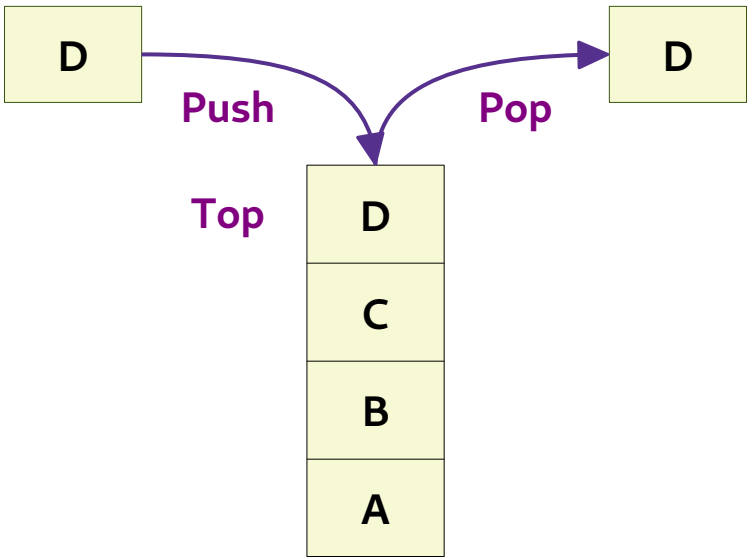
Лекция 6.
Стеки и очереди

Даниил Михайлович Берлизов
Старший преподаватель Кафедры вычислительных систем СибГУТИ
E-mail: sillyhat34@gmail.com

Курс «Структуры и алгоритмы обработки данных»
Весенний семестр, 2021 г.

Стек (stack)

- **Стек** (stack) — это структура данных для хранения элементов с дисциплиной доступа «последним пришёл — первым вышел» (Last In — First Out, LIFO)
- Элементы помещаются и извлекаются из головы стека (top)



Подходы к реализации стека

- **На основе связных списков**

  Длина стека ограничена объёмом доступной памяти

- **На основе статических массивов**

  Длина стека фиксирована (задана его максимальная длина — количество элементов в массиве)

Реализация стека на основе связных списков

- Элементы стека хранятся в односвязном списке (singly linked list)
- Операции добавления (push) и удаления (pop) выполняются за время ***O*(1)**

Реализация стека на основе связных списков

```
#include <stdio.h>
#include <stdlib.h>

#include “llist.h”          /* Реализация связного списка */

struct stack {
    struct listnode *top;  /* Вершина стека */
    int size;
};
```

Реализация стека на основе связных списков

```
/*
 * Фрагмент файла llist.c
 */

struct listnode {
    int value;          /* Значение элемента в стеке */
    struct listnode *next;
};
```

Создание пустого стека

```
struct stack *stack_create()
{
    struct stack *s = malloc(sizeof(*s));
    if (s != NULL) {
        s->size = 0;
        s->top = NULL;
    }
    return s;
}
```

$T_{Create} = O(1)$

Удаление стека

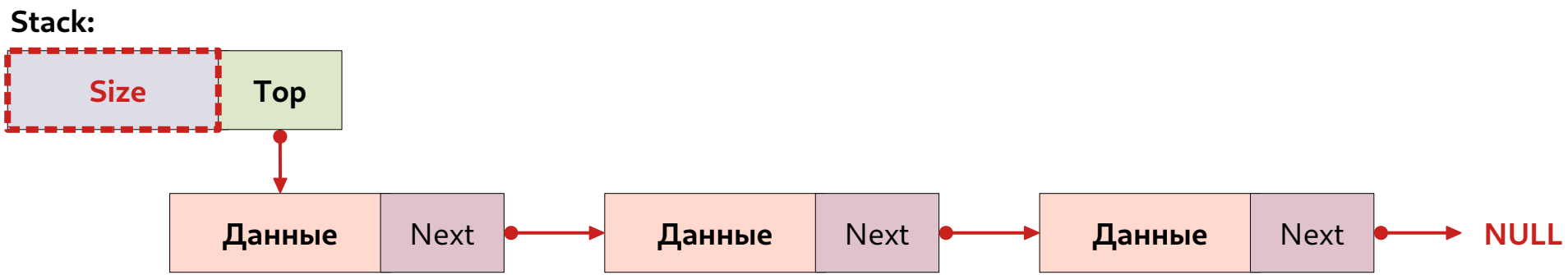
```
void stack_free(struct stack *s)
{
    while (s->size > 0)
        stack_pop(s);
    free(s);
}
```

