Абстрактные типы данных: списки, стеки, очереди, деки

Лекция 2

Схема процесса создания программ для решения прикладных задач

Абстрактные типы данных

Абстрактный тип данных (АТД) — это множество объектов, определяемое списком компонентов (операций , применимых к этим объектам, и их свойств).

Вся внутренняя структура такого типа спрятана от разработчика программного обеспечения — в этом и заключается суть абстракции.

Абстрактный тип данных определяет набор функций, независимых от конкретной реализации типа, для оперирования его значениями.

Конкретные реализации АТД называются структурами данных.

АТД - интерфейс

В программировании абстрактные типы данных обычно представляются в виде интерфейсов, которые скрывают соответствующие реализации типов.

Программисты работают с абстрактными типами данных исключительно через их интерфейсы, поскольку реализация может в будущем измениться.

Такой подход соответствует принципу инкапсуляции в объектно-ориентированном программировании.

Сильной стороной этой методики является именно сокрытие реализации. Раз вовне опубликован только интерфейс, то пока структура данных поддерживает этот интерфейс, все программы, работающие с заданной структурой абстрактным типом данных, будут продолжать работать.

Разработчики структур данных стараются, не меняя внешнего интерфейса и семантики функций, постепенно дорабатывать реализации, улучшая алгоритмы по скорости, надежности и используемой памяти.

АТД - список

- это множество, состоящее из n (n≥0) узлов (элементов) X[1], X[2], … , X[n], структурные свойства которого ограничены линейным (одномерным) относительным положением узлов (элементов), т.е. следующими условиями:

- если n > 0, то X[1] – первый узел;
- если 1 < k < n,

  то k-му узлу X[k] предшествует узел X[k-1],

  а за узлом X[k] следует узел X[k+1];
- X[n] – последний узел.

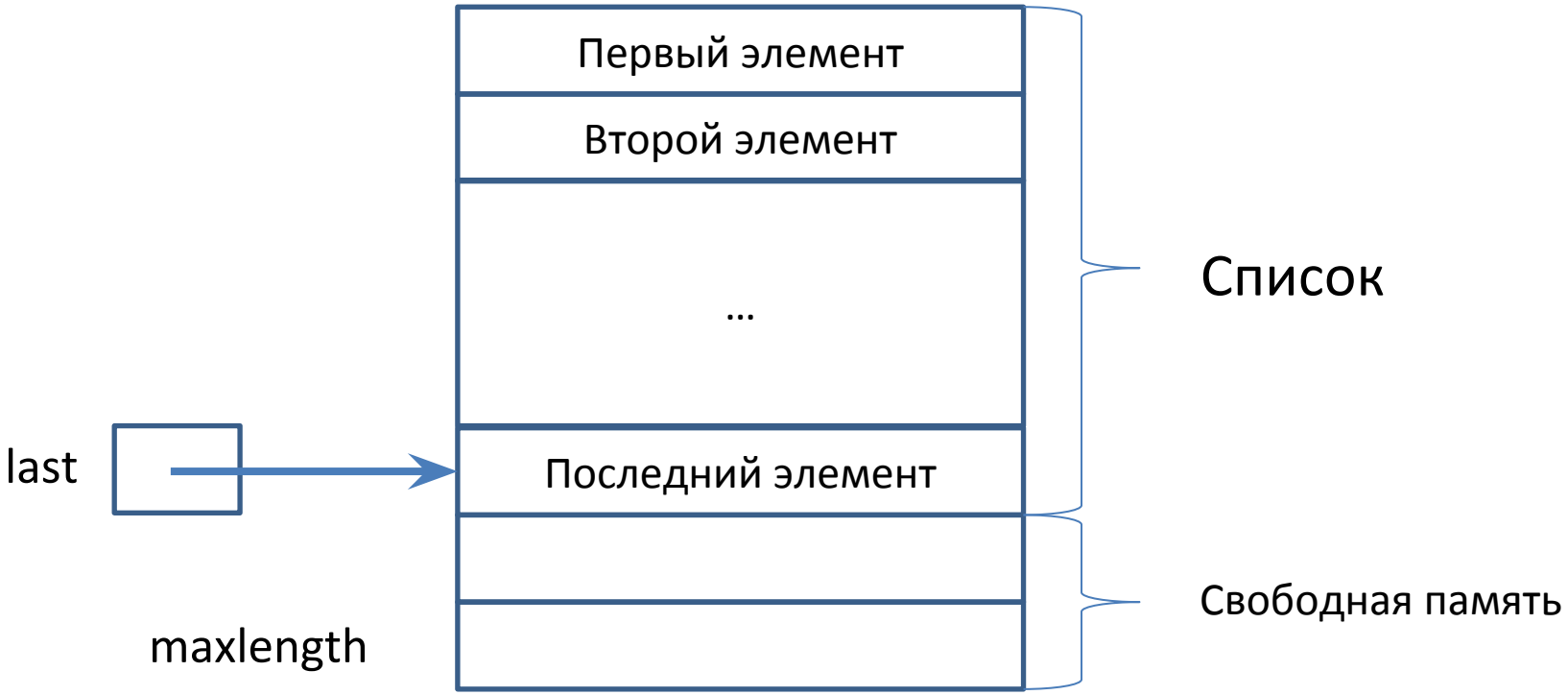
Операции над АТД «список»

- **END(L)** - возвращает позицию, следующую за позицией n в n-элементном списке L.
- **INSERT(x,p,L)** – вставляет объект x в позицию p в списке L.
- **LOCATE(x, L)** – возвращает позицию элемента x в списке L. Если в списке L нет объекта x, то возвращается **END(L)**.
- **RETRIEVE (p, L)** – возвращает элемент, который стоит в позиции p в списке L.
- **DELETE(p, L)** – удаляет элемент, стоящий в позиции p в списке L.
- **NEXT (p, L)** – возвращает следующую позицию за позицией p в списке L.
- **PREVIOUS (p, L)** – возвращает предыдущую позицию от позиции p в списке L.
- **MAKENULL (L)** – делает список L пустым и возвращает **END(L)**.

