Задача.

Реализовать структуру данных, выполняющую запросы:

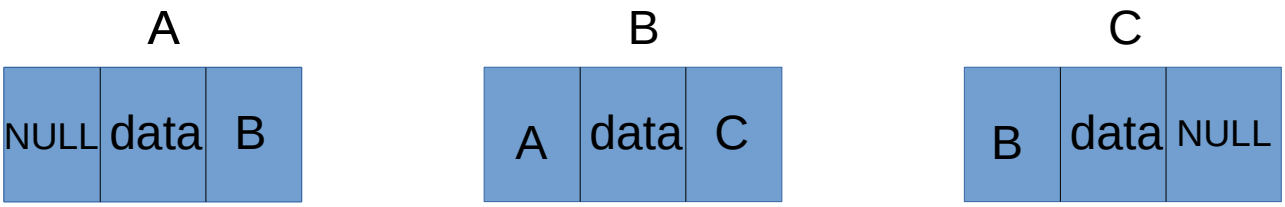
- **Insert <position> <data>** */*вставка данных в указанную позицию */*
- **Erase <position>** */*удаление данных из указанной позиции */*

Требуемая память: меньше, чем в двусвязном списке.
Время работы: быстрее, чем в односвязном спике.

Пример

Insert(1, 1)	1
Insert(2, 2)	1 2
Insert(3, 3)	1 2 3
Erase(2)	1 3

Двусвязный список



A, B, C – адреса соответствующих узлов(Node)

```
struct Node{
	long long data;
	struct Node* prev;
	struct Node* next;
};
```

Для 64 битного компилятора:
sizeof(long long) = 8 байт
sizeof(struct Node*) = 8 байт

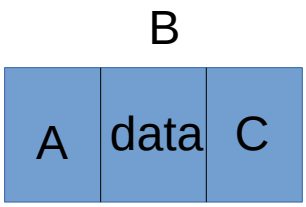
Каждый узел занимает:
sizeof(long long) + 2 * sizeof(struct Node*) = 24 байта

Xor - список

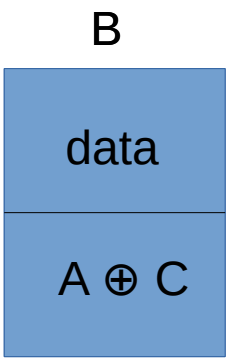
Вместо того чтобы хранить в каждом Node указатели на предыдущий и следущий узел,

будем в каждом Node хранить только один составной адрес — результат выполнения операции XOR над адресами предыдущего и следующего элементов списка.

Двусвязный список:



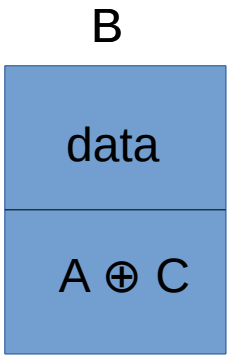
Xor - список



A, B, C – адреса соответствующих узлов(Node)

Узел xor списка

```
struct Node{
	long long data;
	struct Node* ptr;
};
```



Для 64 битного компилятора:
sizeof(long long) = 8 байт
sizeof(struct Node*) = 8 байт

