```
typedef int value_type;
typedef struct vector
{
    value_type *arr;
    size_t size;
    size_t capacity;
} vector;
```

Инициализация

```c
void vector_init(vector *v, size_t capacity)
{
    v->arr = (value_type *)calloc(capacity, sizeof(value_type));
    if (v->arr == NULL)
    {
        printf("Memory allocation failed");
        exit(1);
    }
    v->size = 0;
    v->capacity = capacity;
}
```

Размер массива

```
size_t vector_size(vector *v)
{
    return v->size;
}
size_t vector_capacity(vector *v)
{
    return v->capacity;
}
```

Изменение размера массива

```
void new_capacity(vector *v, size_t capacity)
{
    v->arr = (value_type *)realloc(v->arr, capacity *
sizeof(value_type));
    if (v->arr == NULL)
    {
        printf("Memory allocation failed");
        exit(1);
    }
    for (size_t i = v->capacity; i < capacity; i++)
    {
        v->arr[i] = 0;
    }
    v->capacity = capacity;
}
```

Добавление элемента в массив

```
void vector_push(vector *v, value_type data)
{
    if (v->size == v->capacity)
    {
        new_capacity(v, 2 * v->capacity);
    }
    v->arr[v->size] = data;
    v->size++;
}
void vector_set(vector *v, int index, value_type data)
{
    if (index >= v->capacity)
    {
        new_capacity(v, index + 1);
        v->size = index + 1;
    }
    v->arr[index] = data;
}
```

Удаление элемента из массива

```
int vector_delete(vector *v, size_t index)
{
    if (index >= v->capacity)
    {
        printf("Element not found!");
        return -1;
    }
    value_type a = v->arr[index];
    for (size_t i = index; i < v->size; i++)
    {
        v->arr[i] = v->arr[i + 1];
    }
    v->size--;
    return a;
}
```

Другие функции

```
value_type vector_get(vector *v, size_t index)
{
    if (index >= v->capacity)
    {
        printf("Element not found!");
        exit(1);
    }
    return v->arr[index];
}
void vector_clear(vector *v)
{
    if (v->arr != NULL)
        free(v->arr);
    v->size = 0;
    v->capacity = 0;
}
void vector_display(vector *v)
{
    if (v->arr != NULL)
        for (size_t i = 0; i < v->size; ++i)
            printf("%d ", v->arr[i]);
    printf("\n");
}
```

План лекции

Сортировки (продолжение)	Медианы и порядковые статистики	Растущий массив	Очередь
25 минут	15 минут	25 минут	25 минут

Введение

- Люди, движущиеся по эскалатору. Люди, которые первыми поднялись на эскалатор, будут первыми, сходящими с него.
- Люди, ожидающие автобус. Первый человек, стоящий в очереди, будет первым, заходящим в автобус.
- Люди, стоящие у кассового окна кинотеатра. Первый человек в очереди получит билет первым и, таким образом, будет первым, кто выйдет из нее.
- Багаж хранится на конвейерных лентах. Сумка, которая была помещена первой, будет первой, которая выйдет на другом конце.
- Автомобили, выстроившиеся в ряд у платного моста. Первая машина, которая доедет до моста, будет первой, кто уедет.

Очередь — это структура данных FIFO (первым пришел, первым вышел), в которой элемент, который вставлен первым, первым извлекается. Элементы в очереди добавляются на одном конце, называемом REAR, и удаляются с другого конца, называемого FRONT. Очереди могут быть реализованы с помощью массивов или связанных списков.

Представление очередей массивом

Очередь

12	9	7	18	14	36				
0	1	2	3	4	5	6	7	8	9

Очередь после вставки нового элемента

12	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

Очередь после удаления элемента

	9	7	18	14	36				
0	1	2	3	4	5	6	7	8	9

```c
#define MAX 10 // Changing this value will change length of array
int queue[MAX];
int front = -1, rear = -1;
```

Представление очередей массивом

```
void insert()
{
    int num;
    printf("\n Enter the number to be inserted in the queue : ");
    scanf("%d", &num);
    if (rear == MAX - 1)
        printf("\n OVERFLOW");
    else if (front == -1 && rear == -1)
        front = rear = 0;
    else
        rear++;
    queue[rear] = num;
}
```

Представление очередей массивом

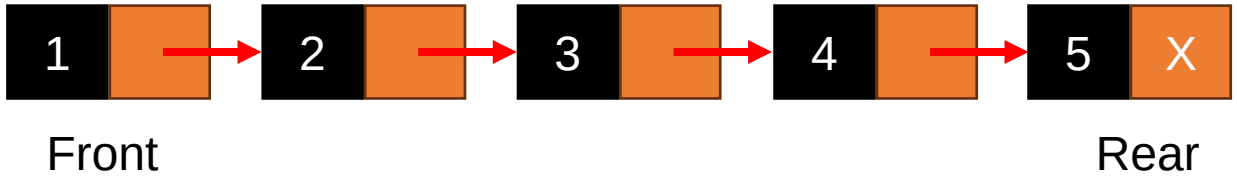
```
int delete_element()
{
    int val;
    if (front == -1 || front > rear)
    {
        printf("\n UNDERFLOW");
        return -1;
    }
    else
    {
        val = queue[front];
        front++;
        if (front > rear)
            front = rear = -1;
        return val;
    }
}
```

