

Связные списки

Лекция 5

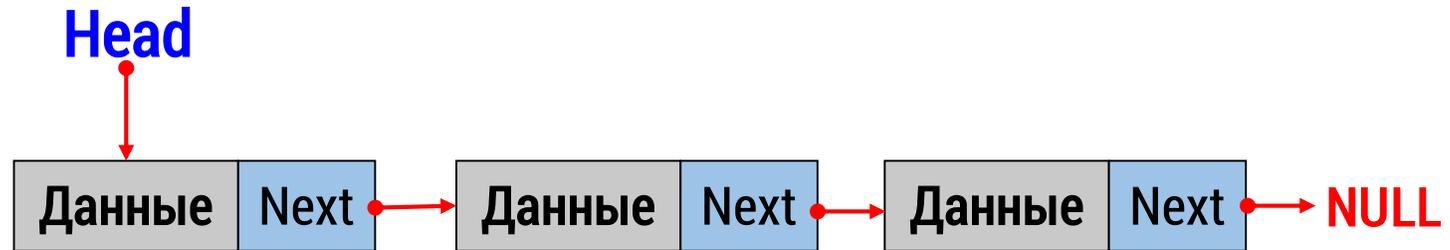
Связные списки (Linked lists)

- **Связный список (linked list)** – динамическая структура данных для хранения информации, в которой каждый элемент хранит указатели на один или несколько других элементов

Операция	Описание	Вычислительная сложность	Сложность по памяти
AddFront(L, x)	Добавляет элемент x в начало списка L	$O(1)$	$O(1)$
AddEnd(L, x)	Добавляет элемент x в конец списка L	$O(n)$	$O(1)$
Lookup(L, x)	Отыскивает элемент x в списке L	$O(n)$	$O(1)$
Size(L)	Возвращает количество элементов в списке L	$O(1)$ или $O(n)$	$O(1)$

Односвязный список (Singly linked list)

- Размер списка заранее не известен – элементы добавляются во время работы программы (динамически)
- Память под элементы выделяется динамически (функции: malloc, calloc, free)



Односвязный список (Singly linked list)

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

struct listnode {
    char *data;           /* Data */
    int value;           /* Data */
    struct listnode *next; /* Next node */
};
```

Односвязный список (Singly linked list)

```
struct listnode *list_createNode(char *data,  
                                int value)  
{  
    struct listnode *p;  
  
    p = malloc(sizeof(*p));    // Выделяем память  
    if (p != NULL) {  
        p->data = strdup(data);  
        p->value = value;  
        p->next = NULL;  
    }  
    return p;  
}
```

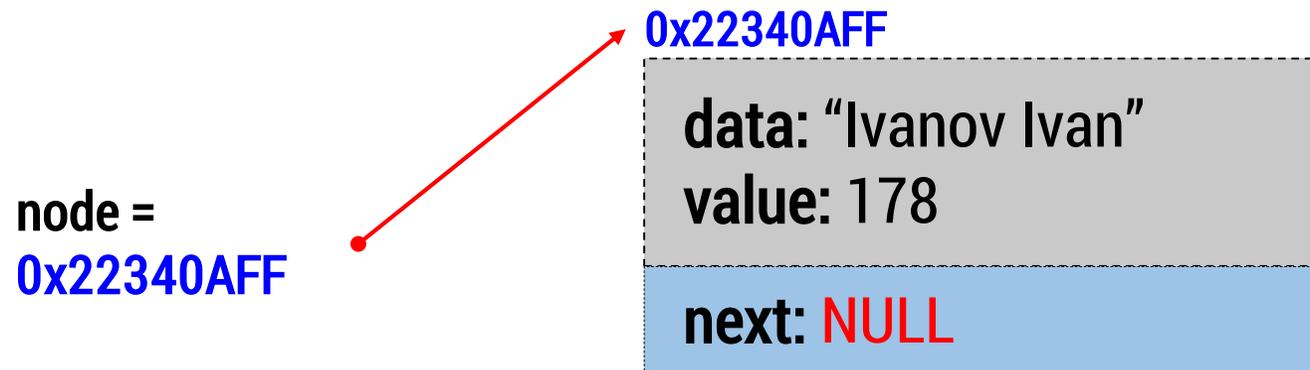
Сложность создания элемента
 $T_{CreateNode} = O(1)$

Создание элемента (выделение памяти)

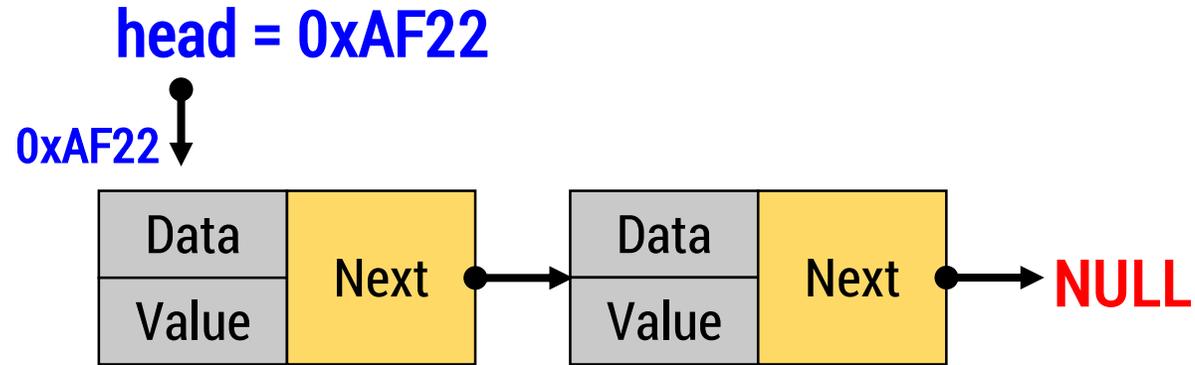
```
int main()
{
    struct listnode *node;

    /* Список из одного элемента */
    node = list_createnode("Ivanov Ivan", 178);