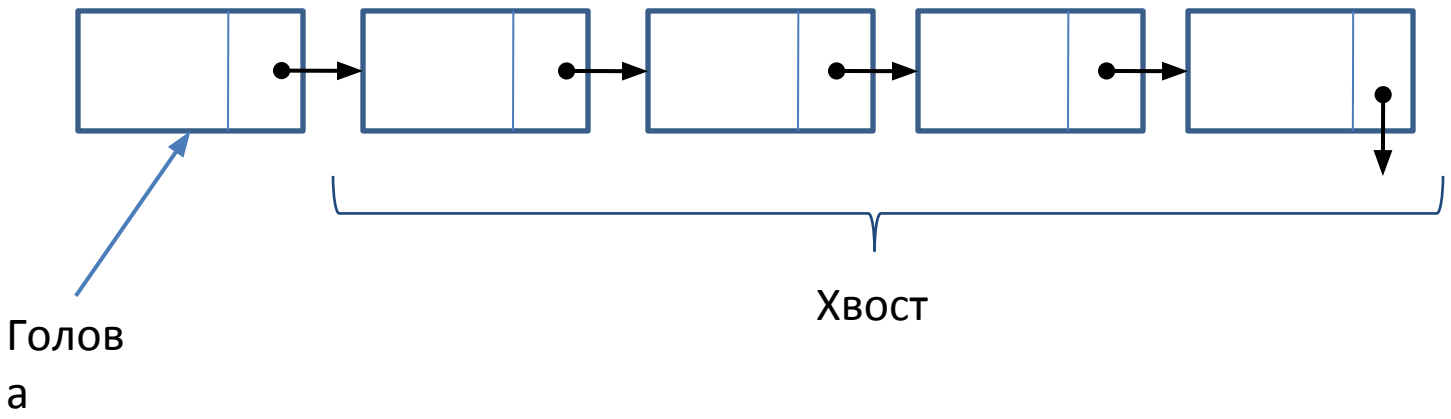
Реализация операций над АТД «Список»

- На массивах
- На динамических списках (с помощью указателей)
- На массивах с помощью курсоров

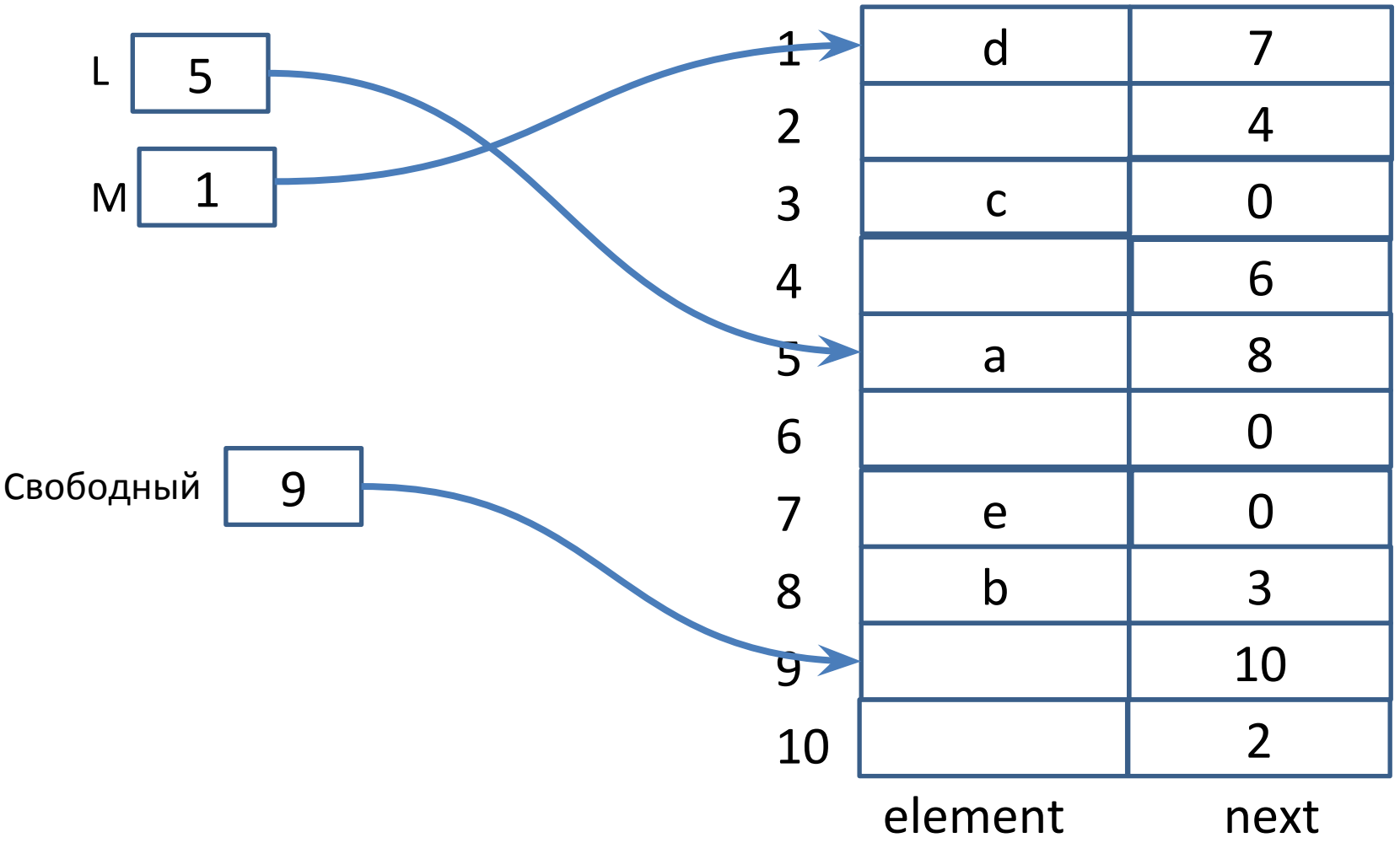
Реализация операций над АТД «Список» на массивах