Связанное представление очередей

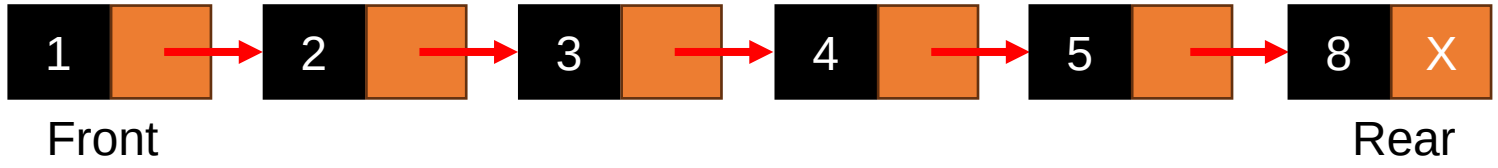
Хотя метод создания очереди массивом прост, его недостатком является то, что массив должен быть объявлен как имеющий некоторый фиксированный размер.
В случае, если очередь очень маленькая или ее максимальный размер известен заранее, то реализация очереди в виде массива дает эффективную реализацию. Но если размер массива невозможно определить заранее, используется другая альтернатива, т. е. связанное представление. Требование к памяти связанного представления очереди с n элементами составляет O(n), а типичное требование по времени для операций составляет O(1).

Связанное представление очередей

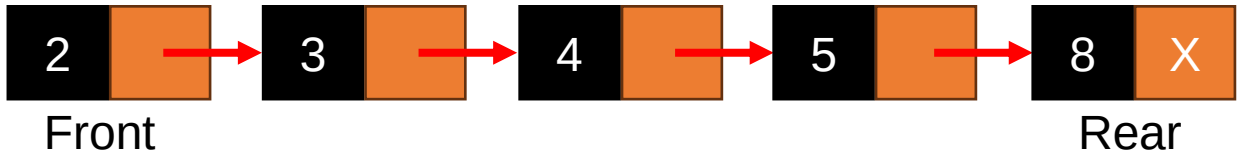
Очередь



Очередь после вставки нового элемента



Очередь после удаления элемента




```
struct node
{
    int data;
    struct node *next;
};
struct queue
{
    struct node *front;
    struct node *rear;
};
struct queue *q;
void create_queue(struct queue *q)
{
    q->rear = NULL;
    q->front = NULL;
}
```

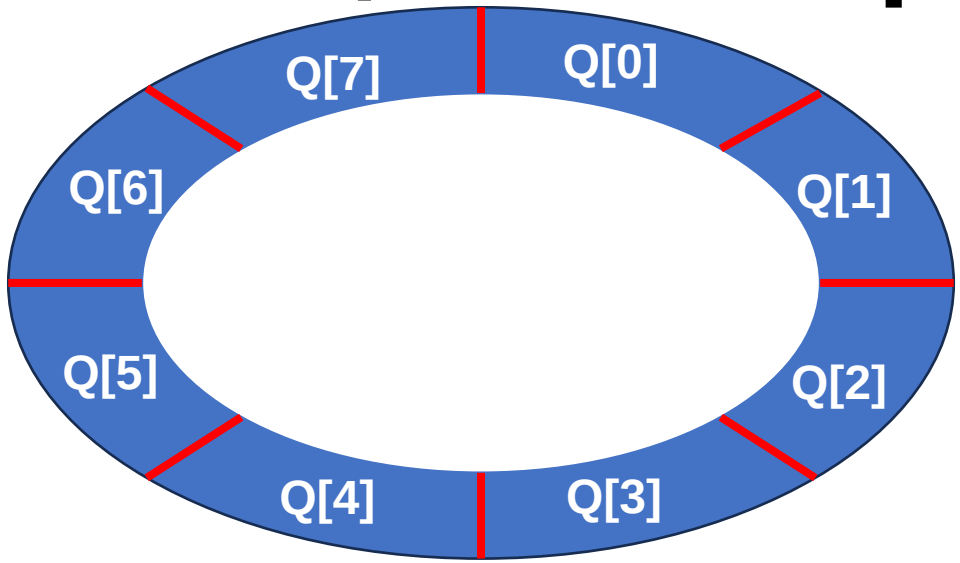
Связанное представление очередей

```
struct queue *insert(struct queue *q, int val)
{
    struct node *ptr;
    ptr = (struct node *)malloc(sizeof(struct node));
    ptr->data = val;
    if (q->front == NULL)
    {
        q->front = ptr;
        q->rear = ptr;
        q->front->next = q->rear->next = NULL;
    }
    else
    {
        q->rear->next = ptr;
        q->rear = ptr;
        q->rear->next = NULL;
    }
    return q;
}
```

