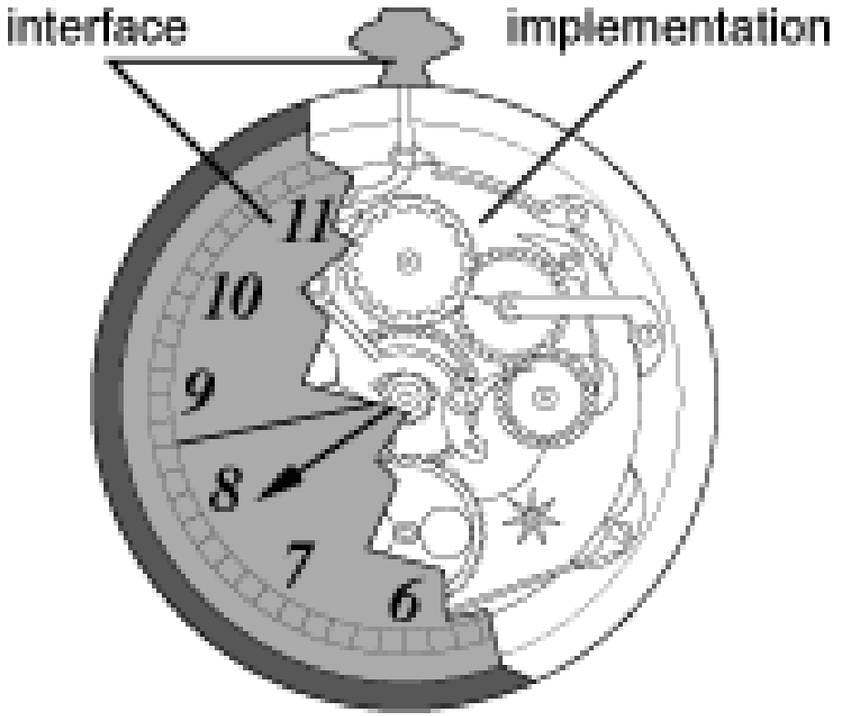
Структуры данных
Списки
Очередь

Основы языка C, лекция 13

Интерфейс и реализация

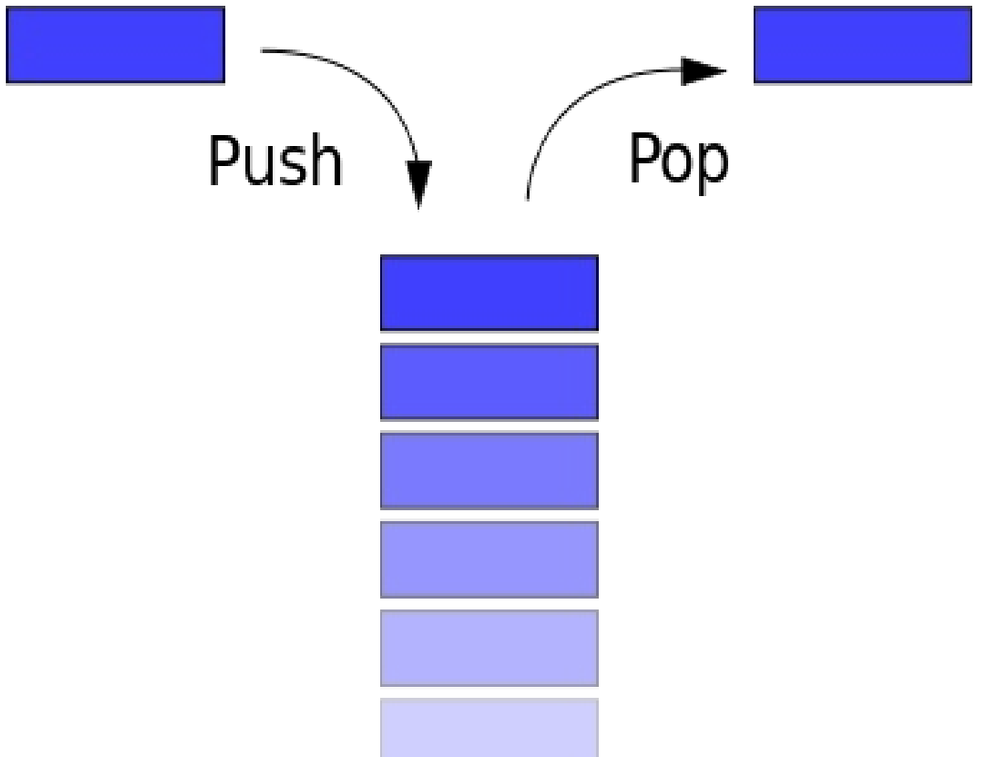
- Интерфейс — набор функций
- Реализация спрятана



