Queue

Основная теория

В чем суть структуры

Queue (очередь) - структура данных, работающая по принципу **FIFO (First In First Out)**. Главные преимущества данной структуры - быстрое обращение к элементу массива, которые вошел в него позже других.

На практике мы использовали очереди при обходе графа в ширину. (BFS)

Идейная реализация

Добавляем элементы в конец очереди, а забираем сначала. Для этого объект очереди содержит два указателя (или индекса) на ее начало и конец.

Реализация через массив
Для реализации через массив нам необходимо хранить следующие значения: `head`, `tail` - указатели на начало и конец очереди, `capacity` - текущее количество элементов в очереди, `size` - размер очереди, `data` - массив со значениями элементов в очереди.

Проверка на простоту: `queue.capacity == 0`, проверка на полноту - `queue.capacity == queue.size`.

Добавление в очередь: проверяем очередь на переполнение, если все хорошо, то добавляем новый элемент по индексу `queue.tail` и прибавляем к индексу хвоста +1 (`queue.tail = queue.tail + 1) % queue.size`)

Удаление элементов: проверяем на переполнение очереди, если все хорошо, то двигаем индекс головы на 1 (`queue.head = (queue.head + 1) % queue.size`)

Реализация через список

Для реализации очереди нам понадобится две структуры: структура для элемента очереди (поля структуры: `next`, `prev` - указатели на предыдущий и следующий элементы очереди, `value` - значение элемента), а также структура самой очереди (поля структуры: `head`, `tail` - указатели на первый и последний элементы очереди, `capacity` - количество элементов в очереди)

Добавление элемента: если очереди пуста, то делаем головой и хвостом очереди новый элемент, иначе присваиваем указателю `tail.prev` значение нового элемента (`tail.prev = newNode`).

Удаление элемента: если очередь не пуста, то меняем голову очереди `queue.head = queue.head.prev`. (Сборщик мусора в **Go** сам почистит лишние объекты из памяти, поэтому достаточно лишь переопределить указатели).

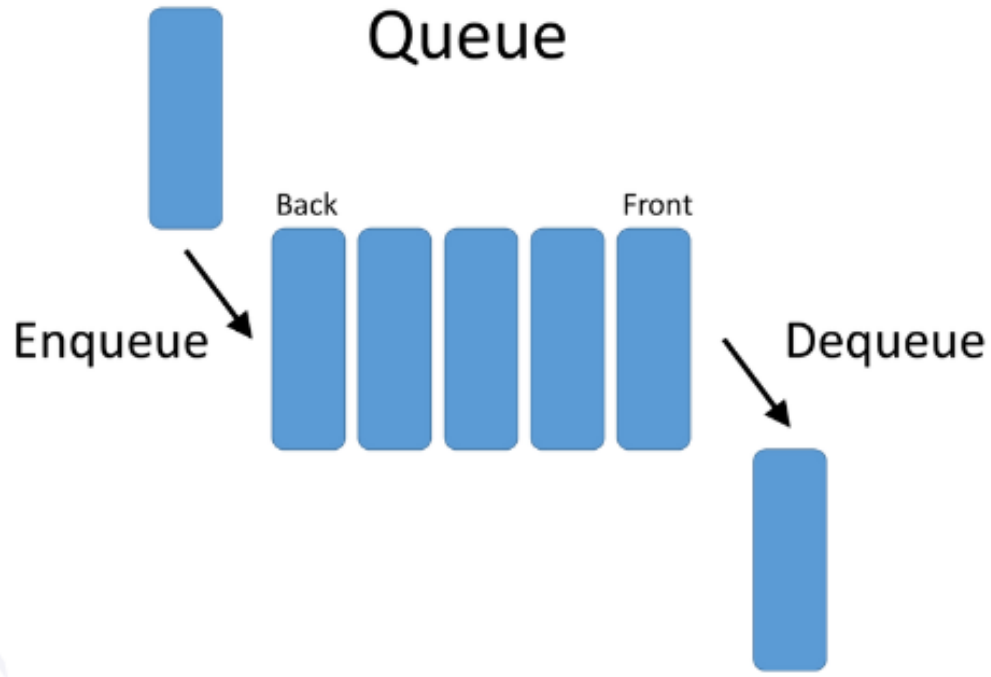
> **Важно**
>
> Индекс головы никогда на зайдет за индекс хвоста. Это связано с особенностью добавления элементов и проверки очереди на пустоту и переполнение.

Операции над структурой и асимптотическая сложность

Операции:

- `Enqueue()` - добавление в очередь за **O(1)**. (в любом из реализаций)
- `Dequeue()` - удаление элемента из очереди за **O(1)** (в каждой из реализаций)
- `IsEmpty()` - проверка очереди на пустоту
- `IsFull()` - проверка на переполнение (неактуально для очереди на списках)

Иллюстрации



Примеры кода

Реализация на массиве

Инициализация структуры

```
type QueueArray[T any] struct {
    data     []T
    head     int
    tail     int
    size     int
    capacity int
}
```

```go
func NewQueueArray[T any](size int) *QueueArray[T] {
    return &QueueArray[T]{
        data: make([]T, size),
        head: 0,
        tail: 0,
        size: size,
    }
}
```

Методы очереди:

```go
func (q *QueueArray[T]) IsEmpty() bool {
    return q.capacity == 0
}

func (q *QueueArray[T]) IsFull() bool {
    return q.capacity == q.size
}

func (q *QueueArray[T]) Enqueue(value T) error {
    if q.IsFull() {
        return errors.New("queue is full")
    }

    q.data[q.tail] = value
    q.tail = (q.tail + 1) % q.size
    q.capacity++
    return nil
}

func (q *QueueArray[T]) Dequeue() (T, error) {
    var retval T
    if q.IsEmpty() {
        return retval, errors.New("queue is empty")
    }
    retval = q.data[q.head]
    q.head = (q.head + 1) % q.size
    q.capacity--
    return retval, nil
}
```

Очередь на связном списке

Инициализация структуры:

```go
type QueueListNode[T any] struct {
    value T
    next  *QueueListNode[T]
    prev  *QueueListNode[T]
}
```

```go
type QueueList[T any] struct {
    head     *QueueListNode[T]
    tail     *QueueListNode[T]
    capacity int
}

func NewQueueList[T any]() *QueueList[T] {
    return &QueueList[T]{
        head:     nil,
        tail:     nil,
        capacity: 0,
    }
}
```

Методы структуры

```go
func (q *QueueList[T]) IsEmpty() bool {
    return q.capacity == 0
}

func (q *QueueList[T]) Enqueue(value T) {
    newNode := &QueueListNode[T]{
        value: value,
        next:  nil,
        prev:  nil,
    }

    if q.IsEmpty() {
        q.capacity++
        q.head = newNode
        q.tail = newNode
        return
    }

    q.capacity++
    q.tail.prev = newNode
    newNode.next = q.tail

    q.tail = newNode
}

func (q *QueueList[T]) Dequeue() (T, error) {
    var retval T

    if q.IsEmpty() {
        return retval, errors.New("queue is empty")
    }

    q.capacity--
    retval = q.head.value
    q.head = q.head.prev
```

```
    return retval, nil
}
```

Ресурсы

- Кормен - "Алгоритмы: построение и анализ" стр. 266
- Queue (abstract data type) - Wikipedia
- Queue Data Structure and Implementation in Java, Python and C/C++ (programiz.com)