Билет 23. Линейные контейнеры

Билет 23. Линейные контейнеры

Основная идея: Элементы располагаются последовательно друг за другом. Разные правила доступа = разные структуры данных

Линейные контейнеры — структуры данных, где элементы располагаются в линейной последовательности. Отличаются правилами доступа и модификации.

Обзор линейных контейнеров

Структура	**Принцип работы**	**Основные операции**
Динамический массив	Автоматически расширяемый массив	get/set по индексу, add, remove
Стек (Stack)	Last In - First Out	push, pop, peek
Очередь (Queue)	First In - First Out	enqueue, dequeue, peek
Дек (Deque)	Двусторонняя очередь	addFront, addBack, removeFront, removeBack
Циклический буфер	Фиксированный размер, циклическое использование	enqueue, dequeue, isFull, isEmpty

1. Динамический массив (ArrayList, Vector)

Принцип работы

- Обычный массив, но с автоматическим расширением

- При заполнении создаётся новый массив большего размера (обычно $\times 1.5$ или $\times 2$)
- Старые элементы копируются в новый массив

Сложность операций

- **Доступ по индексу:** O(1)
- **Вставка в конец:** O(1) амортизированно
- **Вставка в начало/середину:** O(n)
- **Удаление:** O(n) в худшем случае

2. Стек (LIFO)

"Last in - First out"

Основные операции

- **push(x)** - добавить элемент на вершину
- **pop()** - удалить и вернуть верхний элемент
- **peek()** - посмотреть верхний элемент без удаления
- **isEmpty()** - проверка на пустоту

Сложность: O(1) для всех операций

3. Очередь (FIFO)

"First in - First out"

Основные операции

- **enqueue(x)** - добавить в конец очереди
- **dequeue()** - удалить и вернуть первый элемент

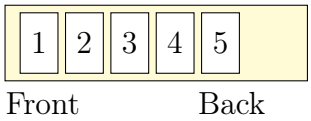
- **front()** - посмотреть первый элемент
- **isEmpty()** - проверка на пустоту

Реализации

- **На массиве:** O(1) амортизированно
- **На связном списке:** O(1) всегда

4. Дек (Двусторонняя очередь)

Можно добавлять/удалять с обоих концов



Основные операции

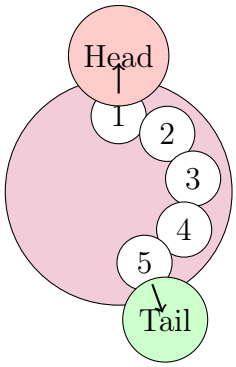
- **addFront(x), addBack(x)** - добавление
- **removeFront(), removeBack()** - удаление
- **getFront(), getBack()** - получение без удаления

Реализации

- **На двусвязном списке:** O(1) для всех операций
- **На циклическом массиве:** O(1) амортизированно

5. Циклический буфер (Кольцевой буфер)

Фиксированный размер, циклическое использование



Основные операции

- **enqueue(x)** - добавить (если есть место)
- **dequeue()** - удалить и вернуть старейший элемент
- **isFull(), isEmpty()** - проверки
- **peek()** - посмотреть следующий элемент

Преимущества

- **Фиксированная память** - не нужно расширение
- **Эффективность** - O(1) для всех операций
- **Предсказуемость** - известны границы памяти

Циклический буфер

Основная идея: Буфер фиксированного размера, который "зацикливается когда доходим до конца, начинаем с начала!

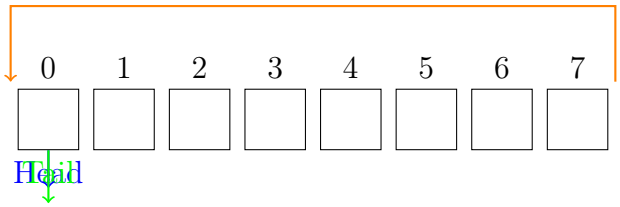
Что такое циклический буфер?

Циклический буфер (кольцевой буфер, circular buffer) — структура данных, использующая единственный буфер фиксированного размера, как будто бы после последнего элемента сразу же снова идет первый.

Основные компоненты

- **Буфер:** Массив фиксированного размера
- **Head (голова):** Указатель на начало данных (откуда читаем)
- **Tail (хвост):** Указатель на конец данных (куда пишем)
- **Size:** Текущее количество элементов
- **Capacity:** Максимальная вместимость

Принцип работы



Основные операции

1. Добавление элемента (enqueue)

```
enqueue(element):
    if isFull(): return false
    buffer[tail] = element
    tail = (tail + 1) % capacity
    size++
    return true
```

2. Извлечение элемента (dequeue)

```
dequeue():
    if isEmpty(): return null
    element = buffer[head]
    head = (head + 1) % capacity
    size--
    return element
```

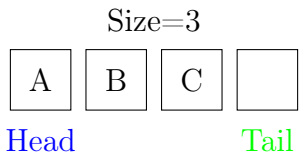
3. Проверки

```
isEmpty(): return size == 0
isFull(): return size == capacity
```

Пошаговый пример

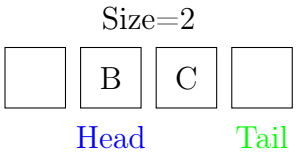
Создаем буфер размером 4: [, , ,] Head=0, Tail=0, Size=0