Реализация операций над АТД «Список» с помощью указателей

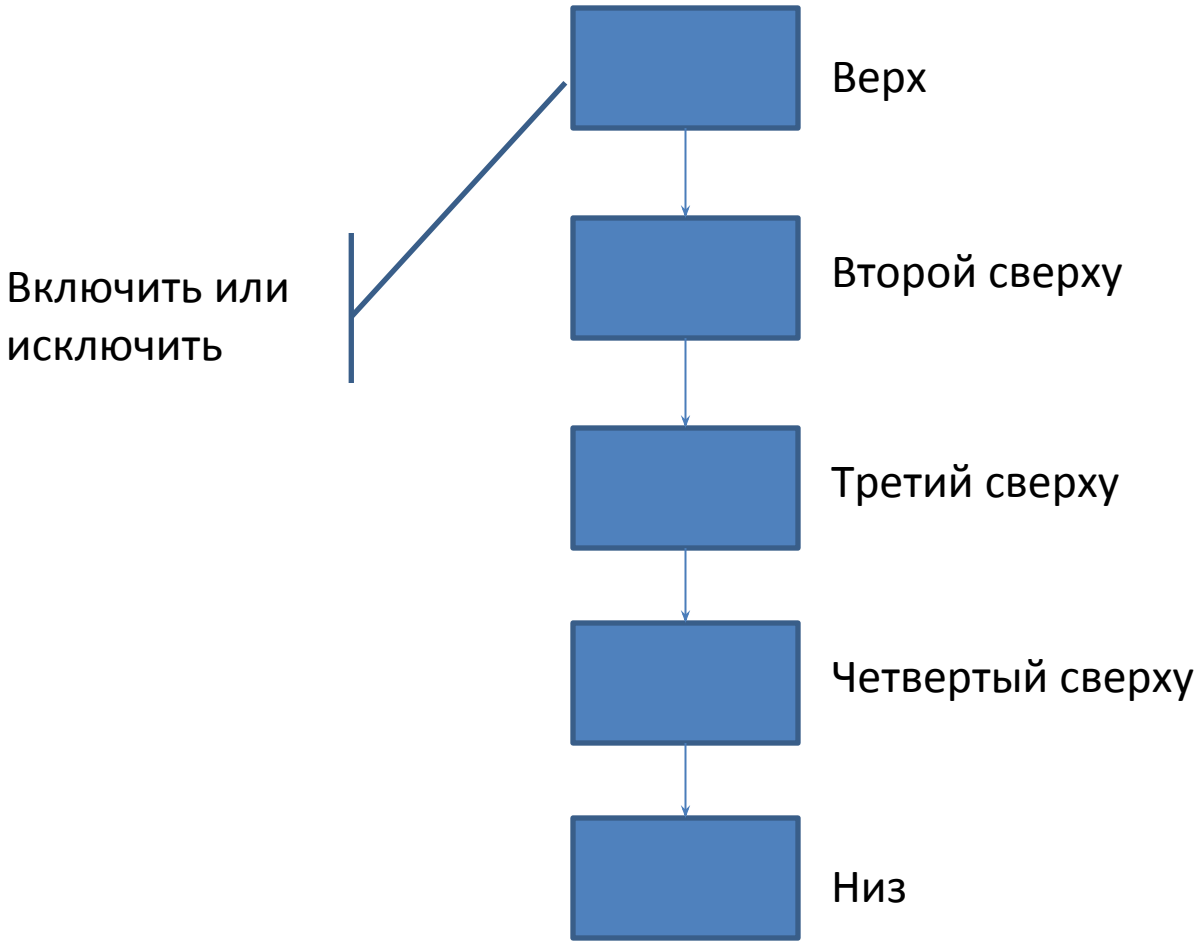


Реализация операций над АТД «Список» с помощью курсоров



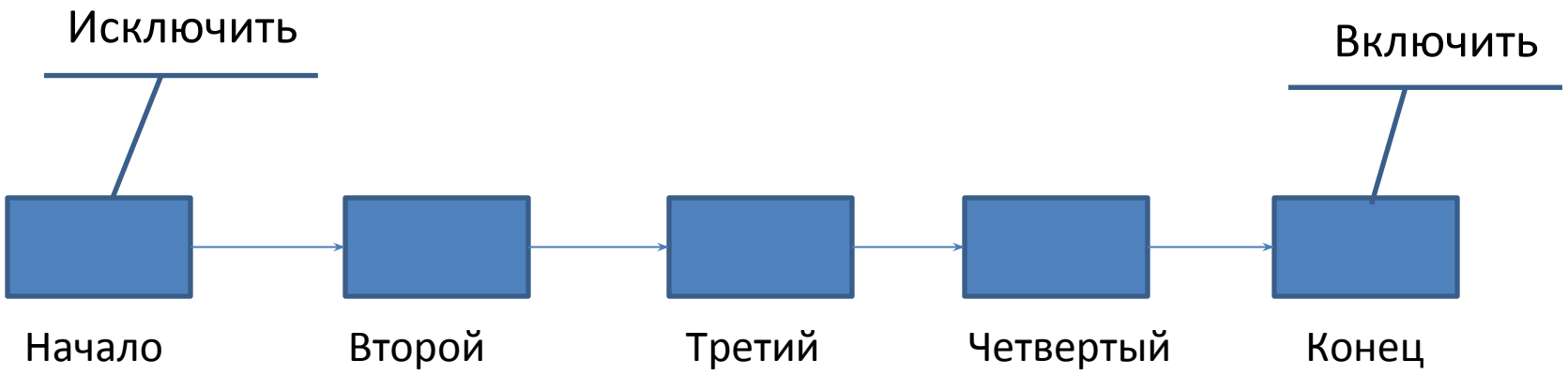
Стек

- это линейный список, в котором все включения и исключения (и всякий доступ) делаются в одном конце списка