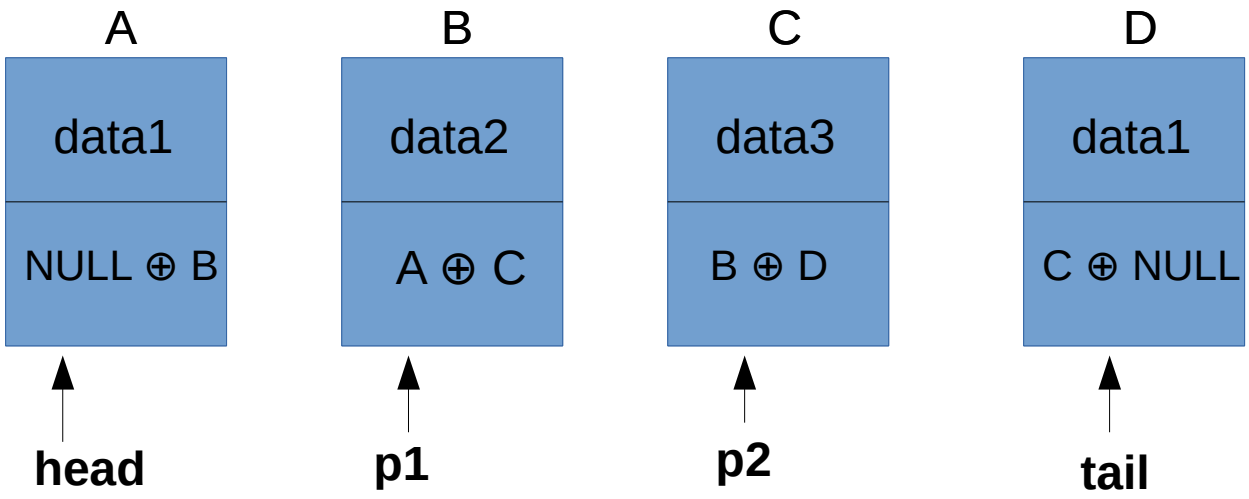
Каждый узел занимает:
sizeof(long long) + sizeof(struct Node*) = 16 байта

В нашей реализации каждый узел xor списка **в 1, 5 раза эффективнее по памяти** узла двусвязного списка.

Xor - список

```
struct Node{
long long data;
struct Node* ptr;
};
```

```
struct Xorlist{
  struct Node *p1;
  struct Node *p2;
  struct Node *head;
  struct Node *tail;
  long long size;
  long long curr_pos;
};
```



Для работы с xor списком **необходимы указатели на два соседних узла**. (p1 и p2)

Будем считать, что **p1 указывает на текущий узел** (curr_pos)
Нумеруем узлы с нуля.

На нашем примере
size = 4
curr_pos = 1

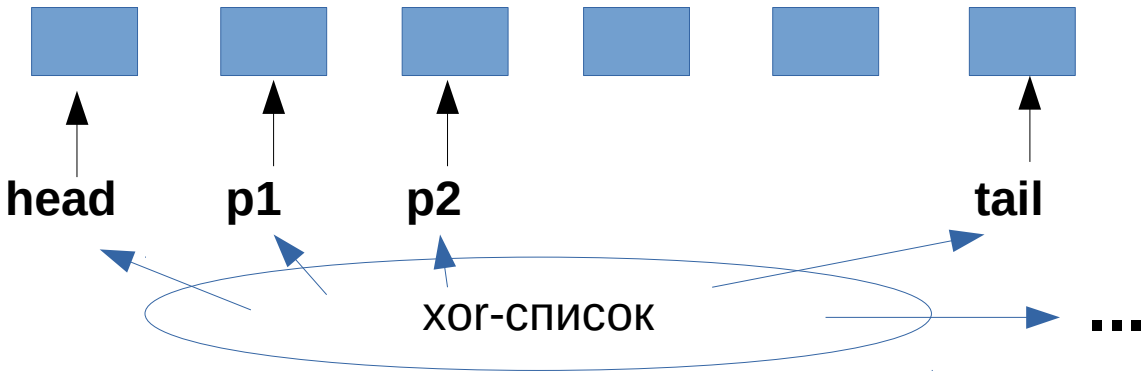
Вспомним
xor(a, a) = 0
xor(a, 0) = xor(0, a) = a

Как работать с xor списком? Функции prev и next

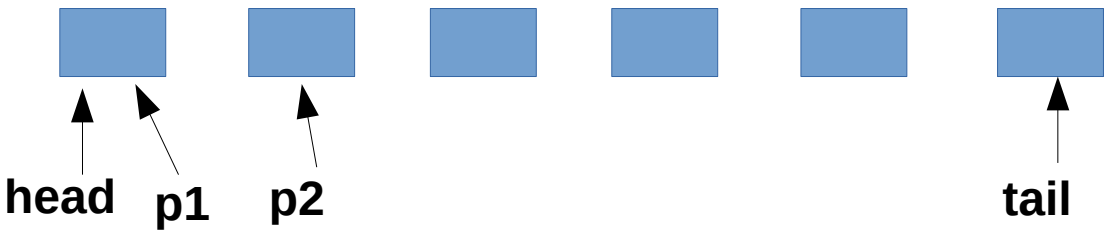
Зачем нужны и что делают?