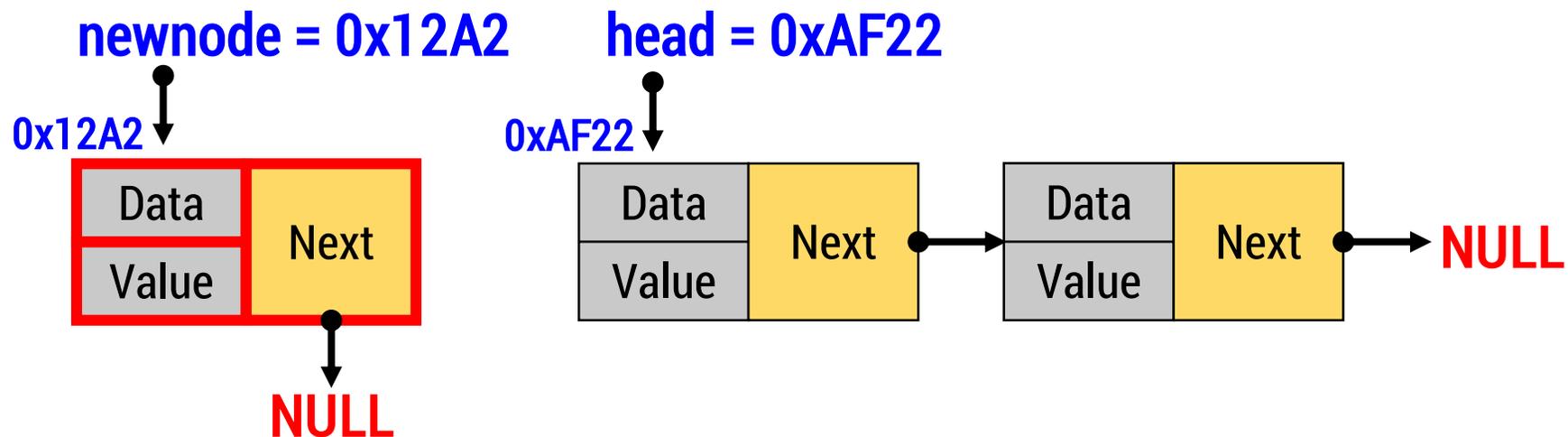
    return 0;
}
```



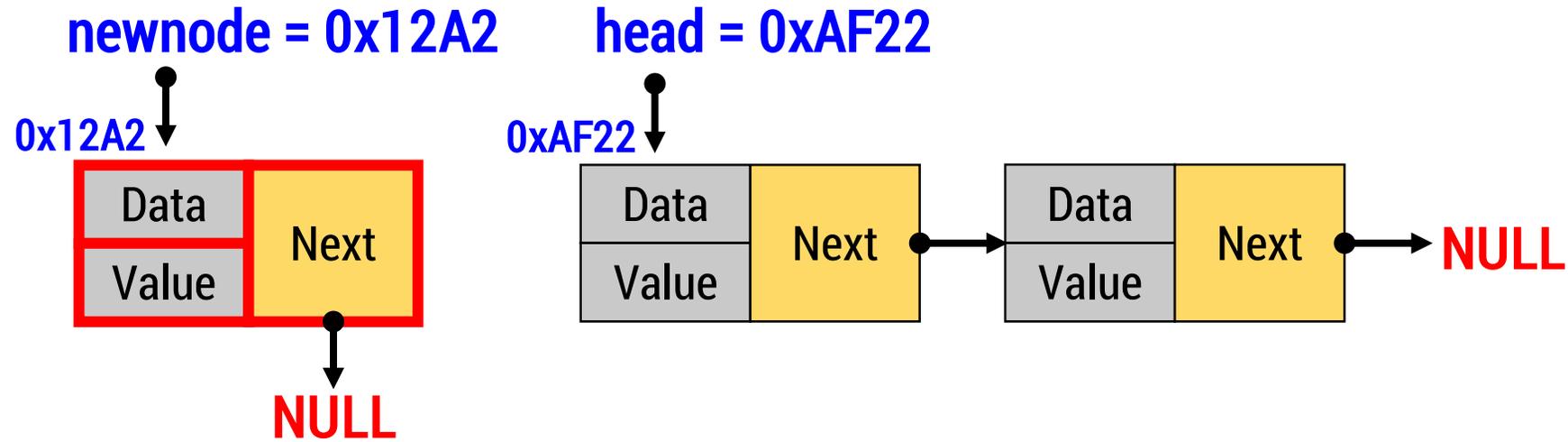
Добавление элемента в начало списка



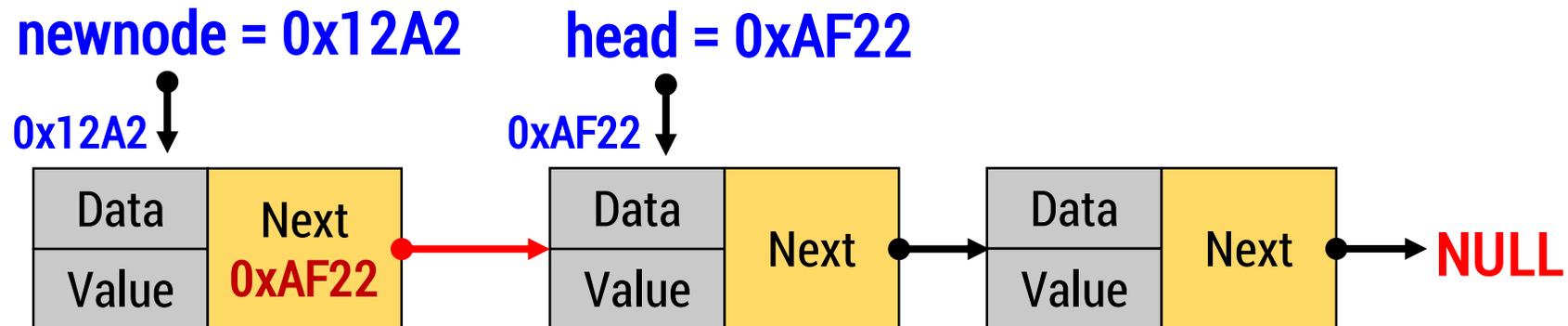
1. Создаем новый узел `newnode` в памяти



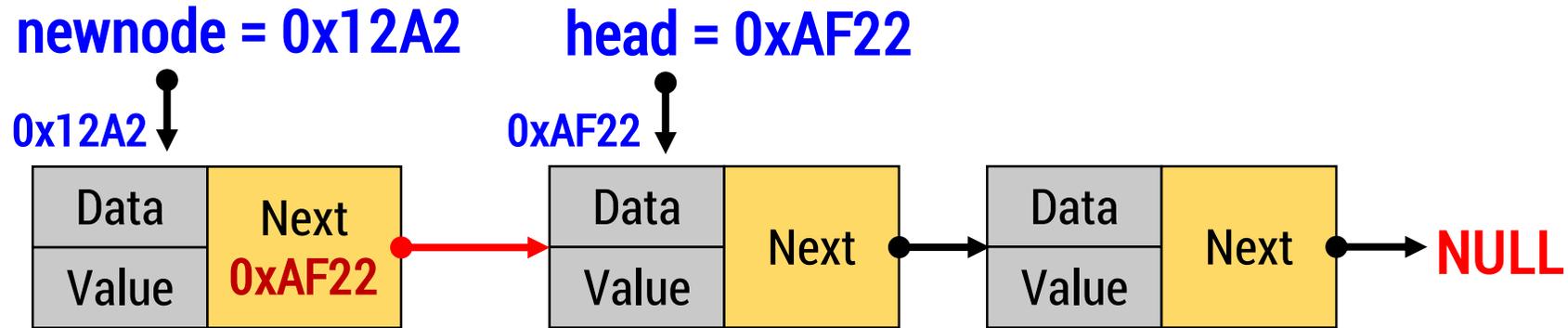
Добавление элемента в начало списка



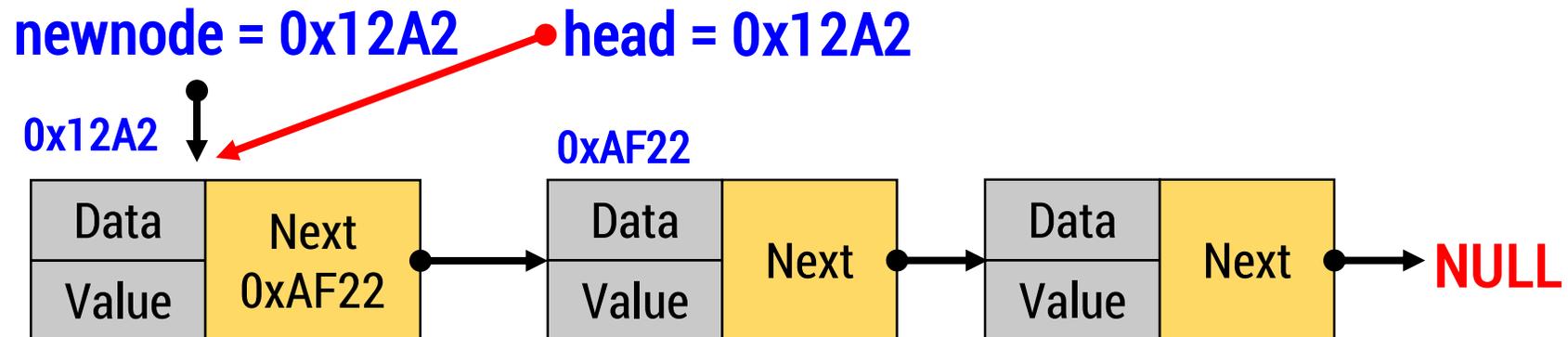
2. Устанавливаем указатель **next** узла **newnode** на **head**



Добавление элемента в начало списка



3. Делаем головой списка узел newnode



Добавление элемента в начало списка