$T_{Free} = O(n)$

Получение размера стека

```
int stack_size(struct stack *s)
{
    return s->size;
}
```

$T_{Size} = O(1)$



Реализация стека на основе связных списков

```
int main()
{
    struct stack *s;
    s = stack_create();
    printf("Stack size: %d\n", stack_size(s));

    stack_free(s);
    return 0;
}
```

Добавление элемента в стек (push)

```
int stack_push(struct stack *s, int value)
{
    s->top = list_addfront(s->top, value);
    if (s->top == NULL) {
        fprintf(stderr, “Stack overflow\n”);
        return -1;
    }
    s->size++;
    return 0;
}
```

$T_{Push} = O(1)$

Удаление элемента из стека (pop)

```
int stack_pop(struct stack *s)
{
    struct listnode *next;
    int value;
    if (s->top == NULL) {
        fprintf(stderr, “Stack underflow\n”);
        return -1;
    }
    next = s->top->next;
    value = s->top->value;
    free(s->top);
    s->top = next;
    s->size--;
    return value;
}
```

$T_{Pop} = O(1)$

Реализация стека на основе связных списков

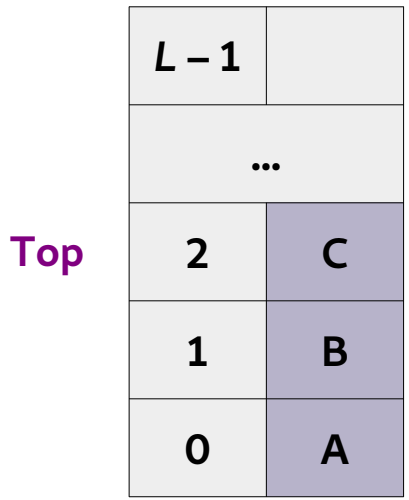
```
int main()
{
    struct stack *s;
    int i, value;
    s = stack_create();
    for (i = 1; i <= 10; i++)
        stack_push(s, i);

    for (i = 1; i <= 11; i++) {
        value = stack_pop(s, i);
        printf("pop: %d\n", value);
    }
    stack_free(s);
    return 0;
}
```

```
pop: 10
pop: 9
pop: 8
pop: 7
pop: 6
pop: 5
pop: 4
pop: 3
pop: 2
pop: 1
pop: -1
```

Реализация стека на основе массива

- Элементы стека хранятся в массиве фиксированной длины *L*
- Операции добавления (push) и удаления (pop) выполняются за время ***O*(1)**



Реализация стека на основе массива

```c
#include <stdio.h>
#include <stdlib.h>

struct stack {
    int *a;
    int top;
    int size;
    int maxsize;
};
```

Создание пустого стека

```
struct stack *stack_create(int maxsize)
{
    struct stack *s = malloc(sizeof(*s));
    if (s != NULL) {
        s->a = malloc(sizeof(int) * maxsize);
        if (s->a == NULL) {
            free(s);
            return NULL;
        }
        s->size = 0;
        s->top = 0;
        s->maxsize = maxsize;
    }
    return s;
}
```

$T_{Create} = O(1)$

Удаление стека. Получение размера стека

```
void stack_free(struct stack *s)
{
    free(s->a);
    free(s);
}

int stack_size(struct stack *s)
{
    return s->size;
}
```

$T_{Free} = O(1)$

$T_{Size} = O(1)$

Добавление элемента в стек (push)

```
int stack_push(struct stack *s, int value;
{
    if (s->top < s->maxsize) {
        s->a[s->top++] = value;
        s->size++;
    } else {
        fprintf(stderr, “Stack overflow\n”);
        return -1;
    }
    return 0;
}
```

$T_{Push} = O(1)$

Удаление элемента из стека (pop)

```
int stack_pop(struct stack *s)
{
    if (s->top == 0) {
        fprintf(stderr, “Stack underflow\n”);
        return -1;
    }
    s->size--;
    return s->a[--s->top];
}
```