Связанное представление очередей

```
struct queue *delete_element(struct queue *q)
{
    struct node *ptr;
    ptr = q->front;
    if (q->front == NULL)
        printf("\n UNDERFLOW");
    else
    {
        q->front = q->front->next;
        printf("\n The value being deleted is : %d", ptr->data);
        free(ptr);
    }
    return q;
}
```

Кольцевая очередь



Кольцевая очередь реализуется так же, как и линейная очередь. Единственное отличие будет в коде, который выполняет операции вставки и удаления.

```
#define MAX 10 // Changing this value will change length of array
int queue[MAX];
int front = -1, rear = -1;
```

Кольцевая очередь

```
void insert()
{
    int num;
    printf("\n Enter the number to be inserted in the queue : ");
    scanf("%d", &num);
    if (front == 0 && rear == MAX - 1)
        printf("\n OVERFLOW");
    else if (front == -1 && rear == -1)
    {
        front = rear = 0;
        queue[rear] = num;
    }
    else if (rear == MAX - 1 && front != 0)
    {
        rear = 0;
        queue[rear] = num;
    }
    else
    {
        rear++;
        queue[rear] = num;
    }
}
```

Кольцевая очередь

```
int delete_element()
{
    int val;
    if (front == -1 && rear == -1)
    {
        printf("\n UNDERFLOW");
        return -1;
    }
    val = queue[front];
    if (front == rear)
        front = rear = -1;
    else
    {
        if (front == MAX - 1)
            front = 0;
        else
            front++;
    }
    return val;
}
```

Deque

Deque (произносится как «дек») — это список, в котором элементы могут быть вставлены или удалены с любого конца. Он также известен как связанный список «голова-хвост», потому что элементы могут быть добавлены или удалены как с переднего (голова), так и с заднего (хвост) конца. Однако ни один элемент не может быть добавлен или удален из середины. В памяти компьютера дека реализована с использованием либо кольцевого массива, либо кольцевого двусвязного списка. В деке поддерживаются два указателя, LEFT и RIGHT, которые указывают на любой из концов деки. Элементы в деке простираются от ЛЕВОГО конца до ПРАВОГО конца, и поскольку она кольцевая, за Dequeue[N–1] следует Dequeue[0].

```
#define MAX 10
int deque[MAX];
int left = –1, right = –1;
```

Deque

Существует два варианта двухсторонней очереди. Они включают

Ограниченная входная очередь

В этой очереди вставки могут быть выполнены только на одном из концов, а удаления — на обоих концах.

Ограниченная выходная очередь

В этой очереди удаления могут быть выполнены только на одном из концов, а вставки — на обоих концах.

			18	14	36	54	63		
0	1	2	3	4	5	6	7	8	9
			left				right		

42	56						63	27	18
0	1	2	3	4	5	6	7	8	9
	right						left		

Deque

```
void insert_right() {
    int val;
    printf("\n Enter the value to be added:");
    scanf("%d", &val);
    if ((left == 0 && right == MAX - 1) || (left == right + 1))
    {
        printf("\n OVERFLOW");
        return;
    }
    if (left == -1) /* if queue is initially empty */
    {
        left = 0;
        right = 0;
    }
    else
    {
        if (right == MAX - 1) /*right is at last position of queue */
            right = 0;
        else
            right = right + 1;
    }
    deque[right] = val;
}
```

Deque

```
void delete_right()
{
    if (left == -1)
    {
        printf("\n UNDERFLOW");
        return;
    }
    printf("\n The element deleted is : %d", deque[right]);
    if (left == right) /*queue has only one element*/
    {
        left = -1;
        right = -1;
    }
    else
    {
        if (right == 0)
            right = MAX - 1;
        else
            right = right - 1;
    }
}
```

Приоритетные очереди

Приоритетные очереди — это структура данных, в которой каждому элементу назначается приоритет. Приоритет элемента будет использоваться для определения порядка, в котором элементы будут обрабатываться. Общие правила обработки элементов очереди приоритетов:

- Элемент с более высоким приоритетом обрабатывается раньше элемента с более низким приоритетом.
- Два элемента с одинаковым приоритетом обрабатываются по принципу «первым пришел — первым обслужен» (FCFS).

Пример:

В случае, если нам нужно запустить два процесса одновременно, где один процесс связан с онлайн-бронированием заказов, а второй — с печатью данных о запасах, то, очевидно, онлайн-бронирование важнее и должно быть выполнено первым.