```
struct listnode *list_addfront(struct listnode *list,  
                               char *data, int value)  
{  
    struct listnode *newnode;  
    newnode = list_createnode(data, value);  
  
    if (newnode != NULL) {  
        newnode->next = list;  
        return newnode;  
    }  
    return list;  
}
```

$$T_{AddFront} = O(1)$$

Добавление элемента в начало списка

```
int main()
{
    struct listnode *head;

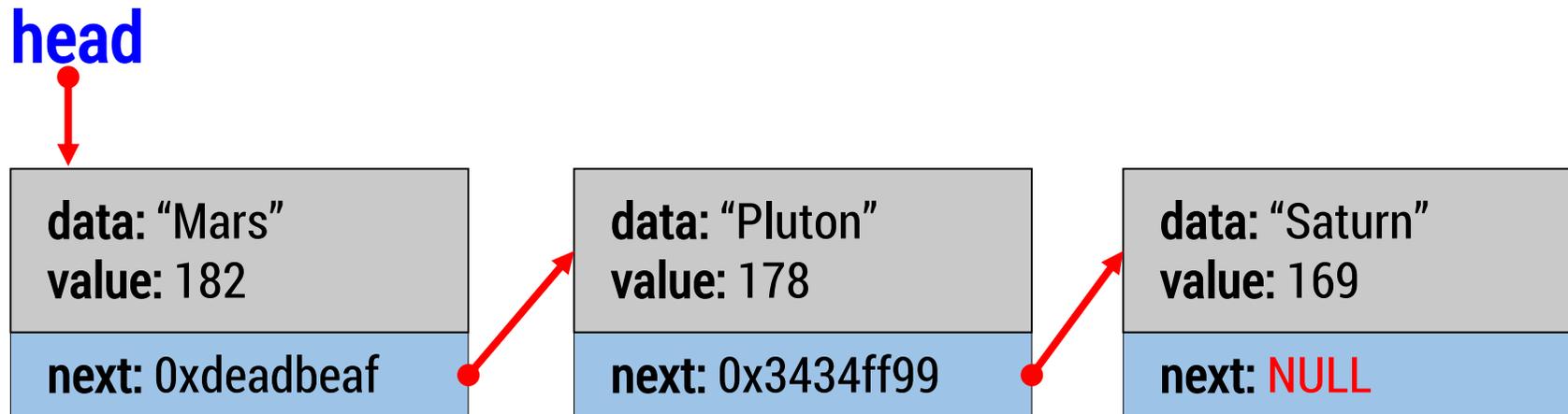
    head = list_addfront(NULL, "Ivanov Ivan", 178);
    head = list_addfront(head, "Petrov Petr", 182);

    return 0;
}
```



Поиск элемента в списке (Lookup)

- Начиная с головы списка, просматриваем все узлы и сравниваем ключи
- В худшем случае требуется просмотреть все узлы, это требует $O(n)$ операций



Поиск элемента в списке (Lookup)

```
struct listnode *list_lookup(struct listnode *list,
                             char *data, int value)
{
    for ( ; list != NULL; list = list->next) {
        if (strcmp(list->data, data) == 0 &&
            list->value == value)
        {
            return list;
        }
    }
    return NULL; // Не нашли
}
```

$$T_{Lookup} = O(n)$$

Поиск элемента в списке (Lookup)

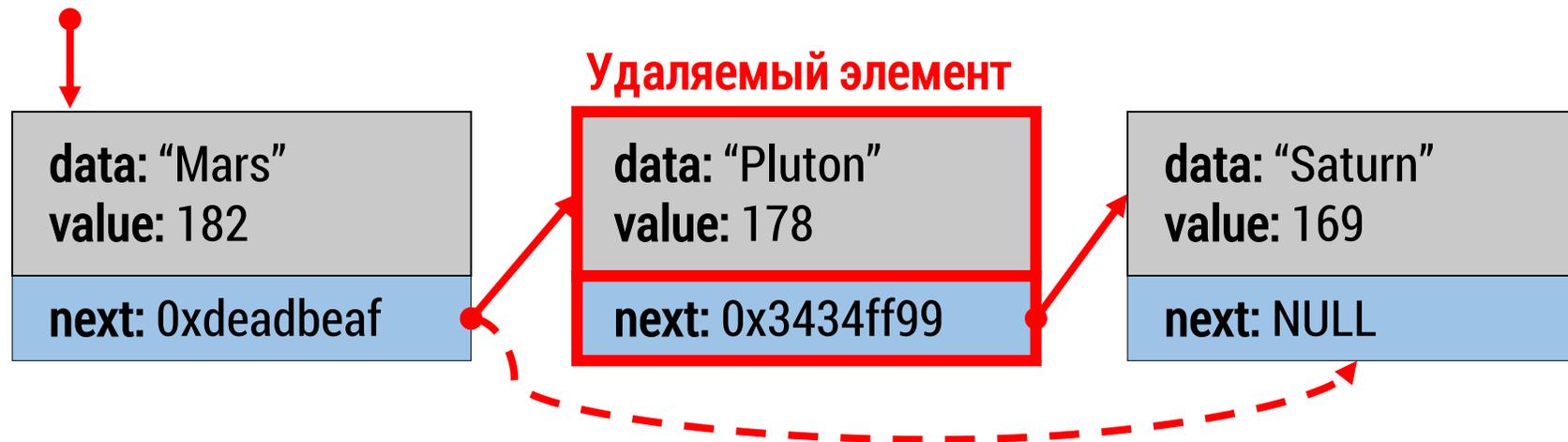
```
int main()
{
    struct listnode *node, *p;

    node = list_addfront(NULL, "Mars", 178);
    node = list_addfront(node, "Pluton", 182);
    node = list_addfront(node, "Saturn", 169);

    p = list_lookup(node, "Pluton", 182);
    if (p != NULL) {
        printf("Data: %s\n", p->data);
    }
    return 0;
}
```

Удаление элемента (Delete)

1. Находим элемент в списке (за время $O(n)$)
2. Корректируем указатели (за $O(1)$)
3. Удаляем элемент из памяти (за $O(1)$)



Три возможных ситуации:

- удаляемый узел – в начале списка
- удаляемый узел – внутренний узел (есть элементы слева и справа)
- удаляемый узел – в конце списка

Удаление элемента (Delete)

```
struct listnode *list_delete(struct listnode *list,
                             char *data, int value)
{
    struct listnode *p, *prev = NULL;

    for (p = list; p != NULL; p = p->next) {
        if (strcmp(p->data, data) == 0 && p->value == value) {
            if (prev == NULL)
                list = p->next;          // Удаляем голову
            else
                prev->next = p->next;    // Есть элемент слева
            free(p);                     // Освобождаем память
            return list;                // Указатель на новую голову
        }
        prev = p;                       // Запоминаем предыдущий элемент (левый)
    }
    return NULL;                        // Не нашли
}
```

$$T_{Delete} = O(n)$$

Удаление элемента (Delete)

```
int main()
{
    struct listnode *node, *p;

    node = list_addfront(NULL, "Mars", 178);
    node = list_addfront(node, "Pluton", 182);
    node = list_addfront(node, "Saturn", 169);