Очередь

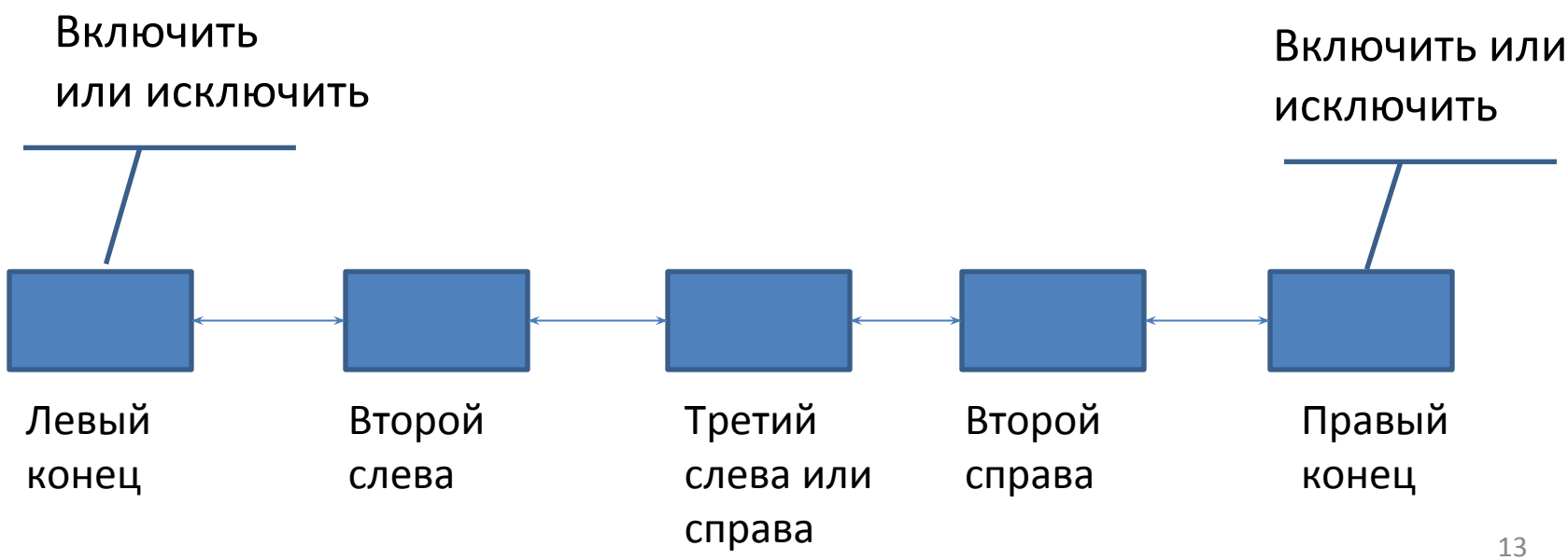
- это линейный список, в котором все включения производятся на одном конце списка, все исключения – на другом его конце.



Дек (double-ended queue)

очередь с двумя концами

- это линейный список, в котором все включения и исключения производятся на обоих концах списка



Стеки

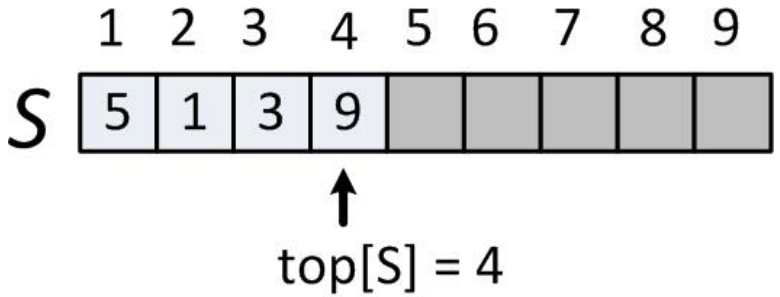
- push-down список
- реверсивная память
- гнездовая память
- магазин
- LIFO (last-in-first-out)
- список йо-йо

Операции работы со стеками

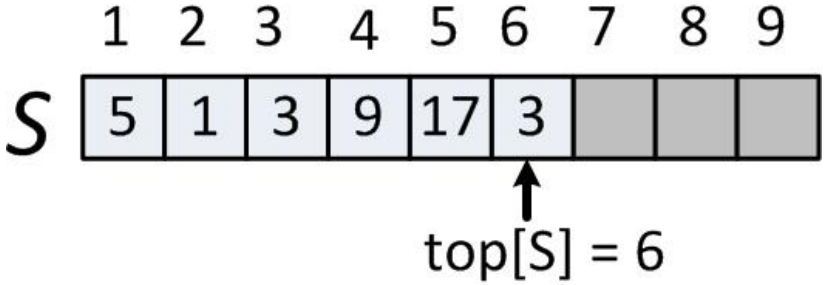
1. makenull (S) – делает стек S пустым
2. create() – создает стек
3. top (S) – выдает значение верхнего элемента стека, не удаляя его
4. pop(S) – выдает значение верхнего элемента стека и удаляет его из стека
5. push(x, S) – помещает в стек S новый элемент со значением x
6. empty (S) - если стек пуст, то функция возвращает 1 (истина), иначе – 0 (ложь).