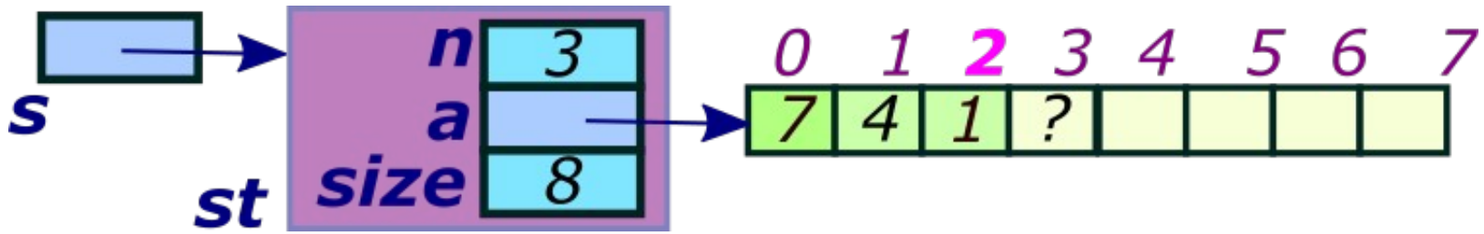
- Интерфейс
  - get_time
  - set_time
  - завести часы
- Реализация
  механические, электронные
  солнечные, свечи, водяные

Стек (stack)

- LIFO - last in, first out
  (последним пришел, первым вышел)
- **push** (x) — положить элемент x в стек
- x = **pop** ( ) - вынуть элемент из стека и вернуть его



на основе динамического массива

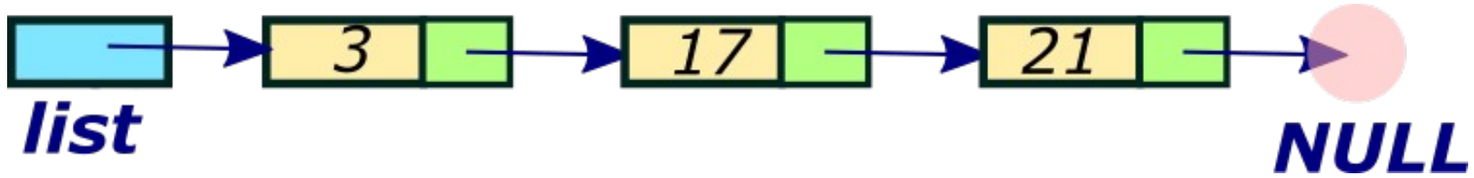


- typedef int **Data**;
  typedef struct {
  unsigned int **n**; // сколько данных в стеке
  Data * **a**; // храним данные в стеке
  size_t **size**; // емкость стека
  } **Stack**;

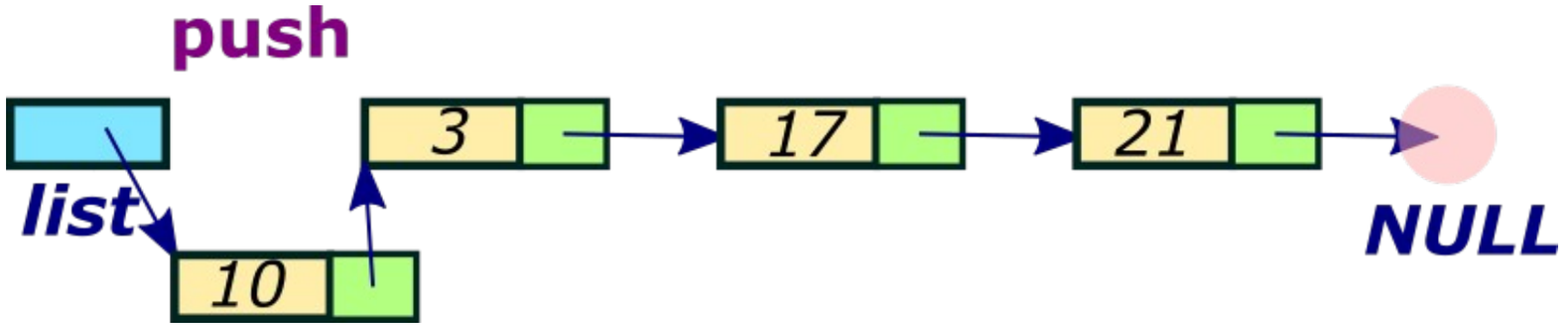
- интерфейс:
  void **push** (**Stack** * s, **Data** x); // O (1)
  Data **pop** (**Stack** * s); // O (1)
  void **init** (**Stack** * s); и так далее

Односвязный список

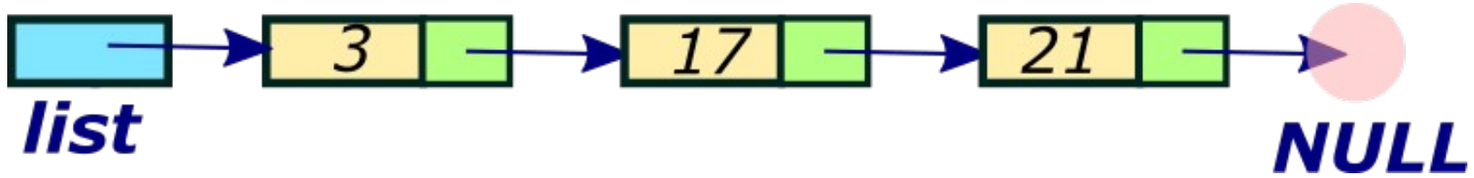
- Одинаковые элементы
- Каждый элемент указывает на следующий
  - Последний указывает на NULL



- На основе списка можно сделать стек,
  list указывает на вершину стека; push и pop - O (1)