- Эти функции нужны **для перемещения** по нашему xor-списку.
- Сам список **они не меняют**, они лишь работают с указателями p1 и p2 в структуре.
- На них базируется работа других функций нашей реализации.

Пример: Пусть имеется список из 6 элементов и нам нужно обратиться к предыдущему элементу:



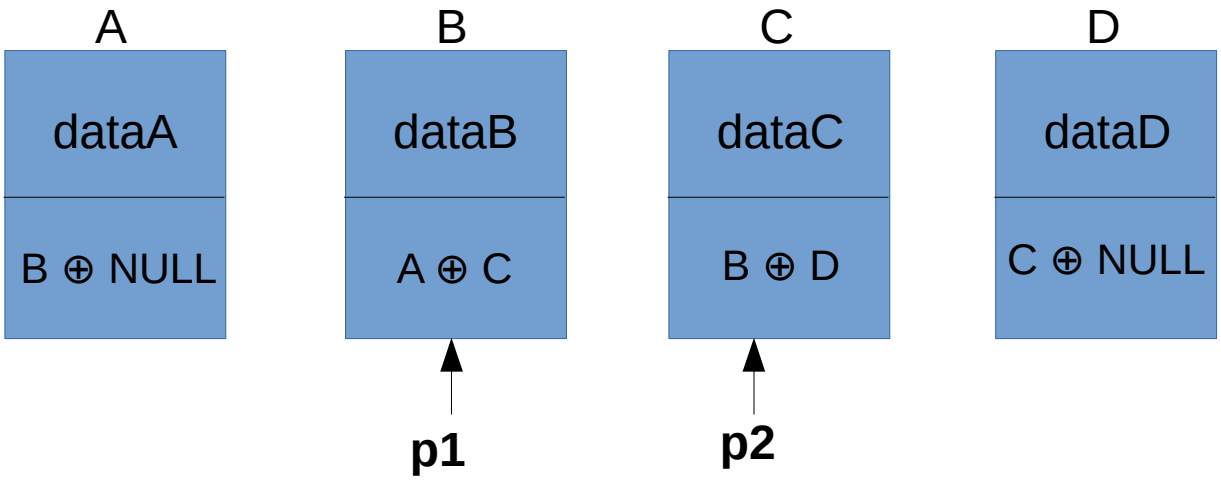
Тогда после выполнения prev(l), мы получим:



Алгоритм next

Для функции next (для prev аналогично)
1) Получив указатель на xor-список вначале мы проверяем, находимся ли мы в конце списка, ведь если так, то мы не сможем перейти на след. элемент.
2) Далее указателю p1 присваиваем p2
3) Вычисляем адрес следующего элемента списка с помощью дополнительной функции “xor_ptr” и указателю p2 присваиваем этот адрес.
4) Увеличиваем текущую позицию xor-списка, показывающую, на каком элементе мы находимся

Ниже представлен рисунок, где наглядно показывается как вычисляется адрес след. элемента.