$T_{Pop} = O(1)$

Очередь (queue)

- **Очередь (queue)** — это структура данных для хранения элементов (контейнер) с дисциплиной доступа «первым пришёл — первым вышел» (First In — First Out, FIFO)
- Элементы добавляются в хвост (tail), извлекаются из головы (head)

Очередь (queue)

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

Очередь (queue)

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

Подходы к реализации очереди

- **На основе связных списков**

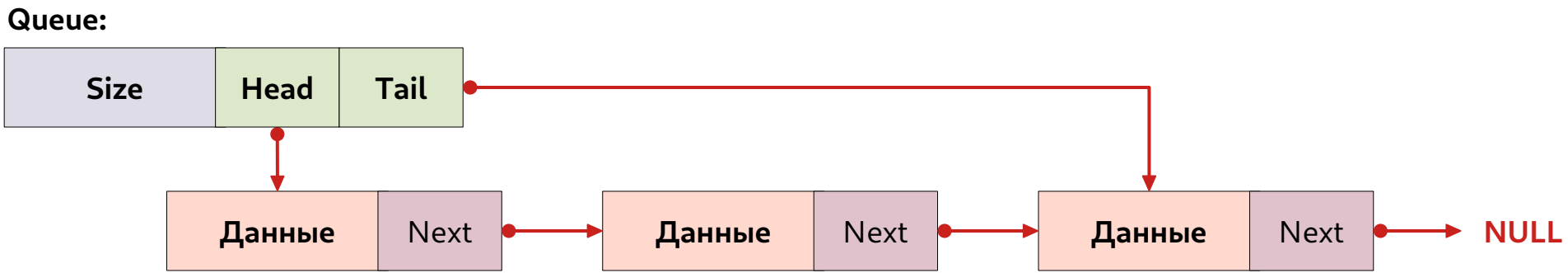
  Длина очереди ограничена лишь объёмом доступной памяти

- **На основе статических массивов**

  Длина очереди фиксирована (задана максимальная длина)

Реализация очереди на основе связных списков

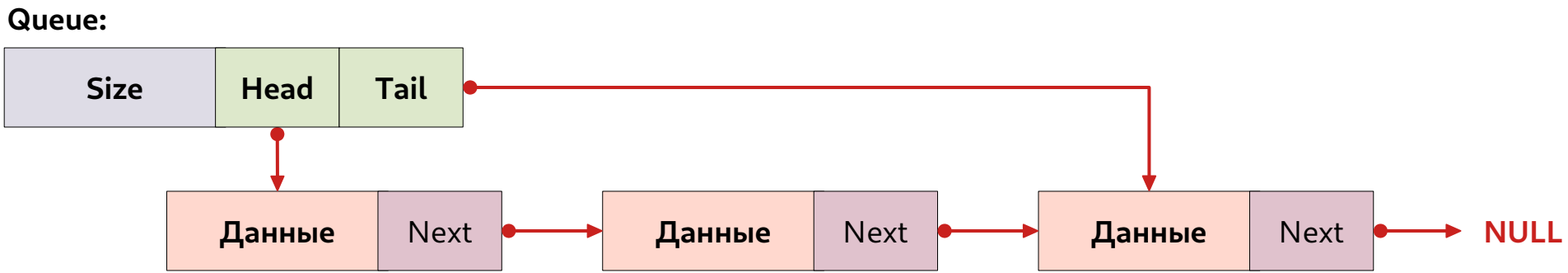
- Элементы очереди хранятся в односвязном списке (singly linked list)
- Для быстрого (за время ***O*(1)**) добавления и извлечения элементов из списка поддерживается указатель на последний элемент (tail)
- Новые элементы добавляются в конец списка



Реализация очереди на основе связных списков

- **Преимущества**: длина очереди ограничена лишь объёмом доступной памяти

- **Недостатки** (по сравнению с реализацией на основе массивов): работа с очередью немного медленнее, требуется больше памяти для хранения одного элемента



Реализация очереди на основе связных списков

```c
#include <stdio.h>
#include <stdlib.h>

#include “llist.h”

struct queue {
    struct listnode *head;
    struct listnode *tail;
    int size;
};
```