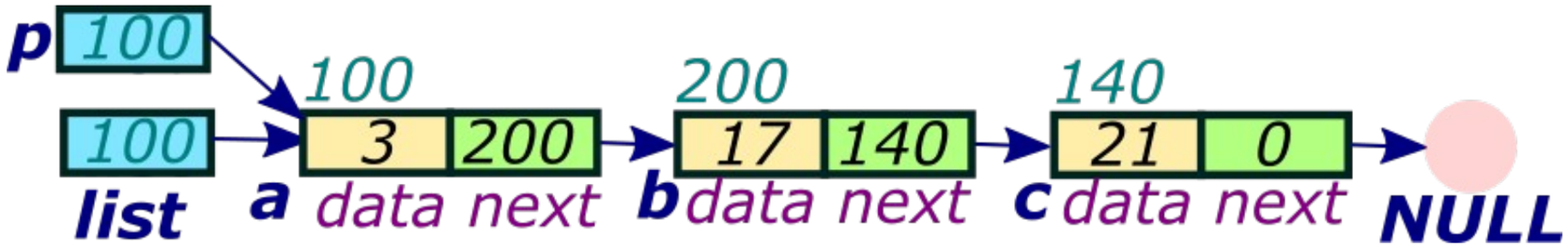
Односвязный список



- typedef int **Data**;
  typedef struct **Node** {
  Data data; // 1 число в 1 узле
  struct Node * next; // указатель на следующий
  } **Node**;
  Node * list = NULL; // сначала список пустой

- интерфейс:
  void **push** (**Node** ** list, **Data** x); // или так
  Node * **push** (**Node** * list, **Data** x); // или так
  Data **pop** (**Node** ** list);

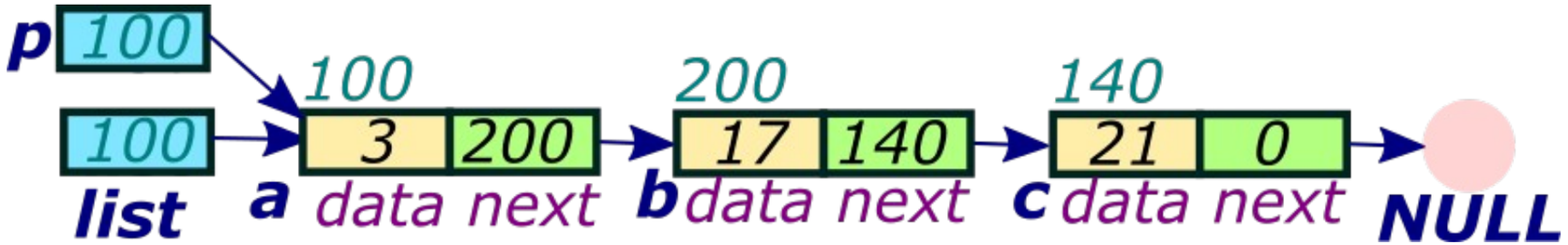
print(Node * list)



- Смоделируем (построим руками) список
  Node a = { 3 }, b = { 17 }, c = { 21 }, t = { 10 };
  Node * list = &a;
  a.next = &b; b.next = & c;   c.next = NULL;

- Напечатаем числа a.data, b.data, c.data используя p
  Node * p = list;   // 100
  printf("%d ", p → data);   p = p → next ;   // 200   **3**
  printf("%d ", p → data);   p = p → next ;   // 140   **17**
  printf("%d ", p → data);   p = p → next ;   // 0   **21**

print(Node * list)

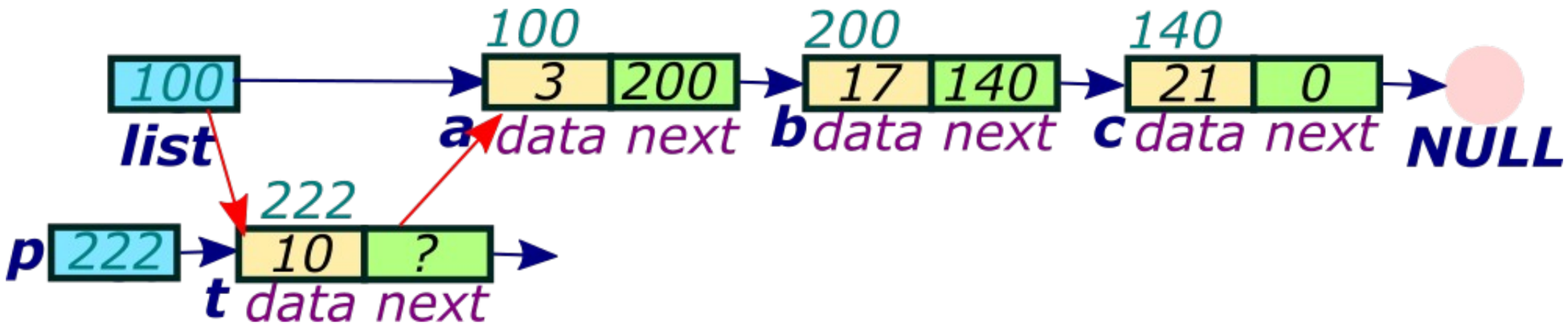


- Через цикл

```
Node * p;
for ( p = list;  p != NULL ; p = p → next)
    printf("%d ", p → data);
```