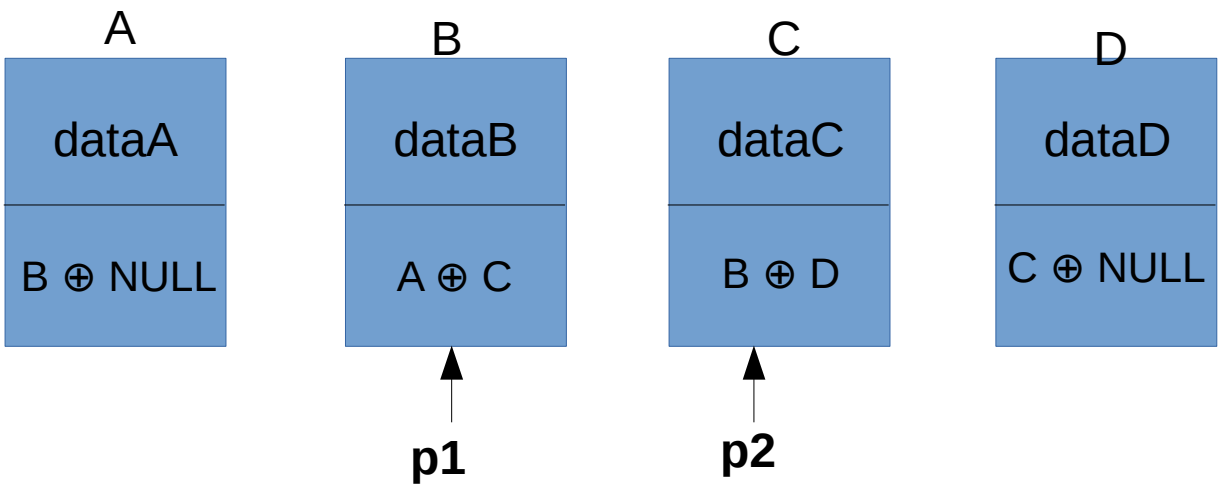


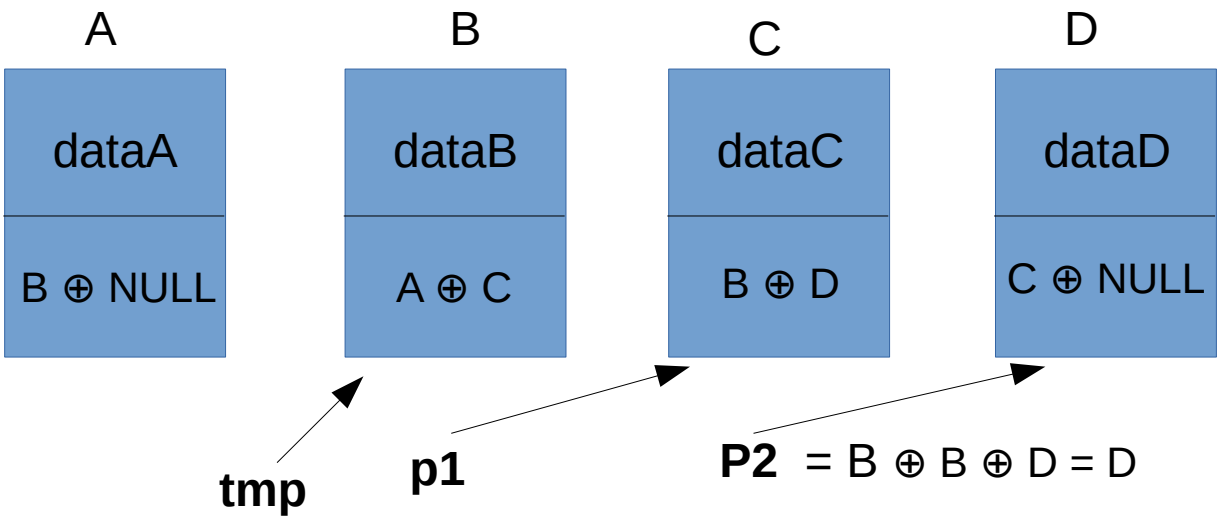
*Для вычисления адреса звена, следующего за **p2**:*
p1 ⊕ p2->prt = B ⊕ B ⊕ D = 0 ⊕ D = D

*Для вычисления адреса звена перед **p1**:*
p1->ptr ⊕ p2 = (A ⊕ C) ⊕ C = A

реализация функций next

```
int next(struct Xorlist* l){

        …

        struct Node *tmp = l->p1;
        l->p1 = l->p2;
        l->p2 = xor_ptr(tmp, l->p2->ptr);
        ++(l->curr_pos);
        return 0;
}
```





