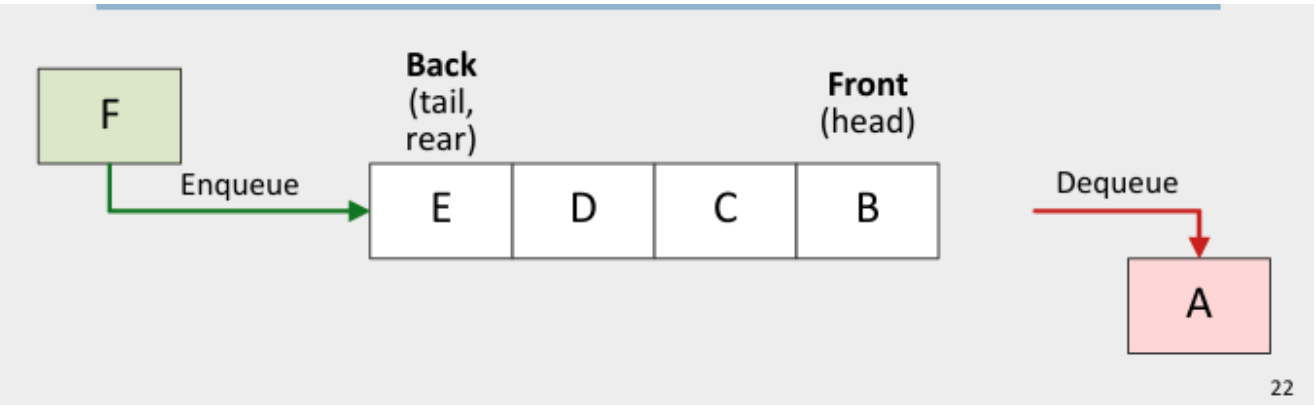
Очередь. Способы реализации очереди. Основные операции и их вычислительная сложность. Реализация очереди на основе циклического массива. Двухсторонняя очередь (дек, deque).

Очередь

- **Очередь** *(queue)* - структура данных для хранения элементов *(контейнер)*
- Дисциплина доступа к элементам:
  - "Первым пришел – первым вышел" *(First In – First Out, FIFO)*
  - Элементы добавляются в хвост *(tail)*, извлекаются с головы *(head)*



- Очереди широко используется в алгоритмах обработки данных:
  – очереди печати
  – буфер ввода с клавиатуры
  – алгоритмы работы с графами

Основные операции:

Операция	Описание
`Enqueue(q, x)`	Добавляет элемент *x* в очередь *q*
`Dequeue(q)`	Извлекает элемент из очереди *q*
`Size(q)`	Возвращает количество элементов в очереди *q*
`Clear(q)`	Очищает очередь *q*

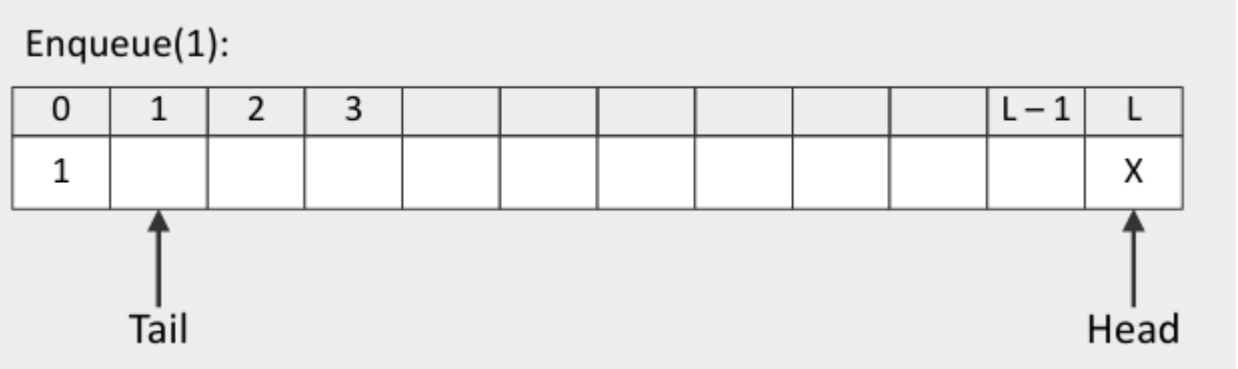
Вычислительная сложность для всех операций: **Θ(1)**