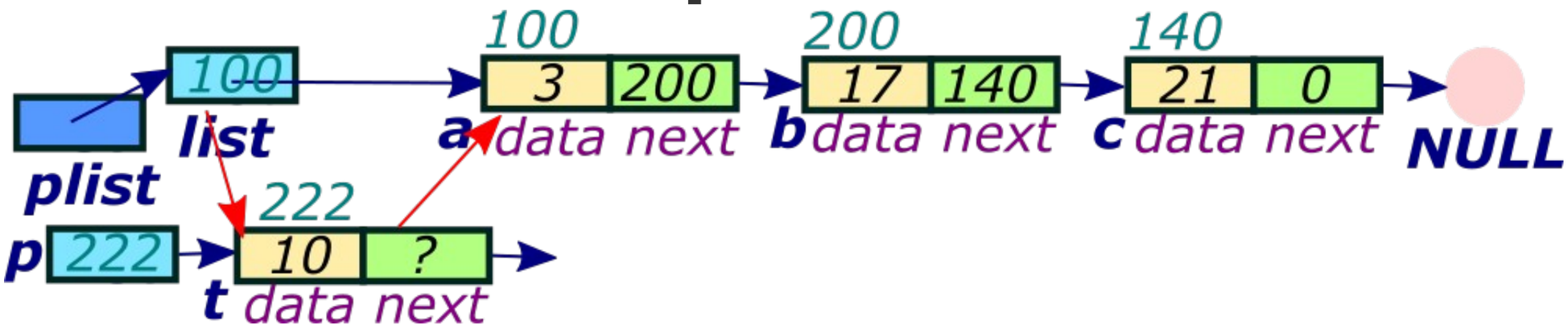
-
```
void print (Node * list) {          // в виде функции
    for (Node * p = list; p != NULL; p = p→next)
        printf("%d ", p→data);
    printf("\n");
}
```

push



- Добавим перед узлом **a** узел **t**: t.next = &a; list = &t;
- То же самое, только с list и указателем на t:
  p = &t;
  p → next = list;
  list = p; // содержимое list изменилось

void push(Node ** list, Data d)



- Так как внутри функции list изменится, нужно передавать указатель на эту переменную plist = &list

```
void push (Node ** plist, Data d) {
    Node * p = malloc(sizeof(Node));   // вместо t
    p → data = d;
    p → next = *plist;
    *plist = p;
}
```

Тестируем push

- int main() {
  Node * list = NULL;
  print(list); // ничего
  push(&list, 21);
  print(list); // 21
  push(&list, 17);
  print(list); // 17 21
  push(&list, 3);
  print(list); // 3 17 21

  return 0;
}

is_empty

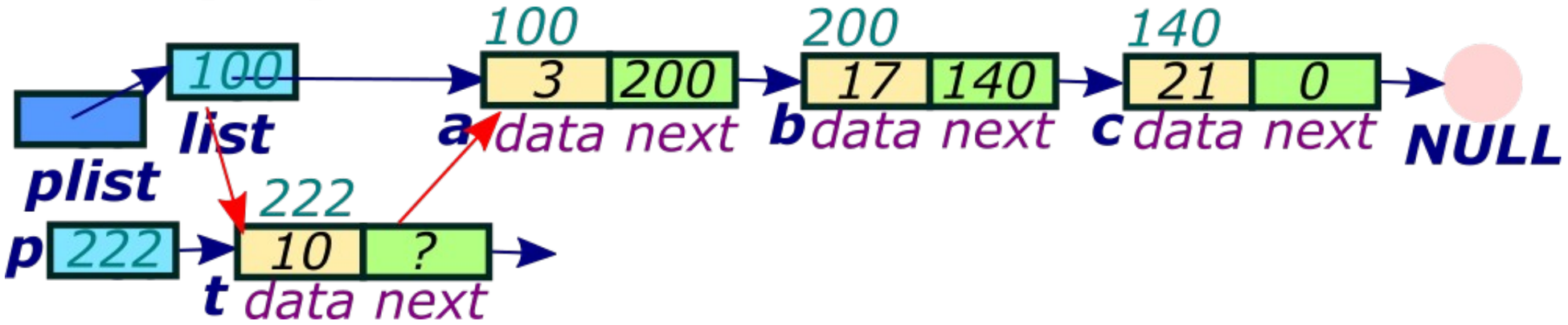
- Напишите тесты для проверки пустой список или нет
- Напишите прототип функции is_empty
- Реализуйте эту функцию

pop — напишем тесты

- тесты + valgrind

```
int main ( ) {
    Data test = {21, 17, 3, 10};
    Node * list = NULL;
    printf("Empty: %d\n", is_empty(list));
    for ( int i = 0; i < sizeof(test)/sizeof(test[0]); i++) {
        push(&list, test[i]);      print(list);
    }
    while( ! is_empty(list)) {
        int d = pop(&list);        print(list);
    }
    return 0;
}
```

Data pop(Node ** list)