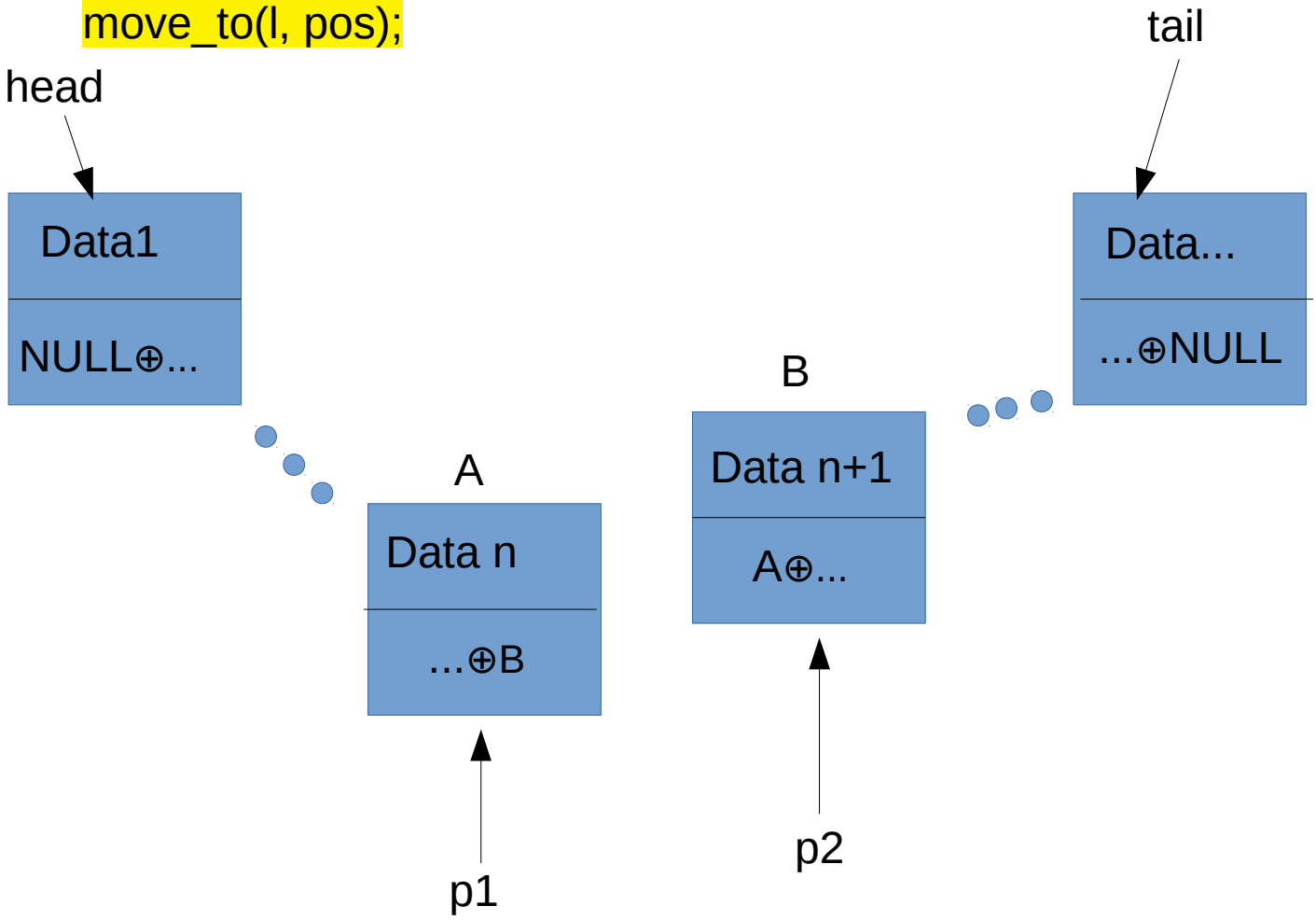
int move_to(struct Xorlist *lst, long long pos)

Функция move_to перемещает указатели p1 и p2 на позиции pos - 1 и pos соответственно.