Стеки. Реализация на масиве

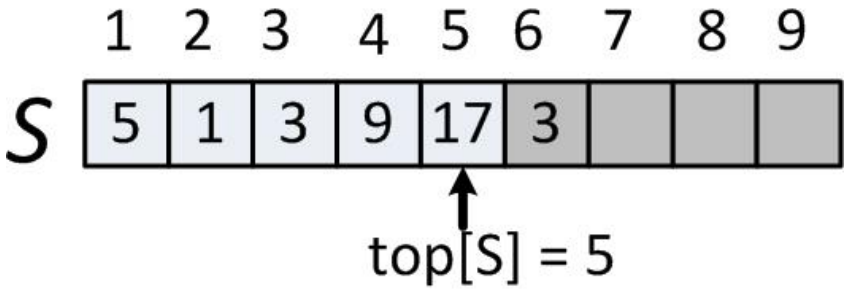
```
Stack_Empty (S)
{
      if top[S] = 0
      then return true
      else return false
}
Push (S, x)
{
      top[S] ← top[S] + 1
      S[top[S]] ← x
}
Pop (S)
{
      if Stack_Empty(S)
      then error “underflow”
      else
            top [S] ← top[S] – 1
            return S[top[S] + 1]
}
```



Push (S, 17); Push(S, 3);



Pop (S);

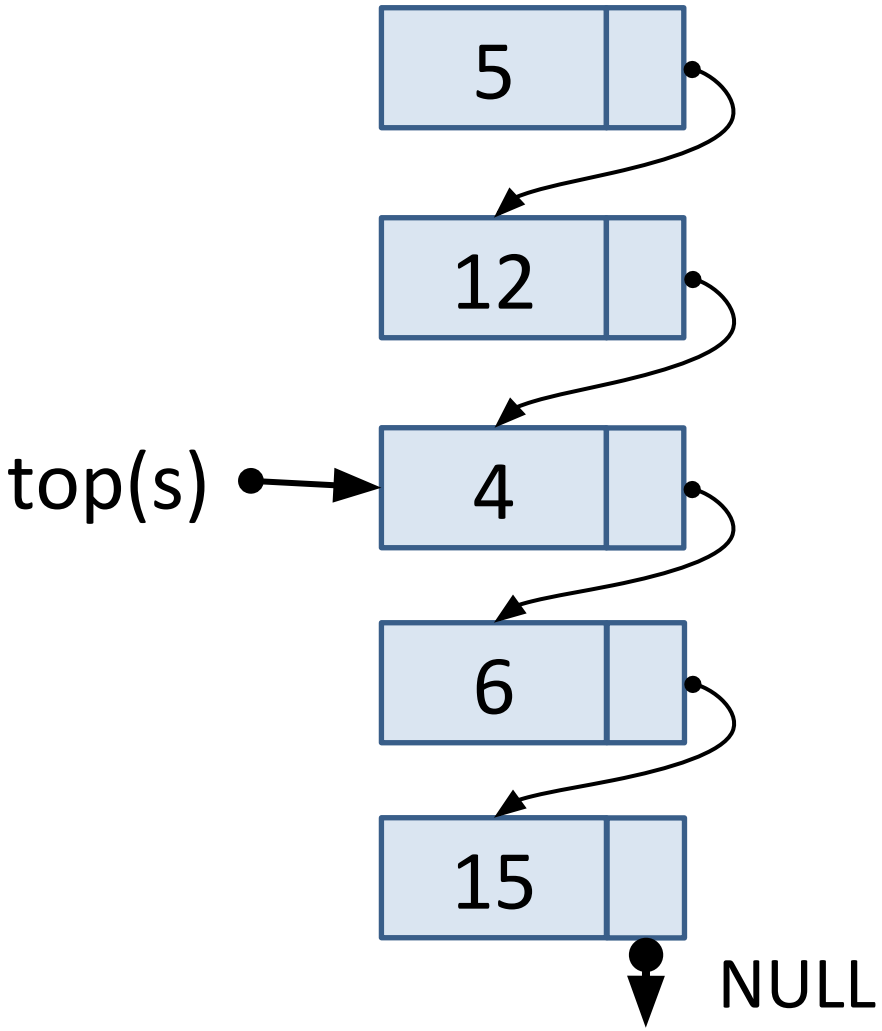


Реализация стека с помощью динамического вектора

top [S] = size [S] $\Rightarrow$ error (“overflow”)

или расширение массива (realloc)