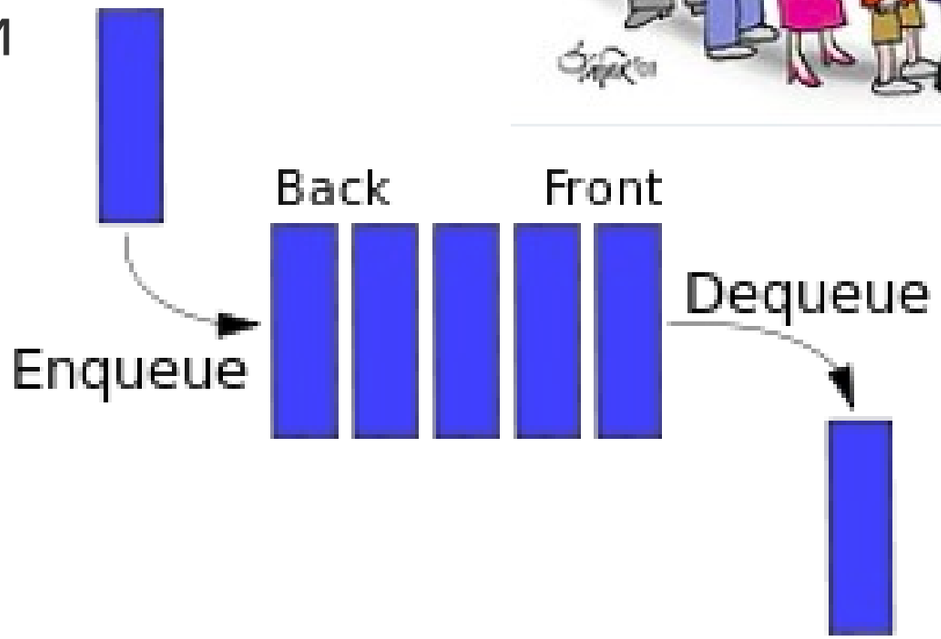


- Обратная функция, красные стрелки на синюю

```
void push (Node ** plist, Data d) {
    Node * p = *plist;
    Data res = p→data;
    *plist = p→next;
    free(p);
    return res;
}
```

Очередь

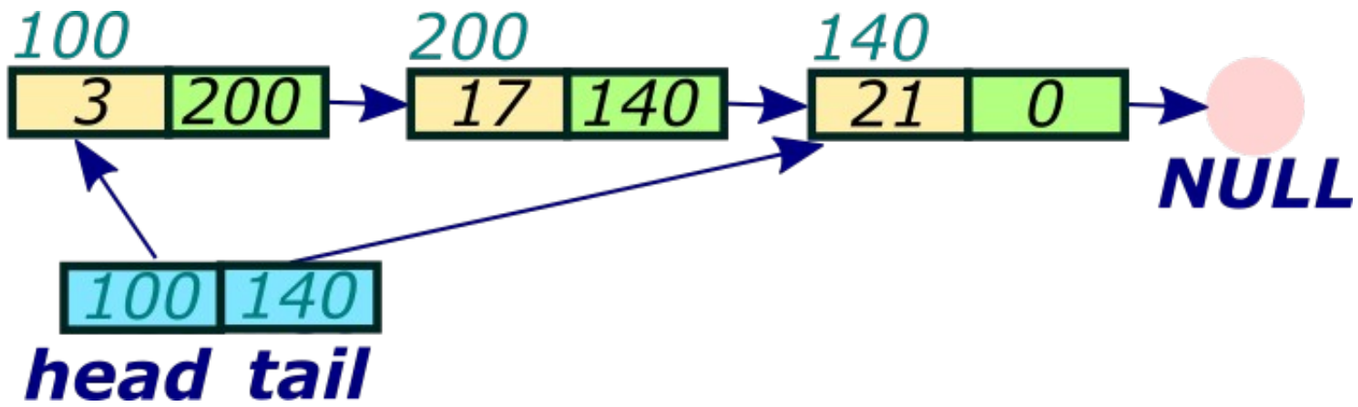
- LILO - last in, last out
  (последним пришел, последним вышел)
- **enqueue** (x) — добавить
  элемент x в очередь
- x = **dequeue** ( ) - удалить
  элемент из очереди



Очередь на основе односвязного списка

- list — указатель на конец очереди (back)
- enqueue = push O(1)
- dequeue — дойти до начала O(n) и удалить первый узел O(1)
- Неэффективно, так как O(n)

Очередь на основе односвязного списка



- enqueue это insert (&tail, d); O(1)
  dequeue это d = remove (&head); O(1)
  typedef struct {
  Node * tail;
  Node * head;
  } Query;