После выполнения lst->curr_pos = pos - 1

Обозначим для удобства n = pos - 1

move_to(l, pos);



После успешного выполнения функции

Гарантируется, что элемент с номером pos присутствует в списке

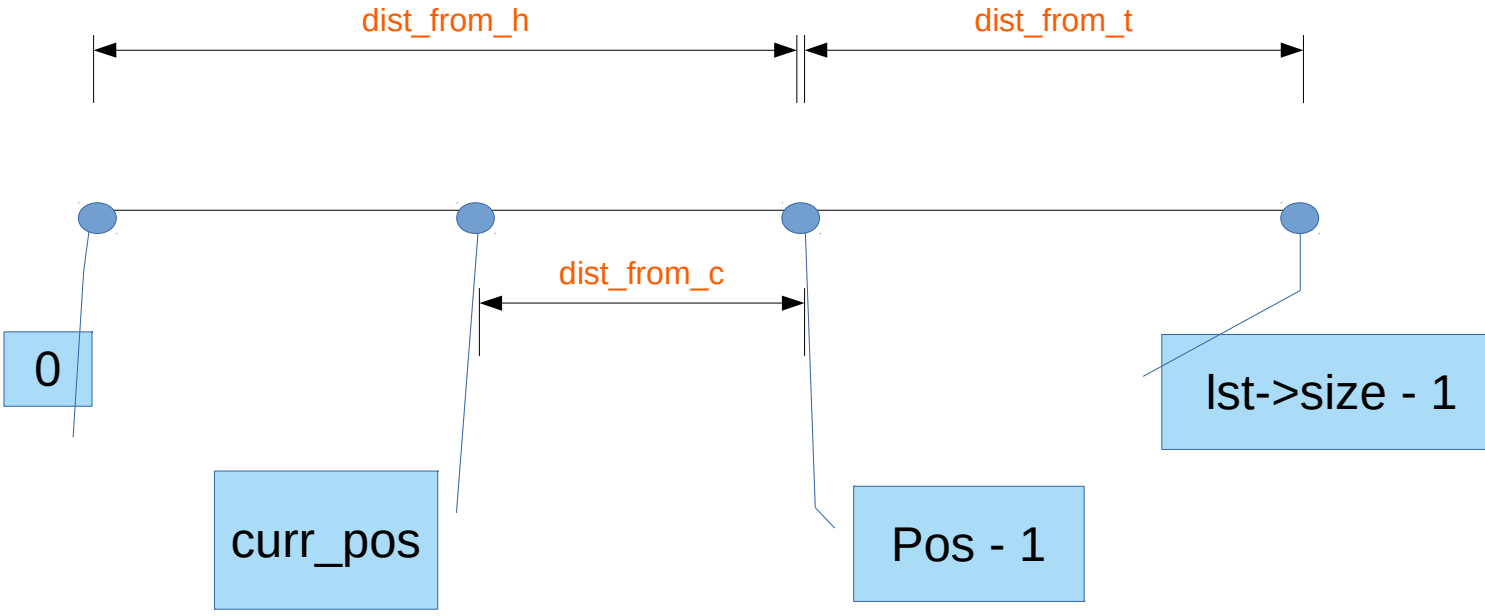
```
if(l->curr_pos == (pos - 1)){
        return 0;
    }
```

Если список уже установлен на нужную позицию, никаких действий не требуется

Благодаря тому, что такая реализация позволяет «пробегать» по нашему списку в обе стороны, мы можем уменьшить количество переходов, необходимых чтобы добраться до нужного элемента.

```
long long dist_from_h = pos - 1;

long long dist_from_t = l->size - pos;

dist_from_c = abs(l->curr_pos - (pos — 1));
```

Найдем количество переходов от начала списка до pos (при движении к концу), от конца до pos (при движении в «обратную» сторону) и от текущего элемента до pos