Способы реализации очереди

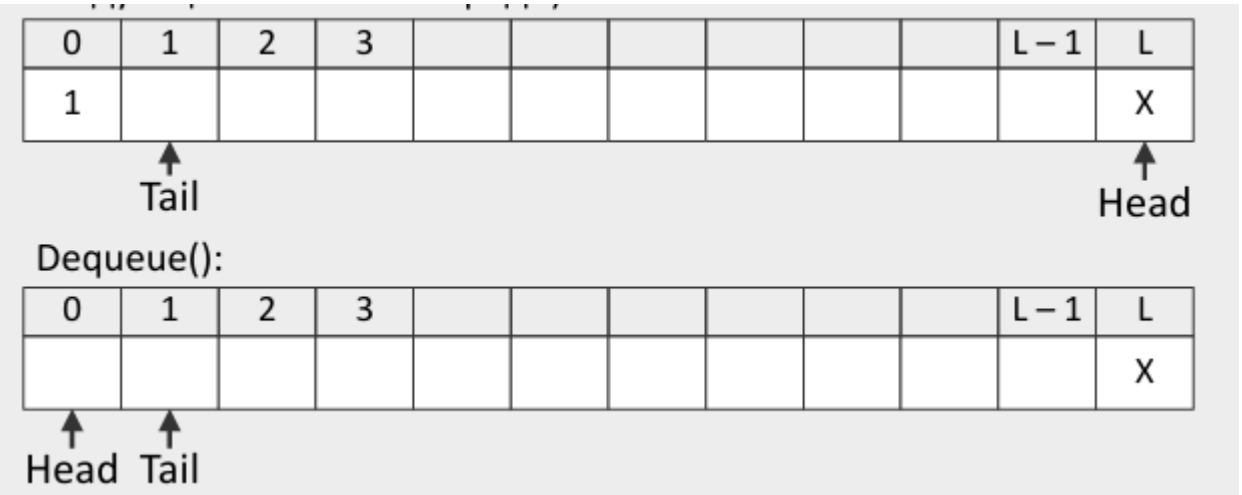
- На основе связных списков– Длина очереди ограничена лишь объемом доступной памяти.
- На основе статических массивов– Длина очереди фиксирована *(задана максимальная длина).*

Реализация очереди на основе циклических массивов

- Элементы очереди хранятся в массиве фиксированной длины [0, L – 1]
- Массив логически представляется в виде кольца *(circular buffer)*
- В пустой очереди Tail = 0, Head = L
- При добавлении элемента в очередь значение Tail циклически увеличивается на 1 (сдвигается на следующую свободную позицию)
- *Если Head = Tail + 1, то очередь переполнена*



- При **удалении** возвращается элемент с номером Head % L
- Значение Head циклически увеличивается на 1 (*указывает на следующий элемент очереди)*




```
struct queue {
    int *v;
    int head;
    int tail;
    int size;
    int maxsize;
};
```

Создание пустой очереди

```
struct queue *queue_create(int maxsize)
{
    struct queue *q = malloc(sizeof(*q));
    if (q != NULL) {
        q->v = malloc(sizeof(int) * (maxsize + 1));
        if (q->v == NULL) {
            free(q);
            return NULL;
        }
        q->maxsize = maxsize;
        q->size = 0;
        q->head = maxsize + 1;
        q->tail = 0;
    }
    return q;
}
```

Удаление очереди

```
void queue_free(struct queue *q)
{
    free(q->v);
    free(q);
}
```

Вернуть размер очереди

```
int queue_size(struct queue *q)
{
    return q->size;
}
```

Добавление элемента в очередь

```
int queue_enqueue(struct queue *q, int value)
{
    if (q->head == q->tail + 1) {
        fprintf(stderr, "queue: Queue overflow\n");
        return -1;
    }
    q->v[q->tail++] = value;
    q->tail = q->tail % (q->maxsize + 1);
    q->size++;
    return 0;
}
```

Получение элемента из очереди

```
int queue_dequeue(struct queue *q)
{
    if (q->head % (q->maxsize + 1) == q->tail) {
    // Очередь пуста
    fprintf(stderr, "queue: Queue is empty\n");
    return -1;
    }
    q->head = q->head % (q->maxsize + 1);
    q->size--;
    return q->v[q->head++];
}
```

Двухсторонняя очередь (дек, deque).

Двухсторонняя очередь (Deque, Double-Ended Queue) — это абстрактный тип данных *(АТД)*, который расширяет функциональность обычной очереди, позволяя добавлять и удалять элементы **с обоих концов** *(головы и хвоста)*.

Типы двухсторонних очередей

1. Двухсторонняя очередь с ограниченной вставкой

В этой двухсторонней очереди вставка элементов осуществляется лишь с одной стороны очереди. Удаление все так же доступно с обеих сторон.

2. Двухсторонняя очередь с ограниченным удалением

В этой двухсторонней очереди удаление элементов осуществляется лишь с одной стороны очереди. Вставка так же доступна с обеих сторон.

Где используется?

1. **Алгоритмы**:
   - Скользящее окно (например, в задачах с подмассивами).
   - Алгоритм поиска в ширину (BFS) с возможностью ветвления.
   - Палиндромные проверки (можно удалять символы с обоих концов).
2. **Системы**:
   - История действий в программах (undo/redo).
   - Планировщики задач с приоритетами с обоих концов.