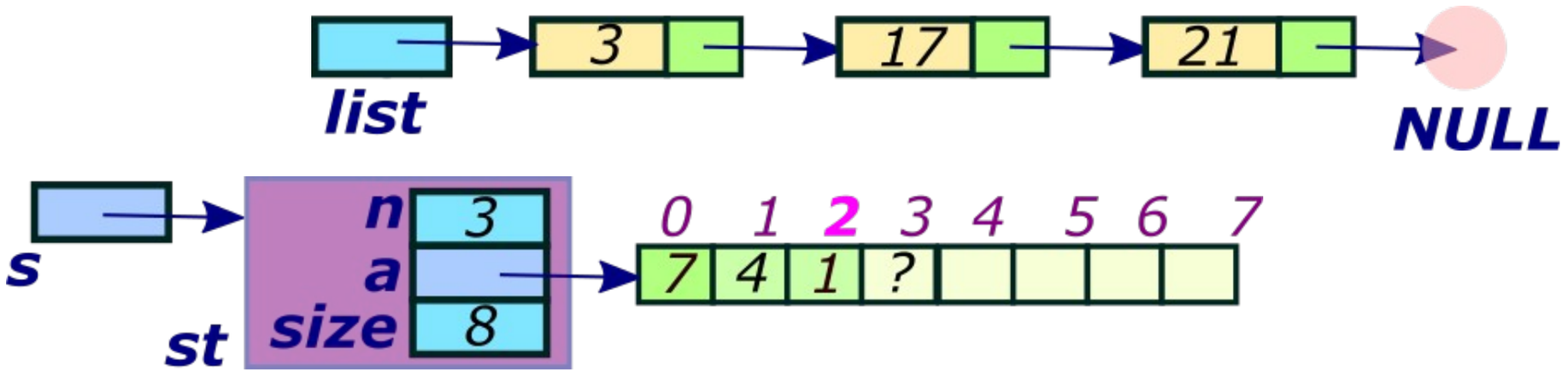
Очередь на основе массива

- Массив, индекс первого, индекс последнего, размер
- добавляем в конец
- убираем из начала
- когда очередь доходит до конца массива, сдвигаем голову очереди в q[0]
- Итерфейс: стек, очередь
  Реализация: массив, список

- Интерфейс
  - стек
  - очередь
- Реализация
  - динамический массив
  - односвязный список

Поиск элемента в структуре search

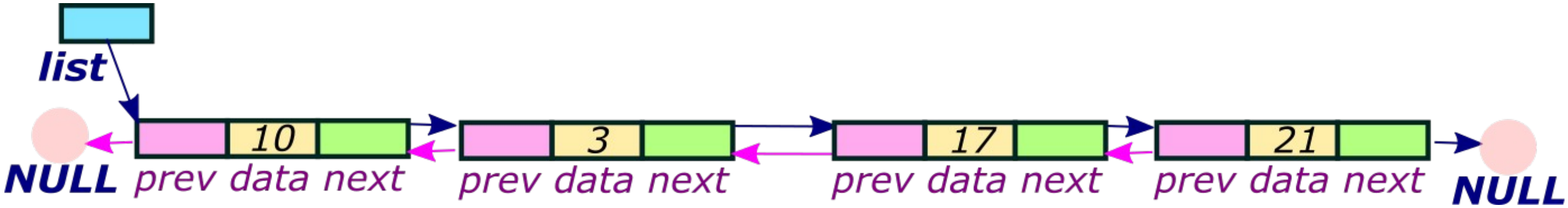
- Поиск — возвращаем указатель на элемент или индекс элемента (массив) или NULL (не нашли)
- Не упорядоченные данные O(N)
- Упорядоченные данные
  - односвязный список O(n) — надо до данных дойти
  - массив — возможен бинарный поиск



Вставка 1 элемента в структуру

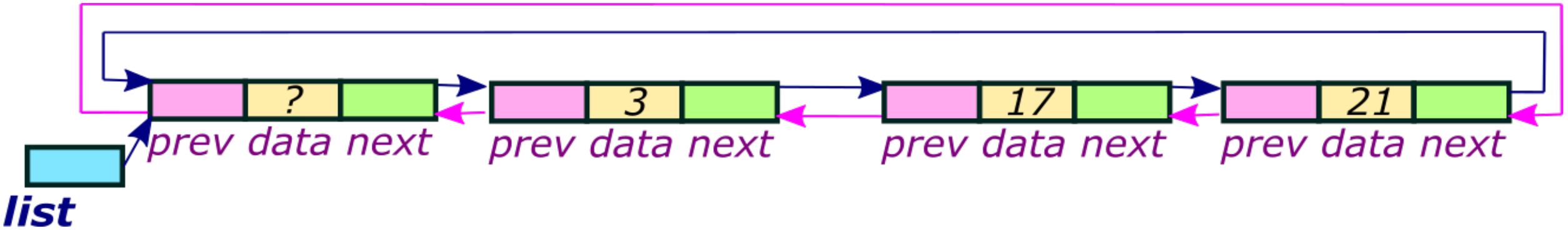
- Вставляем на произвольное место
  (раньше — в начало или конец)
- Вставляем в динамический массив insert(i, t)
  двигаем часть данных массива O(n)
- Вставляем данные после элемента p insert(p, t)
  для односвязного списка O(1)
- Вставляем данные перед элементом insert_before(p, t)
  для односвязного списка O(n)
- Хотим вставку перед и обход в обратном направлении — храним в каждом узле **next** и **prev**
  двусвязный список