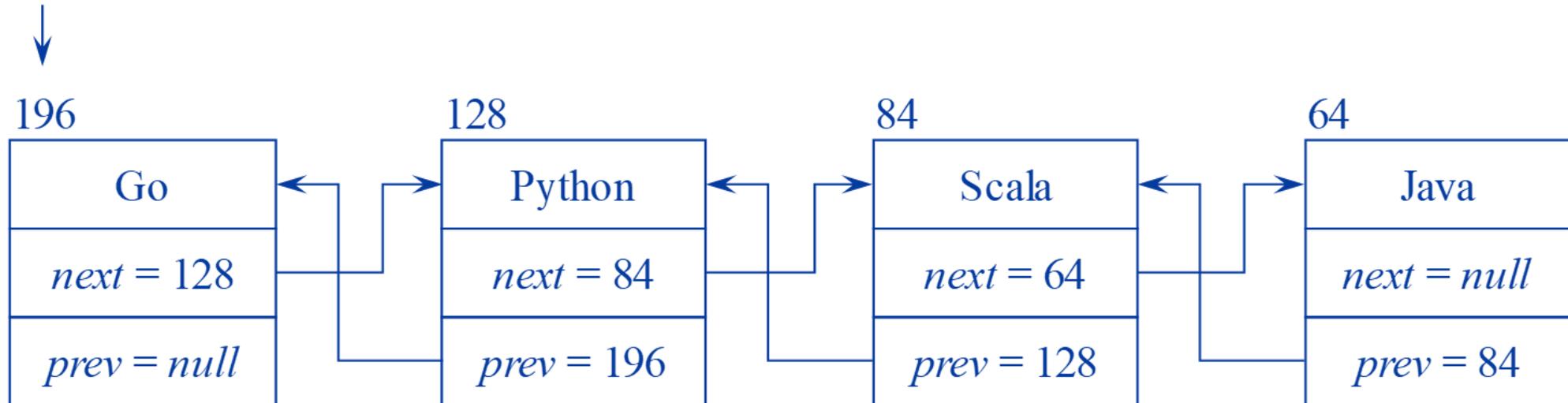
    p = list_delete(node, "Pluton", 182);
    if (p != NULL) {
        node = p; // Указатель на новую голову
        printf("Item deleted\n");
    }
    return 0;
}
```

Двусвязные списки (Doubly linked lists)

- Каждый узел двусвязного списка имеет три поля: `value`, `next` и `prev`
- Поле `value` – это некоторые данные, ассоциированные с узлом
- В поле `next` хранится адрес узла, следующего за текущим, а в поле `prev` – адрес предшествующего узла

head = 196

(ГОЛОВА)



Создание нового узла

```
struct listnode *list_create_node(char *data,  
                                  int value)  
{  
    struct listnode *p;  
  
    p = malloc(sizeof(*p));    // Выделяем память  
    if (p != NULL) {  
        p->data = strdup(data);  
        p->value = value;  
        p->next = NULL;  
        p->prev = NULL;  
    }  
    return p;  
}
```

$T = O(1)$

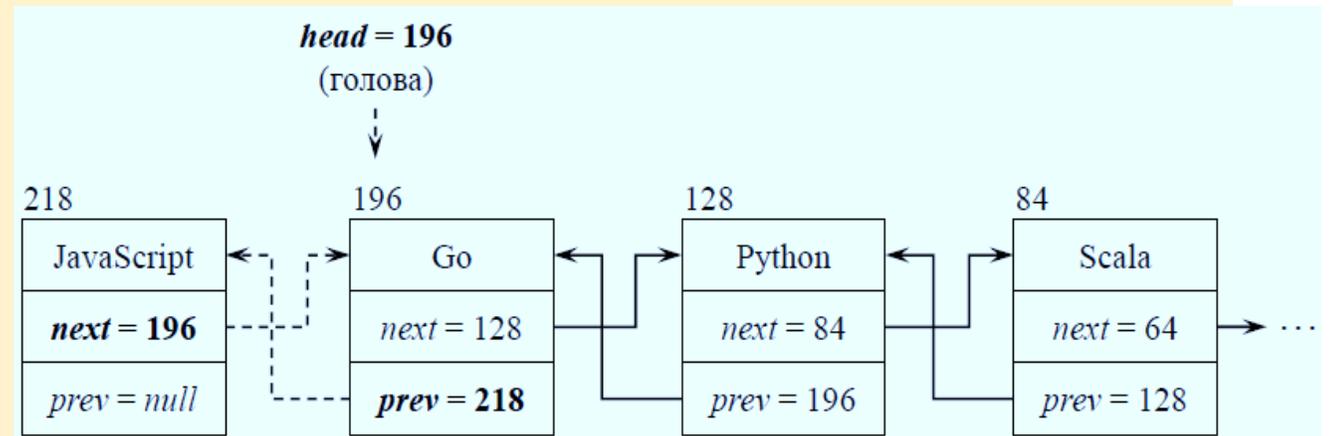
Добавление узла в начало списка

- Функция создает в памяти новый узел с заданным значением поля `value`
- В поле `next` нового узла заносится адрес головы списка
- Если список не пуст, то необходимо записать в указатель `prev` первого узла адрес нового элемента

```
struct listnode *list_addfront(struct listnode *list, char *data, int value)
{
    struct listnode *newnode;
    newnode = list_createnode(data, value);

    if (newnode != NULL) {
        newnode->next = list;
        if (list != NULL)
            list->prev = newnode;
        return newnode;
    }
    return list;
}
```

$T = O(1)$

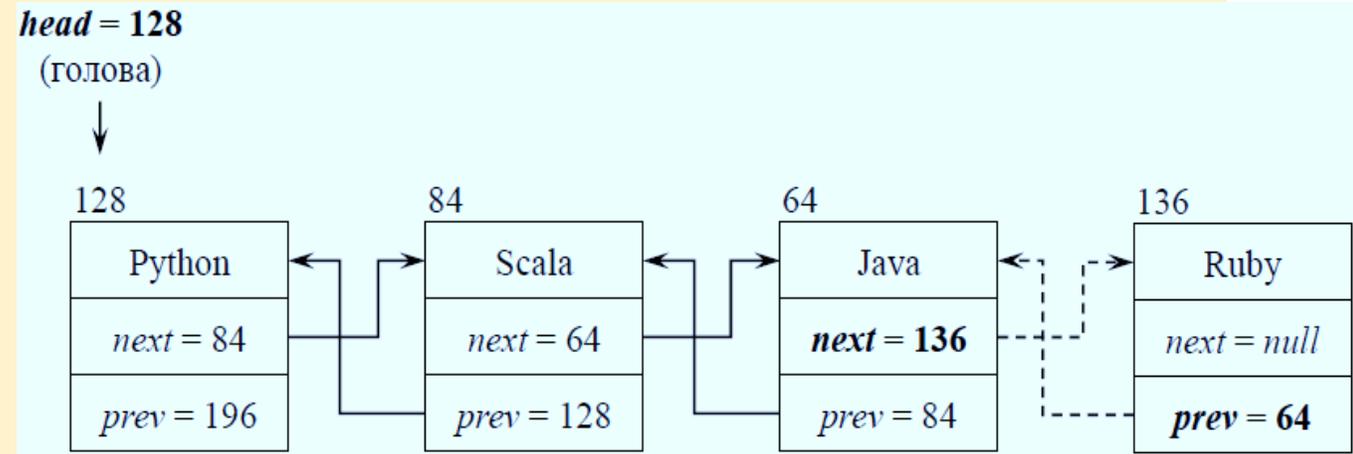


Добавление узла в конец списка