Реализация стека с помощью односвязного списка

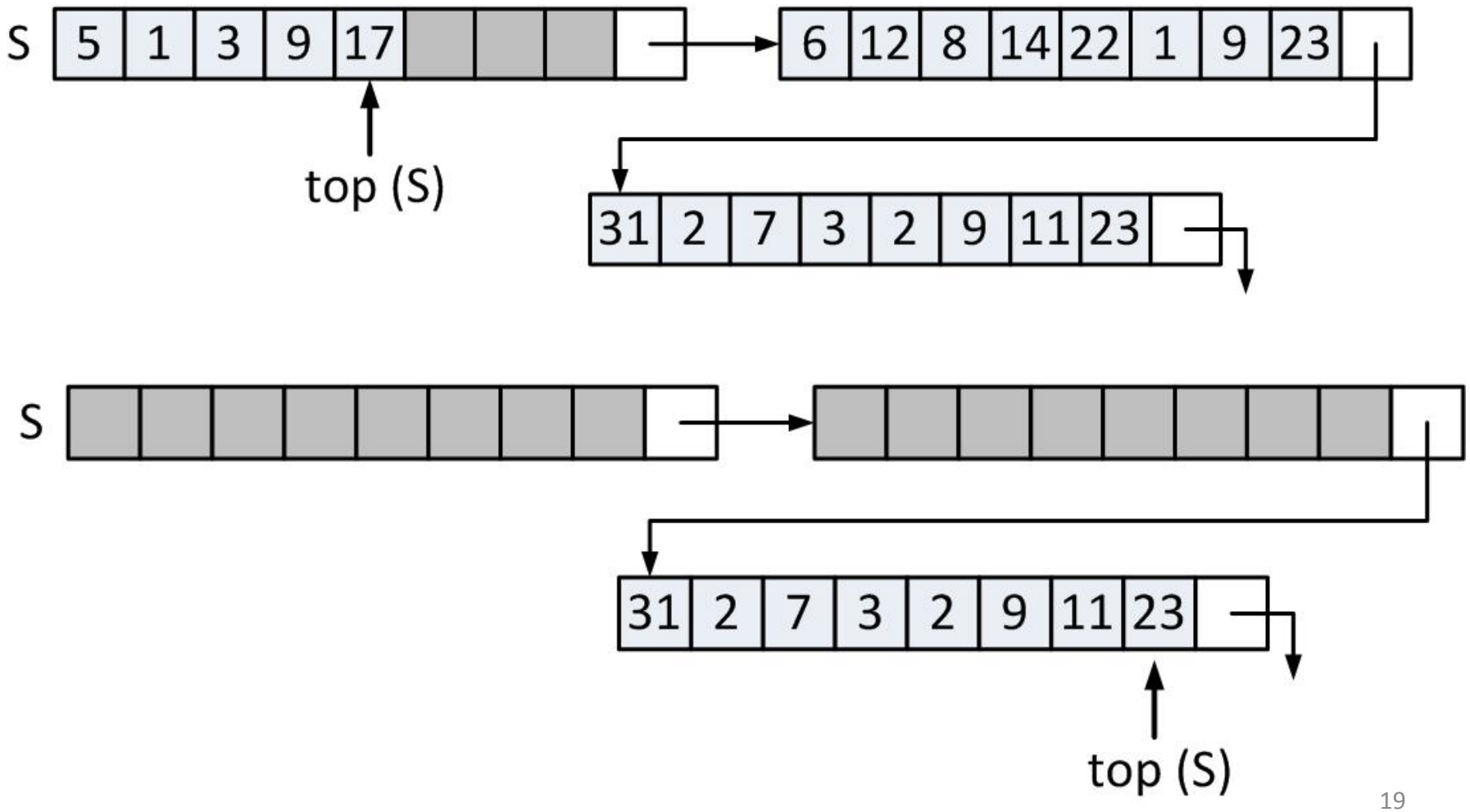


Push (S, 12);

Push (S, 5);

Pop (S);

Реализация стека с помощью списка векторов



Реализация стека (описание структуры данных)

На динамических списках

```
struct list {
    int data;
    struct list * next;
};
typedef struct stack {
     struct list *top;
} Stack;
```

На динамических массивах

```
typedef struct _Stack
{
        int size;    // размер максимальной памяти
        int top;     // текущий размер стека
        int * arr;   // адрес начала массива
} Stack;
```

Реализация стека (создание стека)

На динамических списках

```
Stack * create ( )
{
      Stack * S;
      S = (Stack *) malloc (sizeof (Stack));
      S->top = NULL;
      return S;
}
```

На динамических массивах

```
Stack * create  ( )
{
      Stack * s;
      s = (Stack *)malloc(sizeof(Stack));
      s -> top = 0;
      s -> size = 1;
      s -> arr = (int * )malloc(sizeof (int) * s->size);
      return s;
}
```

Реализация стека (сделать стек пустым, проверка на пустоту)

На динамических списках

```
void makenull (Stack  *S)
{
   while (S->top) {
        struct list *p = S->top;
        S->top = p->next;
        free(p);
   }
}

int empty (Stack *S)
{
        return (S->top == NULL);
}
```

На динамических массивах

```
void makenull (Stack * S)
{
      S->top = 0;
}

int empty (Stack * S)
{
      return ( S->top == 0 );
}
```

Реализация стека (продолжение)

На динамических списках

```
int top (Stack *S)
{
        return (S->top->data);
}

int pop(Stack *S)
{
        int a;
        struct list *p;
        p = S->top;
        a = p->data;
        S-> top = p->next;
        free(p);
        return a;
}
```

На динамических массивах

```
int top (Stack * S)
{
        return (S->arr[ S->top - 1]);
}

int pop(Stack *S)
{
        int a = S->arr [S->top - 1];
        S->top --;
        return a;
}
```

Реализация стека (продолжение)

На динамических списках

```
void push(Stack *S, int a )
{
    struct list *p;
    p = (struct list *) malloc ( sizeof (struct list));
    p->data = a;
    p->next = S-> top;
    S->top = p ;
}
```

На динамических массивах

```
void push(Stack * S, int a)
{
    if ( S->top == S->size )
    {
        S->size *= 2;
        S->arr = (int *)realloc(S->arr, sizeof(int)* S->size);
    }
    S->arr [ S->top++ ] = a;
}
```

Задача:
Нахождение максимума в стеке

S-стек	MAX(S)
15	15
10	10
2	7
7	7
3	5
5	5
1	1

Push (S, 10)

Push (S, 15)

Pop (S)

Виды записи выражений

- Префиксная (операция перед операндами)
- Инфиксная или скобочная (операция между операндами)
- Постфиксная или обратная польская (операция после операндов)

Примеры:

a + (f – b * c / (z – x) + y) / (a * r – k) - инфиксная

+a / + – f /*b c – z x y –*a r k - префиксная

a f b c * z x – / – y + a r * k – / + - постфиксная

Перевод из инфиксной формы в постфиксную

Вход: строка, содержащая арифметическое выражение, записанное в инфиксной форме

Выход: строка, содержащая то же выражение, записанное в постфиксной форме (обратной польской записи).

Обозначения:

числа, строки (идентификаторы) – операнды;

Знаки операций	Приоритеты операций
(	1
)	2
=	3
+, –	4
*, /	5

Алгоритм

Шаг 0:

Взять первый элемент из входной строки и поместить его в X.
Выходная строка и стек пусты.