Двусвязный список



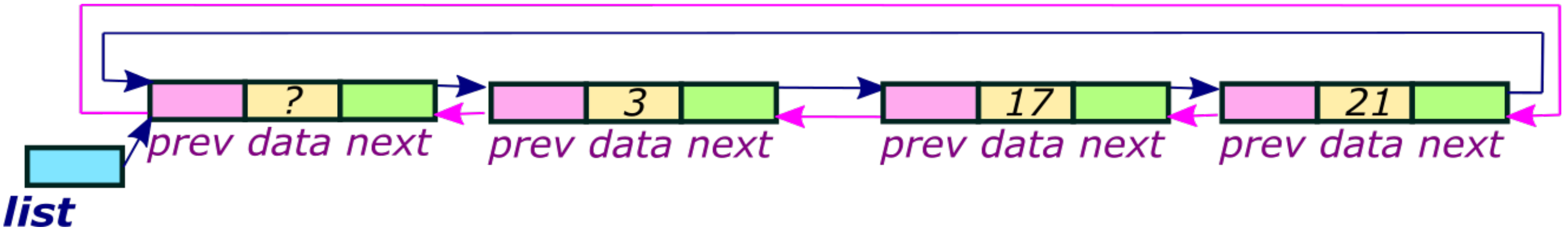
- Классический (с двумя открытыми концами)
- Сложная реализация вставки элемента, нужно учитывать 3 варианта: в середину, начало, конец
- Чаще всего описана в учебниках
  самая сложная в реализации

Двусвязный список циклический



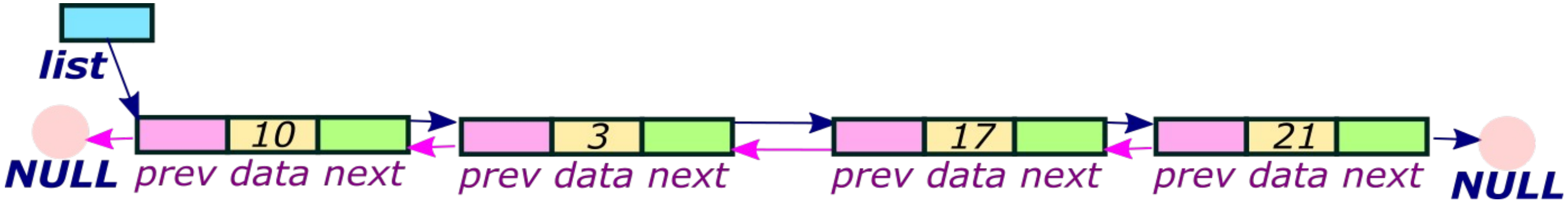
- Циклический
  - next последнего указывает на первый
  - prev первого указывает на последний
- Легче делать insert — всегда в середину

Двусвязный список циклический с барьерным элементом



- Циклический
- Бусинки и замочек (барьер) — особый элемент, где НЕ хранятся данные

Описание типа данных для 1 узла

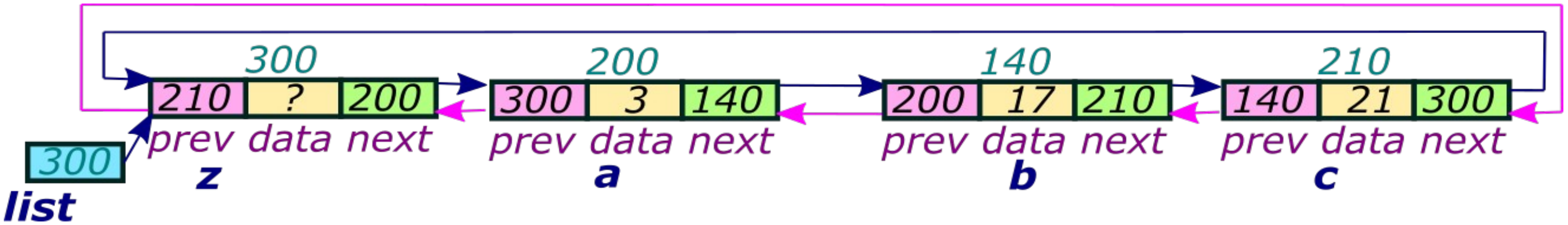


- Любой двусвязный список
- typedef int **Data**;
  typedef struct **Node** {
  Data data; // данные в узле
  Node * prev; // предыдущий узел
  Node * next; // следующий узел
  } **Node**;