Создание пустой очереди

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

$T_{Create} = O(1)$

Удаление очереди. Получение размера очереди

```
void queue_free(struct queue *q)
{
    while (q->size > 0)
        queue_dequeue(q);
    free(q);
}

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

$T_{Free} = O(n)$

$T_{Size} = O(1)$

Добавление элемента в очередь (enqueue)

```
void queue_enqueue(struct queue *q, int value)
{
    struct listnode *oldtail = q->tail;
    q->tail = list_createnode(value);
    if (q->head == NULL)        /* Очередь пуста */
        q->head = q->tail;
    else
        oldtail->next = q->tail;
    q->size++;
}
```

$T_{Enqueue} = O(1)$

Реализация очереди на основе связных списков

```
int main()
{
    struct queue *q;
    int i;
    q = queue_create();
    for (i = 1; i <= 10; i++) {
        queue_enqueue(q, i);
    }
    printf("Queue size: %d\n", queue_size(q));
    queue_free(q);
    return 0;
}
```

Удаление элемента из очереди (dequeue)

```
int queue_dequeue(struct queue *q)
{
    int value;
    struct listnode *p;
    if (q->size == 0)
        return -1;

    value = q->head->value;
    p = q->head->next;
    free(q->head);
    q->head = p;
    q->size--;
    return value;
}
```

$T_{Dequeue} = O(1)$

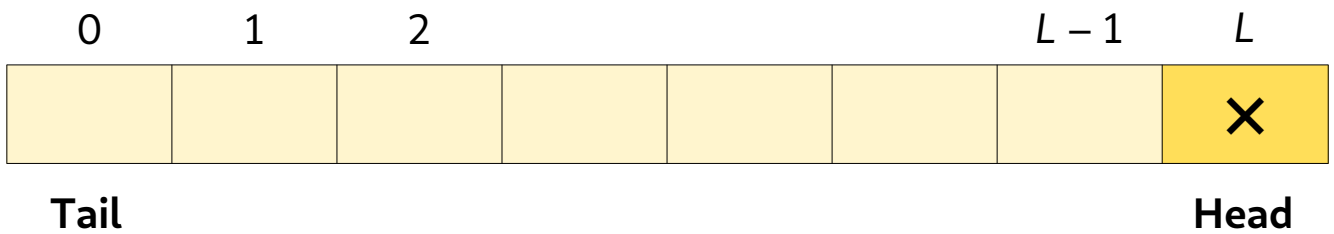
Реализация очереди на основе связных списков

```
int main()
{
    struct queue *q;
    int i, value;
    q = queue_create();