Шаг 1:

Если X – операнд, то дописать его в конец выходной строки.
Если X = ‘(‘, то поместить его в стек.
Если X = ‘)‘, то вытолкнуть из стека и поместить в конец выходной строки все элементы до первой встреченной открывающей скобки. Эту скобку вытолкнуть из стека.
Если X – знак операции, отличный от скобок, то пока стек не пуст, и верхний элемент стека имеет приоритет, больший либо равный приоритету X, вытолкнуть его из стека и поместить в выходную строку. Затем поместить X в стек.

Шаг 2:

Если входная строка не исчерпана, то поместить в X очередной элемент входной строки и перейти на Шаг 1, иначе
пока стек не пуст, вытолкнуть из стека содержимое в выходную строку.

Перевод из инфиксной формы в постфиксную. Пример

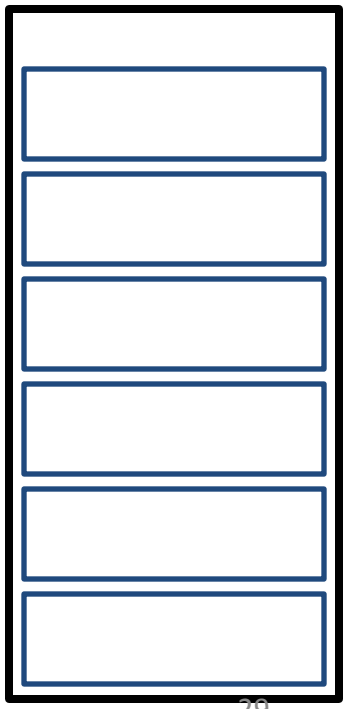
Входная строка:

a + (f – b * c / (z – x) + y) / (a * r – k)

Стек:

X =

Выходная строка:



Вычисления на стеке

Вход: строка, содержащая выражение, записанное в постфиксной форме.

Выход: число - значение заданного выражения.

Алгоритм:

Шаг 0:

Стек пуст.

Взять первый элемент из входной строки и поместить его в X.

Шаг 1:

Если X – операнд, то поместить его в стек.

Если X – знак операции, то вытолкнуть из стека два верхних элемента, применить к ним соответствующую операцию, результат положить в стек.

Шаг 2:

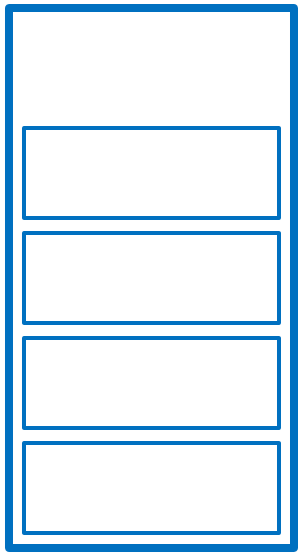
Если входная строка не исчерпана, то поместить в X очередной элемент входной строки и перейти на Шаг 1, иначе вытолкнуть из стека результат вычисления выражения.

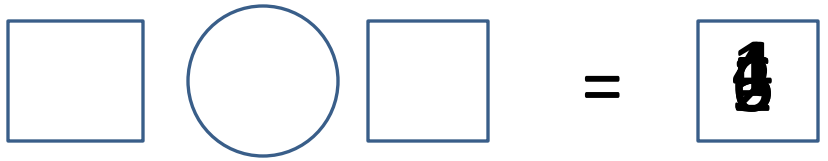
Вычисления на стеке. Пример

Входная строка:

5 2 3 * 4 2 // – 4 / + 1 –

Стек:





Очереди

- FIFO (first-in-first-out) –первый вошел, первый вышел

Можно реализовывать через:

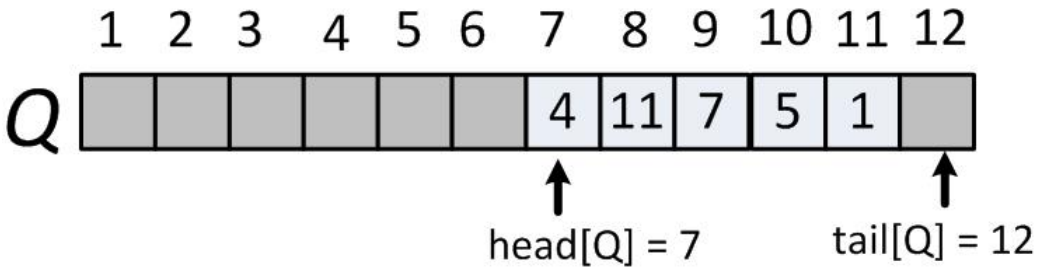
- Односвязный список
- Циклический массив
- Два стека

Операции работы с очередями

1. makenull (Q) – делает очередь Q пустой
2. create() – создает очередь
3. first (Q) – выдает значение первого элемента очереди, не удаляя его
4. dequeue(Q) – выдает значение первого элемента очереди и удаляет его из очереди
5. enqueue(x, Q) – помещает в конец очереди Q новый элемент со значением x
6. empty (Q) - если очередь пуста, то функция возвращает 1 (истина), иначе – 0 (ложь).