Приоритетные очереди

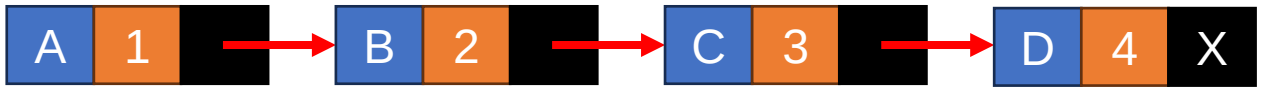
Скорость

Существует два способа реализации приоритетной очереди. Мы можем либо использовать отсортированный список для хранения элементов, чтобы при извлечении элемента в очереди не приходилось искать элемент с наивысшим приоритетом, либо мы можем использовать несортированный список, чтобы вставки всегда выполнялись в конце списка. Каждый раз, когда элемент должен быть удален из списка, элемент с наивысшим приоритетом будет найден и удален. В то время как отсортированный список занимает O(n) времени для вставки элемента в список, требуется только O(1) времени для удаления элемента. Напротив, несортированный список займет O(1) времени для вставки элемента и O(n) времени для удаления элемента из списка. На практике оба эти метода неэффективны, и обычно применяется смесь этих двух подходов, которая занимает примерно O(log n) времени или меньше.

Приоритетные очереди

Связанное представление приоритетной очереди

В памяти компьютера приоритетная очередь может быть представлена с помощью массивов или связанных списков. Когда приоритетная очередь реализована с помощью связанного списка, то каждый узел списка будет иметь три части: (a) информационная или часть данных, (b) номер приоритета элемента и (c) адрес следующего элемента. Если мы используем отсортированный связанный список, то элемент с более высоким приоритетом будет предшествовать элементу с более низким приоритетом.




```
struct node
{
    int data;
    int priority;
    struct node *next;
}
struct node *start=NULL;
```

Массивы приоритетных очередей

FRONT	REAR
3	3
1	3
4	5
4	1

	1	2	3	4	5
1			A		
2	B	C	D		
3				E	F
4	I			G	H

Матрица очереди приоритетов

FRONT	REAR
3	3
1	3
4	1
4	1

	1	2	3	4	5
1			A		
2	B	C	D		
3	R			E	F
4	I			G	H

Матрица очереди приоритетов после вставки нового элемента

Приоритетные очереди

```
struct node *insert(struct node *start) {
    int val, pri;
    struct node *ptr, *p;
    ptr = (struct node *)malloc(sizeof(struct node));
    printf("\n Enter the value and its priority : ");
    scanf("%d %d", &val, &pri);
    ptr->data = val;
    ptr->priority = pri;
    if (start == NULL || pri < start->priority)
    {
        ptr->next = start;
        start = ptr;
    }
    else
    {
        p = start;
        while (p->next != NULL && p->next->priority <= pri)
            p = p->next;
        ptr->next = p->next;
        p->next = ptr;
    }
    return start;
}
```

Приоритетные очереди

```c
struct node *delete(struct node *start)
{
    struct node *ptr;
    if (start == NULL)
    {
        printf("\n UNDERFLOW");
        return;
    }
    else
    {
        ptr = start;
        printf("\n Deleted item is: %d", ptr->data);
        start = start->next;
        free(ptr);
    }
    return start;
}
```

Несколько очередей

Когда мы реализуем очередь с использованием массива, размер массива должен быть известен заранее. Если очереди выделено меньше места, то будут часто возникать условия переполнения. Чтобы справиться с этой проблемой, код придется изменить, чтобы перераспределить больше места для массива.

Если мы выделим большой объем места для очереди, это приведет к чистой трате памяти. Таким образом, существует компромисс между частотой переполнений и выделенным пространством. Поэтому лучшим решением для решения этой проблемы является наличие нескольких очередей или наличие более одной очереди в одном массиве достаточного размера.

Применение очередей

- Очереди широко используются в качестве списков ожидания для одного общего ресурса, такого как принтер, диск, ЦП.
- Очереди используются для асинхронной передачи данных (данные не обязательно принимаются с той же скоростью, что и отправляются) между двумя процессами (буферами ввода-вывода), например, каналами, файловым вводом-выводом, сокетами.
- Очереди используются в качестве буферов на MP3-плеерах и портативных CD-плеерах, плейлистах iPod.
- Очереди используются в плейлистах для музыкального автомата для добавления песен в конец, воспроизведения с начала списка.
- Очереди используются в операционной системе для обработки прерываний. При программировании системы реального времени, которая может быть прервана, например, щелчком мыши, необходимо обрабатывать прерывания немедленно, прежде чем приступать к выполнению текущего задания. Если прерывания должны обрабатываться в порядке поступления, то очередь FIFO является подходящей структурой данных.