```
struct listnode *list_addend(struct listnode *list, char *data, int value)
{
    struct listnode *newnode, node;
    newnode = list_createnode(data, value);

    if (newnode != NULL) {
        if (list == NULL)
            return newnode;
        node = list;
        while (node.next != NULL)
            node = node.next;
        node.next = newnode;
        newnode.prev = node;
    }
    return list;
}
```

$T = O(n)$



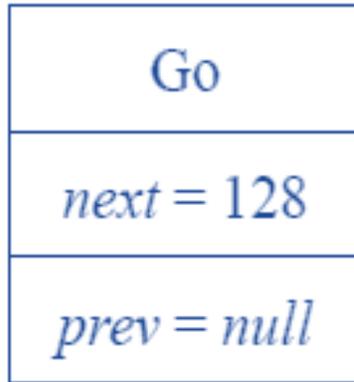
Удаление узла

head = 196

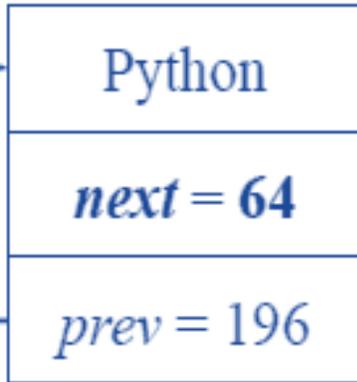
(ГОЛОВА)



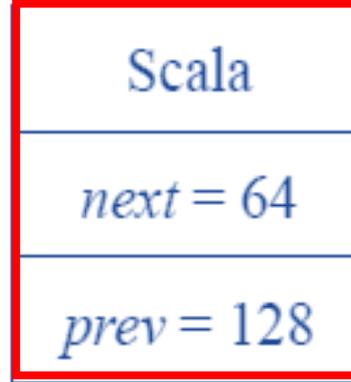
196



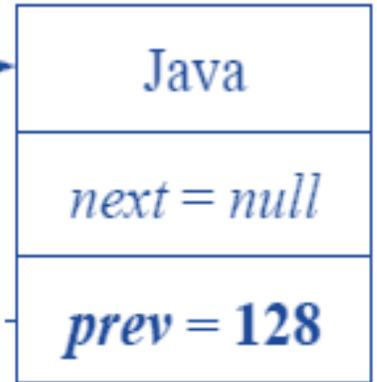
128



84



64



Удаление узла

```
struct listnode *list_delete(struct listnode *list,
                             char *data, int value)
{
    struct listnode *p;

    for (p = list; p != NULL; p = p->next) {
        if (strcmp(p->data, data) == 0 && p->value == value) {
            if (p->prev == NULL)
                list = p->next;           // Удаляем голову
            else
                p->prev->next = p->next;  // Есть элемент слева
            if (p->next != NULL)
                p->next->prev = p->prev;
            free(p);                       // Освобождаем память
            return list;                   // Указатель на новую голову
        }
    }
    return NULL; // Не нашли
}
```

$T = O(n)$

Литература

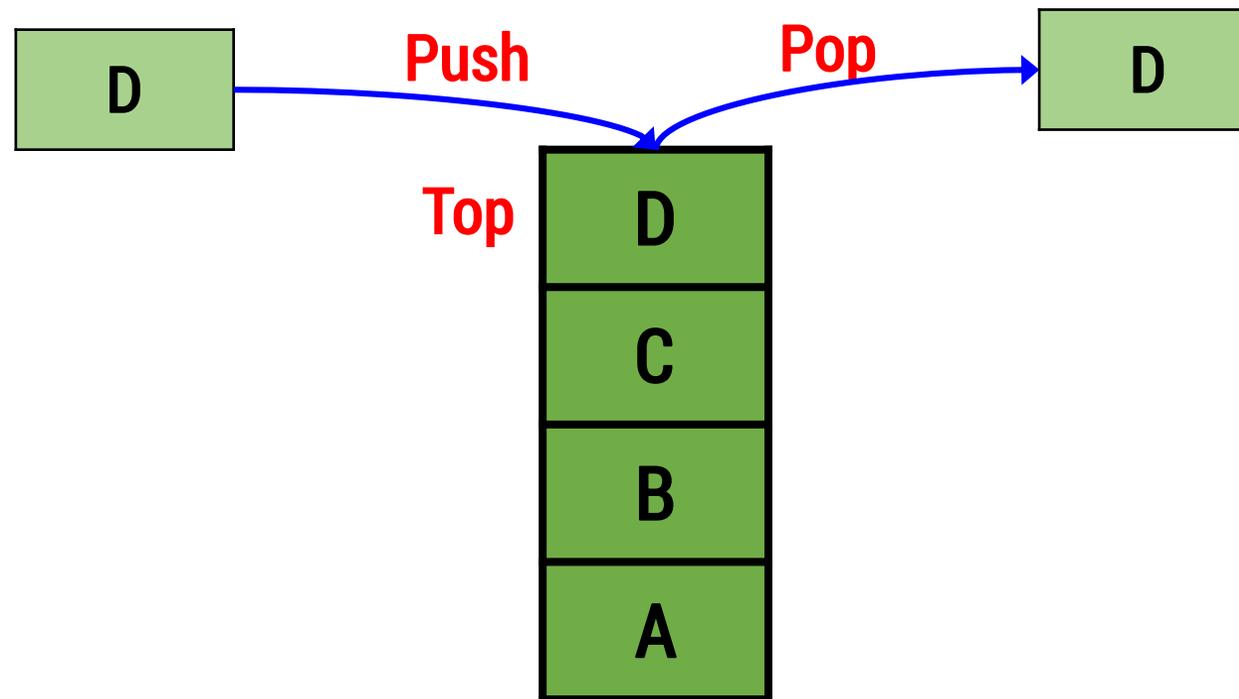
- Керниган Б.У., Пайк Р. Практика программирования. – М.: Вильямс, 2004.
[Kernighan, С. 61-66]
- [DSABook, Глава 6]

Стеки и очереди

Лекция 6

Стек (Stack)

- Стек (Stack) – структура данных для хранения элементов
- Дисциплина доступа к элементам:
“последним пришел – первым вышел” (Last In – First Out, LIFO)
- Элементы помещаются и извлекаются с вершины стека (top)



Подходы к реализации стека

1. На основе связанных списков (linked lists)

Длина стека ограничена объемом доступной памяти

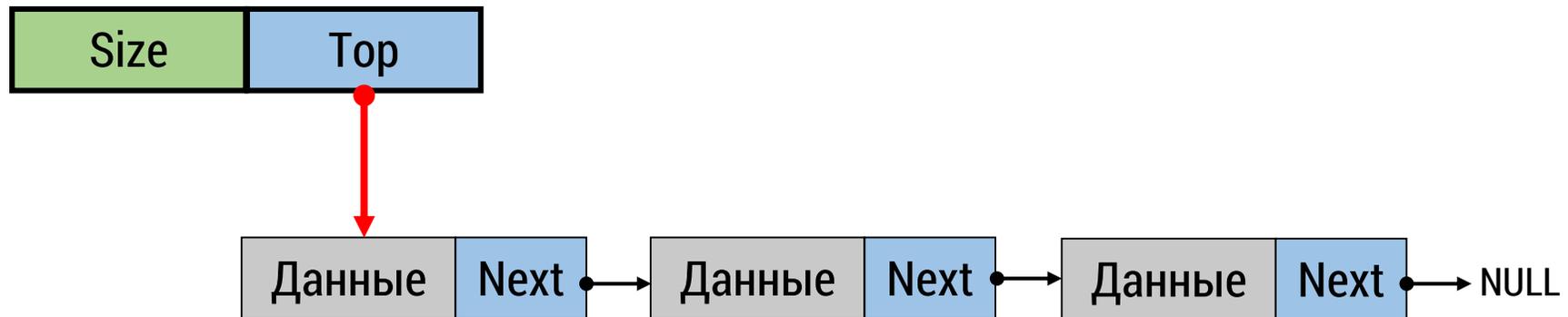
2. На основе статических массивов

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)
- Добавление элемент и удаление выполняется за время $O(1)$

Stack:



Реализация стека на основе СВЯЗНЫХ СПИСКОВ

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

#include "l1ist.h"

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

Реализация стека на основе СВЯЗНЫХ СПИСКОВ