    /* ... */
    for (i = 1; i <= 11; i++) {
        value = queue_dequeue(q);
        printf("Next element: %d\n", value);
    }
    queue_free(q);
    return 0;
}
```

Реализация очереди на основе циклического массива

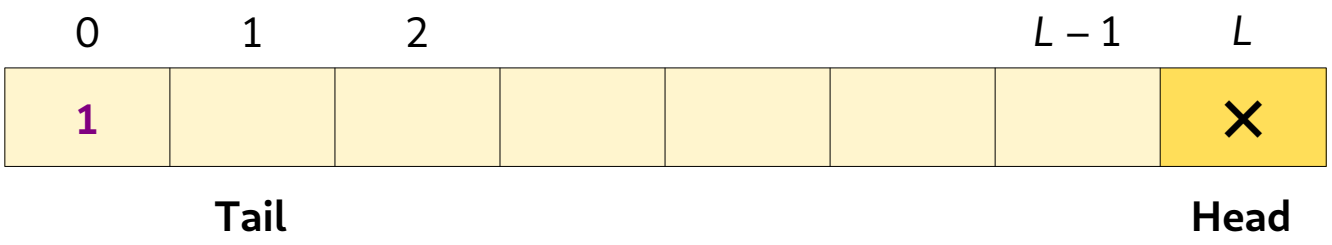
- Элементы очереди хранятся в массиве фиксированной длины [0..*L* – 1]
- Массив логически представляется в виде кольца (circular buffer)
- В пустой очереди *tail* = 0, *head* = *L*



Реализация очереди на основе циклического массива

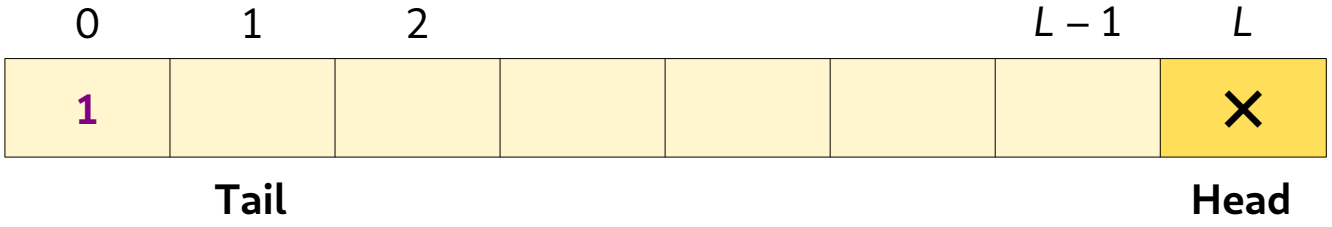
- При добавлении (push) элемента в очередь значение *tail* **циклически увеличивается** на 1 (сдвигается на следующую свободную позицию)
- Если *head* = *tail* + 1, то очередь переполнена!

Enqueue(1):

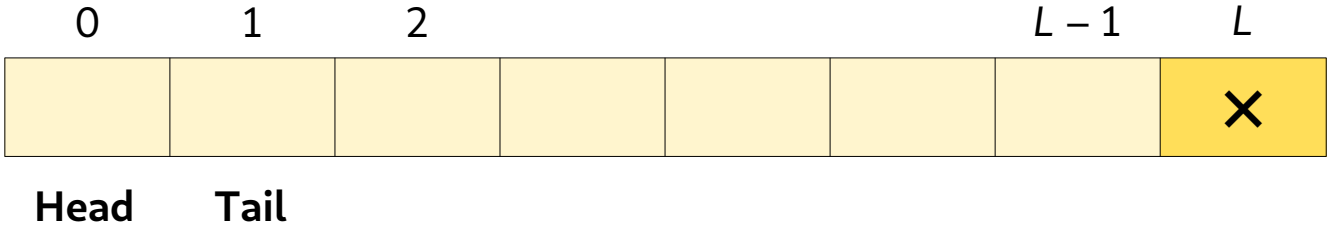


Реализация очереди на основе циклического массива

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



Dequeue():



Реализация очереди на основе циклического массива

```c
#include <stdio.h>
#include <stdlib.h>

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

$T_{Create} = O(1)$

Удаление очереди. Получение размера очереди

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

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

$T_{Free} = O(1)$

$T_{Size} = O(1)$

Добавление элемента в очередь (enqueue)

```
int queue_enqueue(struct queue *q, int value)
{
    if (q->head == q->tail + 1) {
        fprintf(stderr, “Queue overflow\n”);
        return -1;
    }

    q->a[q->tail++] = value;
    q->tail = q->tail % (q->maxsize + 1);
    q->size++;
    return 0;
}
```

$T_{Enqueue} = O(1)$

Удаление элемента из очереди (dequeue)

```
int queue_dequeue(struct queue *q)
{
    if (q->head % (q->maxsize + 1) == q->tail) {
        fprintf(stderr, “Queue is empty\n”);
        return -1;
    }

    q->head = q->head % (q->maxsize + 1);
    q->size--;
    return q->a[q->head++];
}
```

$T_{Dequeue} = O(1)$

Реализация очереди на основе циклического массива

```
int main()
{
    struct queue *q;
    int i, value;
    q = queue_create(8);
    value = queue_dequeue(q);
    for (i = 1; i <= 9; i++)
        queue_enqueue(q, i);
    for (i = 1; i <= 4; i++) {
        value = queue_dequeue(q);
        printf("Next element: %d\n", value);
    }
    queue_free(q);
    return 0;
}
```

Домашнее чтение

- Найти информацию о двухсторонней очереди (**дек**, **deque** — double-ended queue)
- [DSABook] **Глава 7. «Стеки». Глава 8. «Очереди»**

ご清聴ありがとうございました!

Даниил Михайлович Берлизов
Старший преподаватель Кафедры вычислительных систем СибГУТИ
E-mail: sillyhat34@gmail.com

Курс «Структуры и алгоритмы обработки данных»
Весенний семестр, 2021 г.