API двухсвязного списка

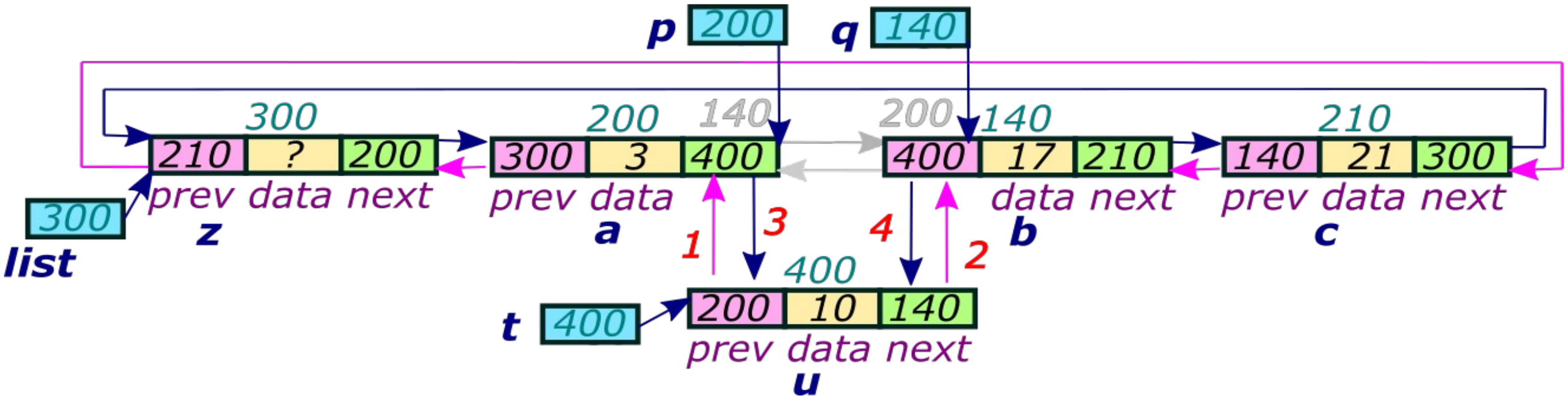
- void **print** (**Node** * list); // O (n)
  int **is_empty** (**Node** * list);
  Node * **create** ( );
  void **init** (**Node** * list);
  void **destroy** (**Node** * list);
- void **insert** (**Node** * p, **Node** * t); // без malloc
  void **insert_before** (**Node** * p, **Node** * t);
  Node * **remove** (**Node** * t);
- void **push** (**Node** * p, **Data** d); // с malloc
  void **push_before** (**Node** * p, **Data** d);
  Data **delete** (**Node** * t);

Модель, print, print_back, print_debug



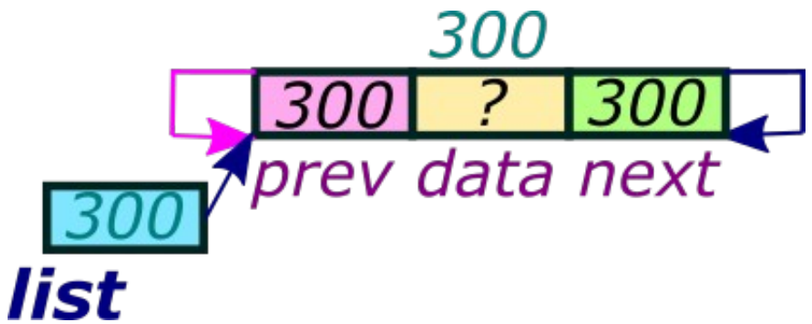
- Node z, a = {3}, b = {17}, c = {21};
  Node * list = &z;
  z.next = &a; z.prev = &c;
  a.next = &b; a.prev = &z;
  b.next = &c; b.prev = &a;
  c.next = &z; c.prev = &b;
- print — напечатать числа 3 17 21 имея только list

void insert (Node * p, Node * t)



- Вставляем узел t после узла p — insert(p, t);
- Вставляем узел t перед узлом q — insert_before(q, t);
- Удаляем узел t — remove_node(t);

void init (Node * list)
int is_empty(Node * list)



- Нет других элементов, кроме "замка"
- Как лучше?
- int is_empty(Node * list) {
  return list→prev == list→next;
  или
  return list→prev == list;
  }

foreach — для каждого узла в списке

- void print (Node * list) {
  for (Node * p = list→next; p != list; p = p→next)
  printf("%d ", p→data);
  }
- void print1 (Data d) { printf("%d ",d); }
- foreach(list, print1);
- void foreach (Node * list, void (*func)(Data d));
- void foreach (Node * list, void (***func**)(Data d)) {
  for (Node * p = list→next; p != list; p = p→next)
  func (p → data);
  }

foreach — для каждого узла в списке

- void print (Node * list, FILE * fout) {
  for (Node * p = list→next; p != list; p = p→next)
  f**printf(fout, "%d ", p→data);**
  }
- void print1 (Data d, FILE * fout) {
  // FILE * fout = (FILE*) arg;
  fprintf(fout, "%d ",d);
  }
- foreach(list, print1, stdout);
- void foreach (Node * list,
  void (*func)(Data d, void * param), void * param);

foreach — для каждого узла в списке

- void print1 (Data **d**, void * **fout**) {
  fprintf((FILE*)**fout**, "%d ", **d**);
  }
- foreach (list, print1, stdout);
- void foreach (Node * list, void (***func**)(Data d, void * a), void * ar)
  {
  for (Node * p = list→next; p != list; p = p→next)
  func (p → data, ar);
  }

foreach — для каждого узла в списке

- void sum (Node * list, Data * pres) {
  for (Node * p = list→next; p != list; p = p→next)
  ***res = *pres + p→data;**
  }
- void sum_it (Data d, void * param) {
  (Data)param += d;
  }
- Data list_sum (struct Node * list) {
  Data res = 0;
  foreach (list, sum_it, &res);
  return sum;
  }

foreach — для каждого узла в списке

- void foreach (Node * list, void (*func)(Data d, void * param), * pres) {
  for (Node * p = list→next; p != list; p = p→next) {
  func (p→data, param
  }
  }
- void sum1 (Node * p, void * param) {
  Data * psum = (Data *) param
  *psum = *psum + p→data;
  }
- Data res;
  foreach(sum1, &res);