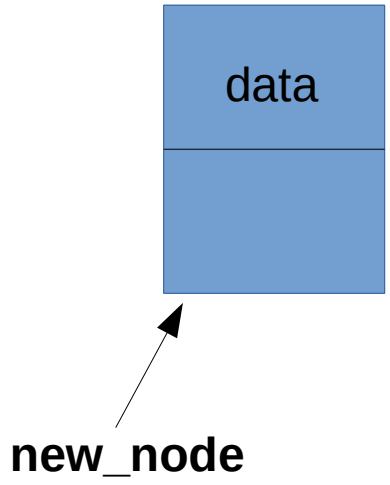


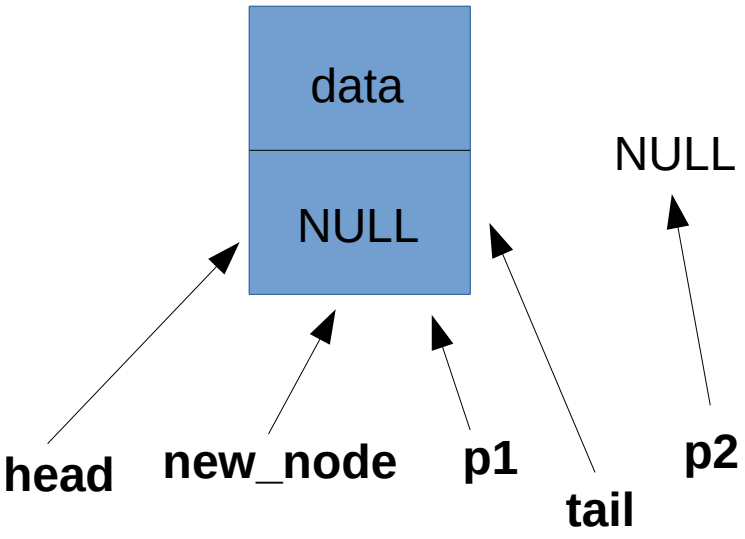
Далее выбираем минимальное из этих трех значений, и выполняем переходы в нужную сторону (при помощи функций prev() и next()) до тех пор, пока не дойдем до нужного элемента

Вставка в xor-список.
Функция insert
Часть 1.

- Возвращаемое значение равно 0, при успешном выполнении, и 1 — если некорректный запрос.
- После вставки узла, p1 указывает на вставленный узел.

```
int insert(struct Xorlist* l, long long pos, long long data){

    struct Node *new_node = malloc(sizeof(struct Node));
    new_node->data = data;

    /* если список пуст */
    if(l->size == 0){
        l->p1 = new_node;
        l->p1->ptr = NULL;
        l->p2 = NULL;
        l->curr_pos = 0;

        l->head = l->p1;
        l->tail = l->p1;
    }`
```