```
/*  
 * Фрагмент файла llist.h  
 */  
  
struct listnode {  
    int value;          /* Данные стека */  
    struct listnode *next;  
};
```

Реализация стека на основе СВЯЗНЫХ СПИСКОВ

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

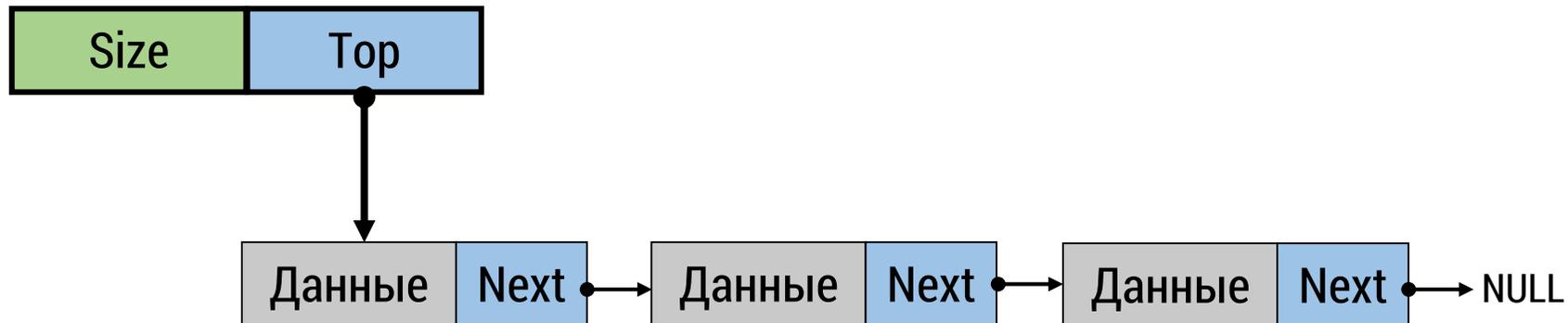
$$T_{Create} = O(1)$$

Реализация стека на основе СВЯЗНЫХ СПИСКОВ

```
/* stack_free: Removes stack */  
void stack_free(struct stack *s)  
{  
    while (s->size > 0)  
        stack_pop(s);    /* Delete All items */  
    free(s);  
}
```

$$T_{Free} = O(n)$$

Stack:



Реализация стека на основе СВЯЗНЫХ СПИСКОВ

```
/* stack_size: Returns size of a stack */  
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;  
}
```

Реализация стека на основе СВЯЗНЫХ СПИСКОВ

```
/* stack_push: Pushes item to the stack */
int stack_push(struct stack *s, int value)
{
    s->top = list_addfront(s->top, value);
    if (s->top == NULL) {
        fprintf(stderr,
                "stack: Stack overflow\n");
        return -1;
    }
    s->size++;
    return 0;
}
```

$$T_{Push} = O(1)$$

Реализация стека на основе СВЯЗНЫХ СПИСКОВ

```
int main()
{
    struct stack *s;
    int i, val;

    s = stack_create();
    for (i = 1; i <= 10; i++) {
        stack_push(s, i);
    }
    for (i = 1; i <= 11; i++) {
        val = stack_pop(s);
        printf("pop: %d\n", val);
    }
    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
- Добавление элемента и его удаление выполняются за время $O(1)$

	L - 1	
	...	
Top	2	C
	1	B
	0	A

Реализация стека на основе массива

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

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

Реализация стека на основе массива

```
/* stack_create: Creates an empty stack */
struct stack *stack_create(int maxsize)
{
    struct stack *s = malloc(sizeof(*s));
    if (s != NULL) {
        s->v = malloc(sizeof(int) * maxsize);
        if (s->v == NULL) {
            free(s);
            return NULL;
        }
        s->size = 0;
        s->top = 0;
        s->maxsize = maxsize;
    }
    return s;
}
```

$$T_{Create} = O(1)$$

Реализация стека на основе массива

```
/* stack_free: Removes stack */  
void stack_free(struct stack *s)  
{  
    free(s->v);  
    free(s);  
}
```

$$T_{Free} = O(1)$$

```
/* stack_size: Returns size of a stack */  
int stack_size(struct stack *s)  
{  
    return s->size;  
}
```

$$T_{Size} = O(1)$$

Реализация стека на основе массива

```
/* stack_push: Pushes item to the stack */
int stack_push(struct stack *s, int value)
{
    if (s->top < s->maxsize) {
        s->v[s->top++] = value;
        s->size++;
    } else {
        fprintf(stderr, "stack: Stack overflow\n");
        return -1;
    }
    return 0;
}
```

$$T_{Push} = O(1)$$

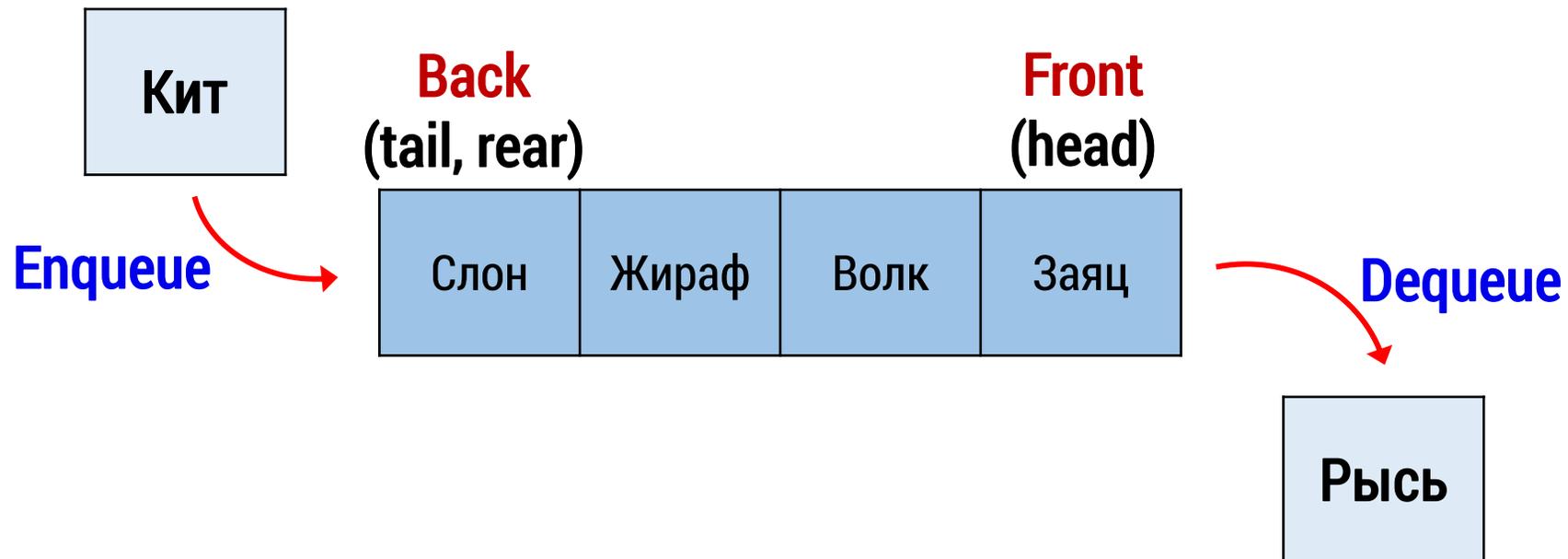
Реализация стека на основе массива

```
/* stack_pop: Pops item from the stack */
int stack_pop(struct stack *s)
{
    if (s->top == 0) {
        fprintf(stderr,
                "stack: Stack underflow\n");
        return -1;
    }
    s->size--;
    return s->v[--s->top];
}
```

$$T_{Pop} = O(1)$$

Очередь (Queue)