Шаг 1: Добавляем A, B, C



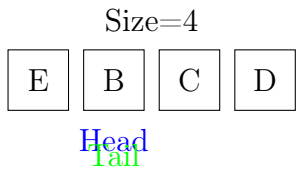
Состояние: [A, B, C, _] Head=0, Tail=3, Size=3

Шаг 2: Извлекаем A



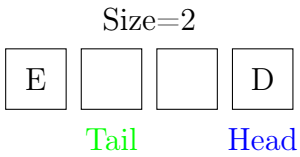
Состояние: [_ , B, C, _] Head=1, Tail=3, Size=2

Шаг 3: Добавляем D, E



Состояние: [E, B, C, D] Head=1, Tail=1, Size=4 **(полный!)**

Шаг 4: Извлекаем B, C



Состояние: [E, _ , _ , D] Head=3, Tail=1, Size=2

Реализация на Python

```
class CircularBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = [None] * capacity
        self.head = 0
        self.tail = 0
        self.size = 0

    def enqueue(self, item):
        if self.is_full():
            return False
        self.buffer[self.tail] = item
        self.tail = (self.tail + 1) % self.capacity
        self.size += 1
        return True

    def dequeue(self):
```

```
        if self.is_empty():
            return None
        item = self.buffer[self.head]
        self.head = (self.head + 1) % self.capacity
        self.size -= 1
        return item

    def is_empty(self):
        return self.size == 0

    def is_full(self):
        return self.size == self.capacity

    def __str__(self):
        elements = []
        current = self.head
        for _ in range(self.size):
            elements.append(self.buffer[current])
            current = (current + 1) % self.capacity
        return f"Buffer: {elements}"
```

Пример использования

```
# Создаем буфер на 3 элемента
cb = CircularBuffer(3)

print("Добавляем 1, 2, 3:")
cb.enqueue(1)
cb.enqueue(2)
cb.enqueue(3)
print(cb)  # Buffer: [1, 2, 3]

print("Пытаемся добавить 4 (места нет):")
success = cb.enqueue(4)
print(f"Успешно: {success}")  # Успешно: False

print("Извлекаем два элемента:")
```

```
print(cb.dequeue())  # 1
print(cb.dequeue())  # 2
print(cb)  # Buffer: [3]

print("Добавляем 4 и 5:")
cb.enqueue(4)
cb.enqueue(5)
print(cb)  # Buffer: [3, 4, 5]
```

Варианты реализации

1. С счетчиком размера (рекомендуется)

```
# Как в примере выше - используем переменную size
# Просто и надежно
```

2. Без счетчика размера

```
# Определяем пустоту/полноту по позициям head/tail
is_empty(): return head == tail and buffer[head] is None
is_full(): return head == tail and buffer[head] is not None
```

3. С перезаписью старых данных

```
# Когда буфер полон, перезаписываем самые старые данные
def enqueue_force(self, item):
    if self.is_full():
        self.dequeue()  # Удаляем самый старый элемент
    self.enqueue(item)
```

1 Ключевая идея:

- Мы не перезаписываем начало, а используем ВСЮ доступную память циклически:
- Есть "окно"данных между head и tail

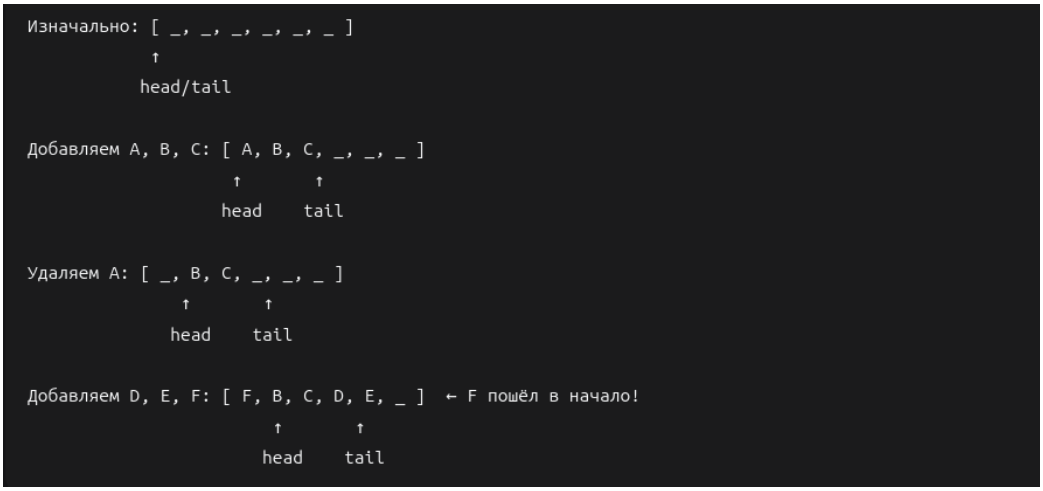


Рис. 1: Cycler Buffer

- Когда tail доходит до конца массива, он "перепрыгивает"в начало
- Head тоже движется - освобождая место для новых данных
- В любой момент времени буфер содержит последовательные данные, просто они могут "оборачиваться"вокруг границ массива