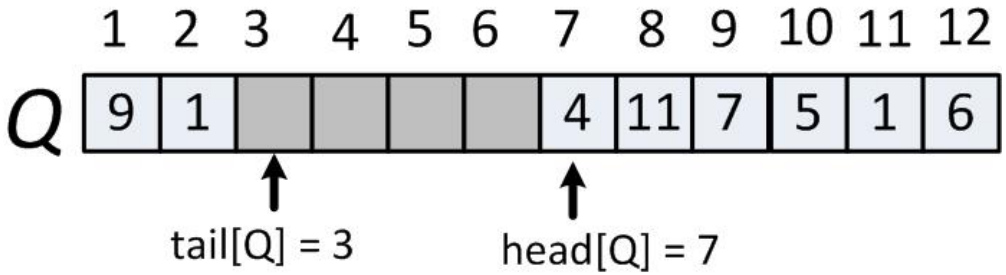
Реализация очереди на циклическом массиве (нет проверки на пустоту и переполнение!)

```
Enqueue (Q, x)
{
    Q[tail[Q]] ← x
    if tail[Q] = length[Q]
    then tail[Q] ← 1
    else tail[Q] ← tail[Q] + 1
}
```

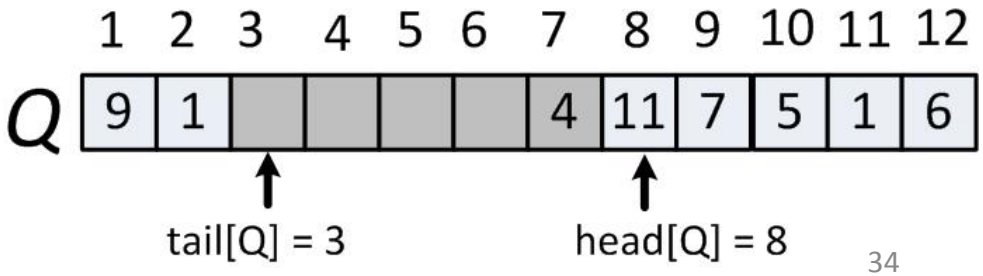



```
Dequeue (Q)
{
    x ← Q[head[Q]]
    if head[Q]= length[Q]
    then head[Q] ←  1
    else head[Q] ← head[Q]+1
    return x
}
```

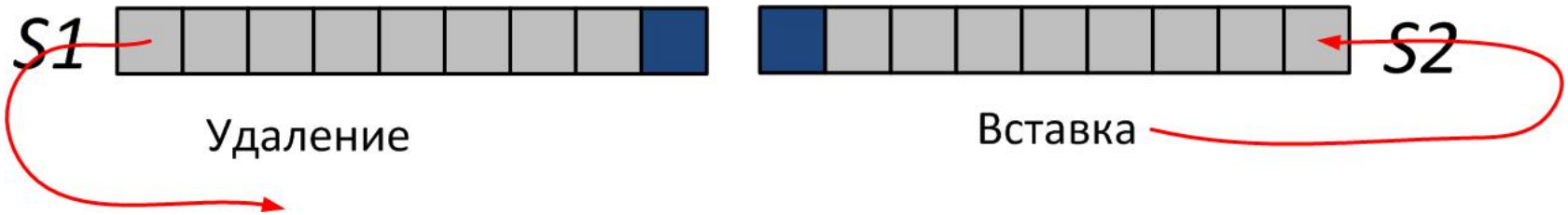
Enqueue(Q, 6); Enqueue(Q, 9); Enqueue(Q, 1);



Dequeue(Q);



Реализация очереди с помощью двух стеков




```
Enqueue (Q, x)
{
      Push (S2, x)
}
Dequeue (Q)
{
      if Stack_Empty (S1)
      then Переложить все содержимое из S2 в S1
      Pop(S1)
}
```

Время работы операции Dequeue в учётном смысле – O(1)

Как хранить максимум в очереди?

Реализация очереди на динамических списках

```
struct list
{
    int data;
     struct list * next;
};

typedef struct queue
{
    struct list *first;
    struct list *end;
} Queue;
```

Реализация очереди на массиве

```
typedef struct _Queue
{
    int size; // размер массива
    int first; // номер первого элемента очереди
    int leng; // длина очереди
    int * arr; // указатель на начало массива
} Queue;
```

Реализация очереди на массиве(продолжение)

```
Queue * create()
{
	Queue *q = (Queue *) malloc ( sizeof( Queue ) );
	q -> first = 0;
	q -> leng = 0;
	q -> size = 1;
	q -> arr = (int *)malloc(sizeof (int) * q -> size);
	return q;
}

int empty( Queue * q )
{
	return (q -> leng == 0);
}
```

Реализация очереди на массиве(продолжение)

```
void enqueue(Queue * q, int a)
{
    if ( q->leng == q->size )
    {
        q->arr = (int *)realloc(q->arr,  sizeof(int)* q->size * 2);
        if (q -> first > 0)
            memcpy( q->arr + q->size, q->arr,  (q->first) * sizeof(int) );
        q->size *= 2;
    }
    q->arr [ (q->first + q->leng++) % q->size ] = a;
}

int dequeue(Queue * q)
{
    int a = q->arr [q->first++];
    q->first %= q->size;
    q->leng --;
    return a;
}
```

Свойство persistence

Персистентными структурами данных мы будем называть такие структуры, что при всяком их изменении остается доступ ко всем предыдущим версиям этой структуры.

Immutable object – это объект, состояние которого нельзя менять после его создания

Mutable object – можно изменять

Персистентный стек

Пусть в начальный момент существует один пустой стек с номером 0;

n = 1 – количество стеков.

Требуется реализовать операции:

- push(i, x) — Добавить элемент x в стек номер i. Результирующий стек будет иметь номер n + 1.
- pop(i) — Вернуть последний элемент стека номер i и «выкинуть его из стека». Результирующий стек будет иметь номер n + 1.

Решение: эмуляция – копирование каждого нового стека.

Сложность одной операции – O(n)

Память – $O(n^2)$

Персистентный стек. Эффективная реализация

Вместо *n* копий стека хранить *n* первых элементов.

- push(x, i) — создает новый элемент со значением x, который ссылается на элемент с номером i, как на предыдущий элемент в стеке.
- pop(i) — возвращает значение, хранящееся в элементе с номером i и копирует элемент, предыдущий для него.