- 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)



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 очередь q
Dequeue(q)	Извлекает элемент из очереди q
Size(q)	Возвращает количество элементов в очереди q
Clear(q)	Очищает очередь q

Подходы к реализации очереди

1. На основе связанных списков

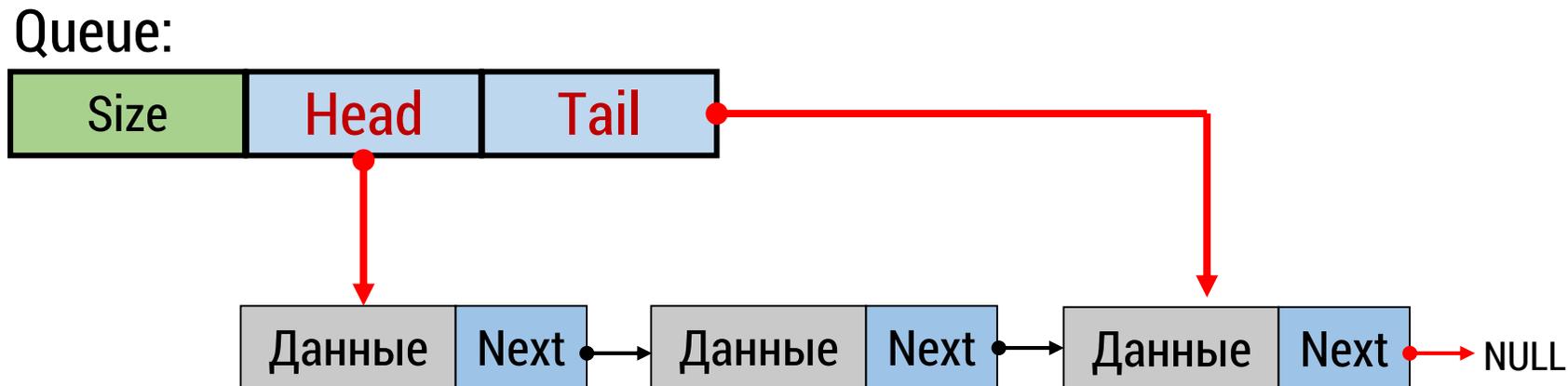
Длина очереди ограничена лишь объемом доступной памяти

2. На основе статических массивов

Длина очереди фиксирована (задана максимальная длина)

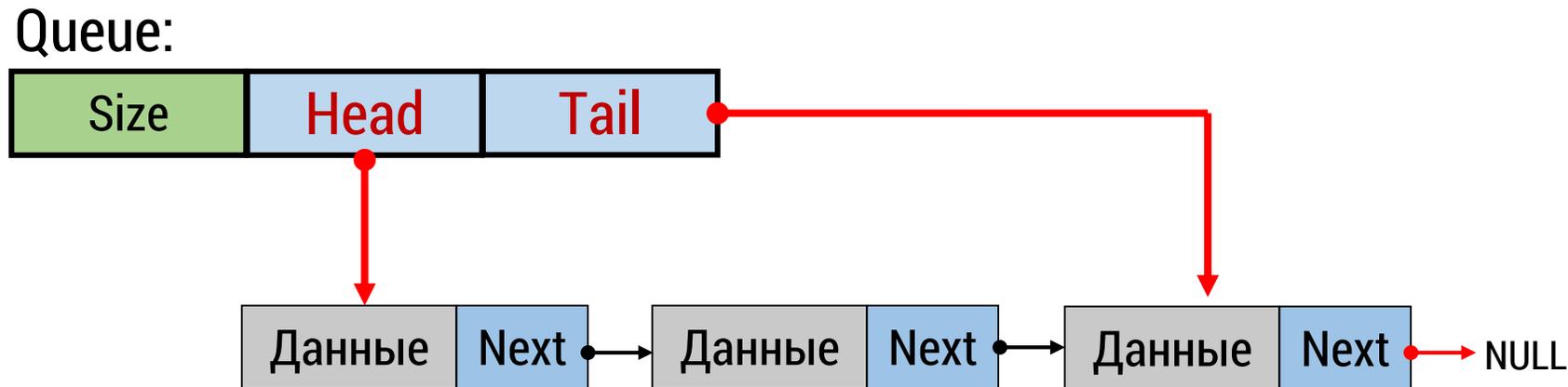
Реализация очереди на основе СВЯЗНЫХ СПИСКОВ

- 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 объемом 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

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

#include "l1ist.h"

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

Реализация очереди на основе СВЯЗНЫХ СПИСКОВ

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

$$T_{Create} = O(1)$$

Реализация очереди на основе СВЯЗНЫХ СПИСКОВ

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

$$T_{Free} = O(n)$$

Реализация очереди на основе СВЯЗНЫХ СПИСКОВ

```
/* queue_size: Returns size of a queue */
int queue_size(struct queue *q)
{
    return q->size;
}

int main()
{
    struct queue *q;

    q = queue_create();

    queue_free(q);
    return 0;
}
```

$$T_{Size} = O(1)$$

Реализация очереди на основе СВЯЗНЫХ СПИСКОВ

```
/* queue_enqueue: Add item to the queue */
void queue_enqueue(struct queue *q, int value)
{
    struct listnode *oldtail = q->tail;

    /* Create new node */
    q->tail = list_createnode(value);

    if (q->head == NULL) {
        /* List is empty */
        q->head = q->tail;
    } else {
        /* Add new node to the end of list */
        oldtail->next = q->tail;
    }
    q->size++;
}
```

$$T_{\text{Enqueue}} = O(1)$$

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;
}
```

Реализация очереди на основе СВЯЗНЫХ СПИСКОВ

```
/* queue_dequeue: Gets item from the queue */
int queue_dequeue(struct queue *q)
{
    int value;
    struct listnode *p;

    if (q->size == 0)
        return -1;

    /* Delete first node */
    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, j;

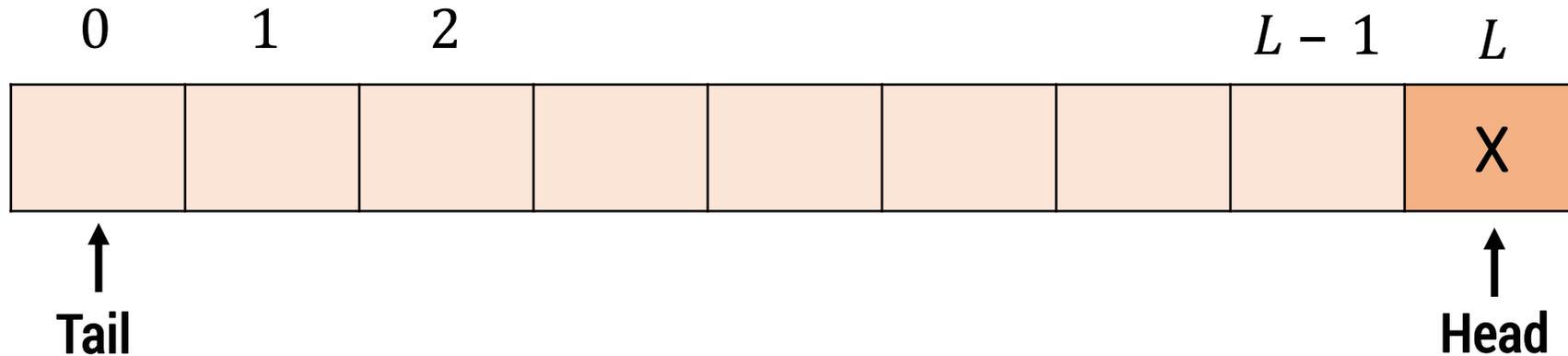
    q = queue_create();

    /* ... */
    for (i = 1; i <= 11; i++) {
        j = queue_dequeue(q);
        printf("%d: next element: %d\n", i, j);
    }

    queue_free(q);
    return 0;
}
```

Реализация очереди на основе циклических массивов

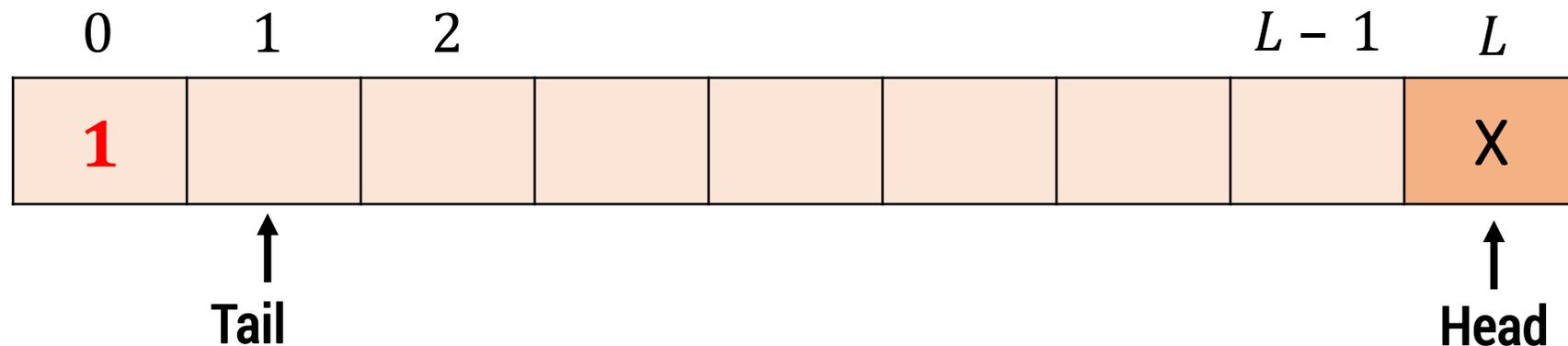
- Элементы очереди хранятся в массиве фиксированной длины $[0..L - 1]$
- Массив логически представляется в виде кольца (circular buffer)
- В пустой очереди $Tail = 0, Head = L$



Реализация очереди на основе циклических массивов

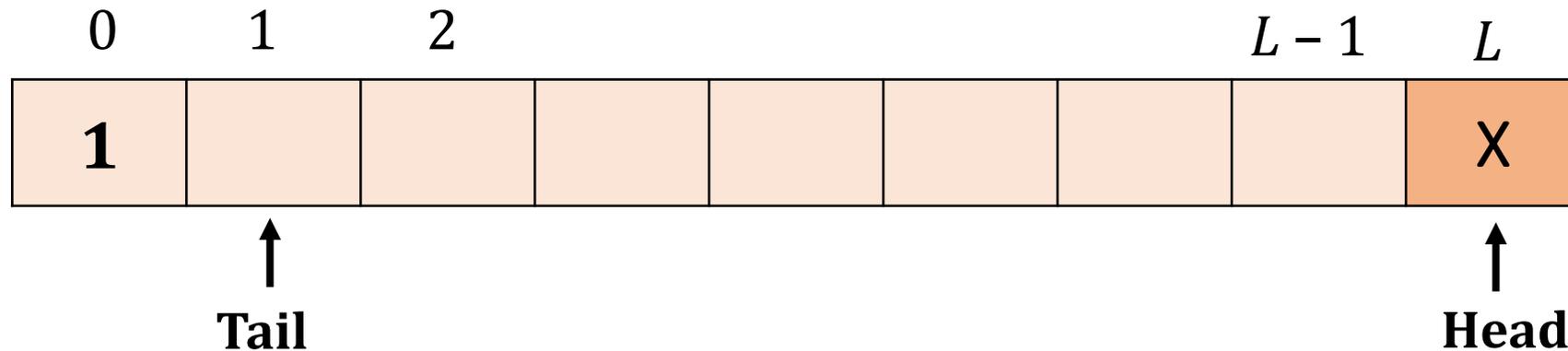
- При добавлении элемента в очередь значение *Tail* циклически увеличивается на 1 (сдвигается на следующую свободную позицию)
- Если $Head = Tail + 1$, то очередь переполнена!

Enqueue(1):

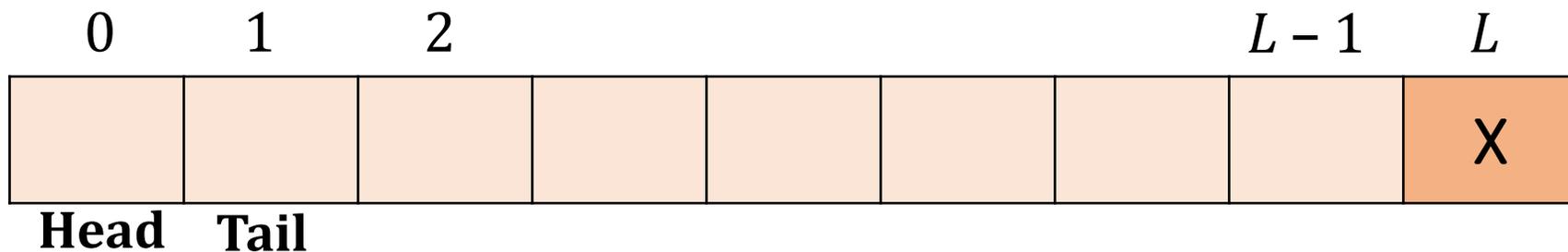


Реализация очереди на основе циклических массивов

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



Dequeue():



Реализация очереди на основе циклических массивов

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

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

Реализация очереди на основе циклических массивов

```
/* queue_create: Creates an empty queue */
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;
}
```

$$T_{Create} = O(1)$$

Реализация очереди на основе циклических массивов

```
/* queue_free: Removes queue */  
void queue_free(struct queue *q)  
{  
    free(q->v);  
    free(q);  
}  
  
/* queue_size: Returns size of a queue */  
int queue_size(struct queue *q)  
{  
    return q->size;  
}
```

Реализация очереди на основе циклических массивов

```
/* queue_enqueue: Add item to the queue */
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;
}
```

$$T_{\text{Enqueue}} = O(1)$$

Реализация очереди на основе циклических массивов

```
/* queue_dequeue: Gets item from the queue */
int queue_dequeue(struct queue *q)
{
    if (q->head % (q->maxsize + 1) == q->tail)
    {
        /* Queue is empty */
        fprintf(stderr,
                "queue: Queue is empty\n");
        return -1;
    }

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

$$T_{Dequeue} = O(1)$$

Реализация очереди на основе циклических массивов

```
int main()
{
    struct queue *q;
    int i, val;

    q = queue_create(10);
    val = queue_dequeue(q);
    for (i = 1; i <= 11; i++) {
        queue_enqueue(q, i);
    }
    for (i = 1; i <= 5; i++) {
        val = queue_dequeue(q);
    }

    queue_free(q);
    return 0;
}
```

Литература

- Найти и прочитать о двухсторонней очереди (дек, **deque** – double ended queue)
- [**DSABook**, Глава 7, Глава 8]