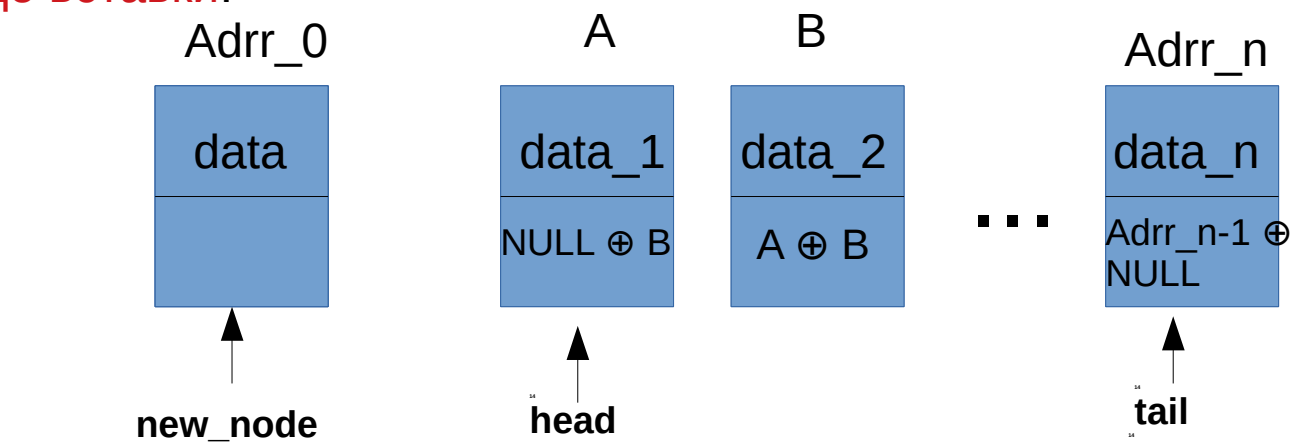




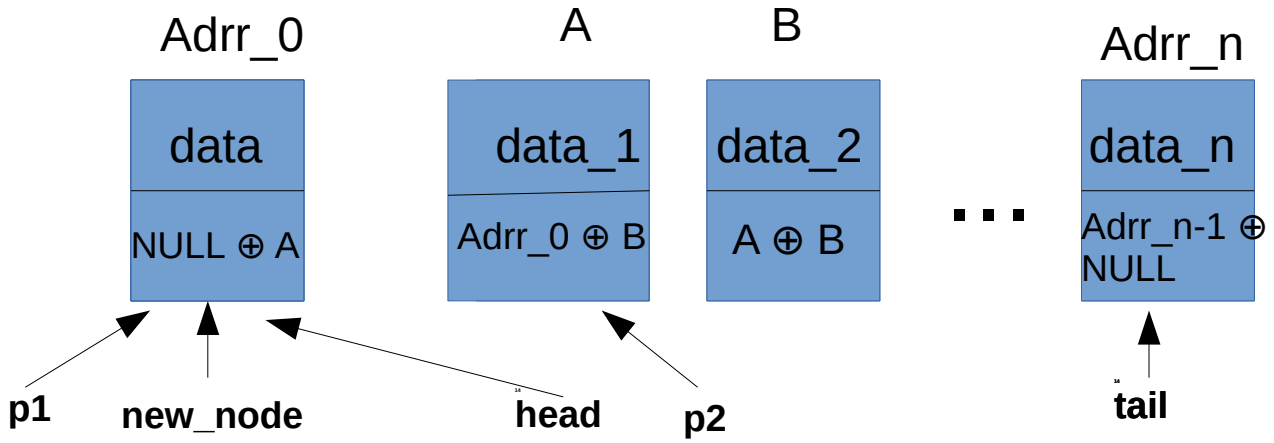
Вставка в xor-список.
Функция insert
Часть 2.

```
/* в начало списка */
        else if((pos - 1) == 0){
                new_node->ptr = xor_ptr(NULL, l->head);
                l->head->ptr = xor_ptr(new_node, l->head->ptr);
                l->head = new_node;

                l->p1 = l->head;
                l->p2 = l->head->ptr;
                l->curr_pos = 0;
        }
```

До вставки:



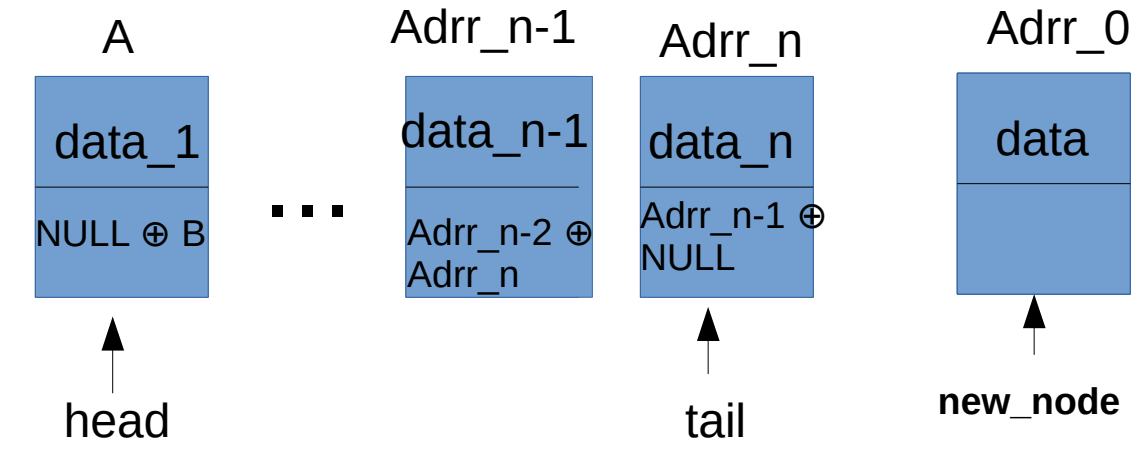
После вставки:



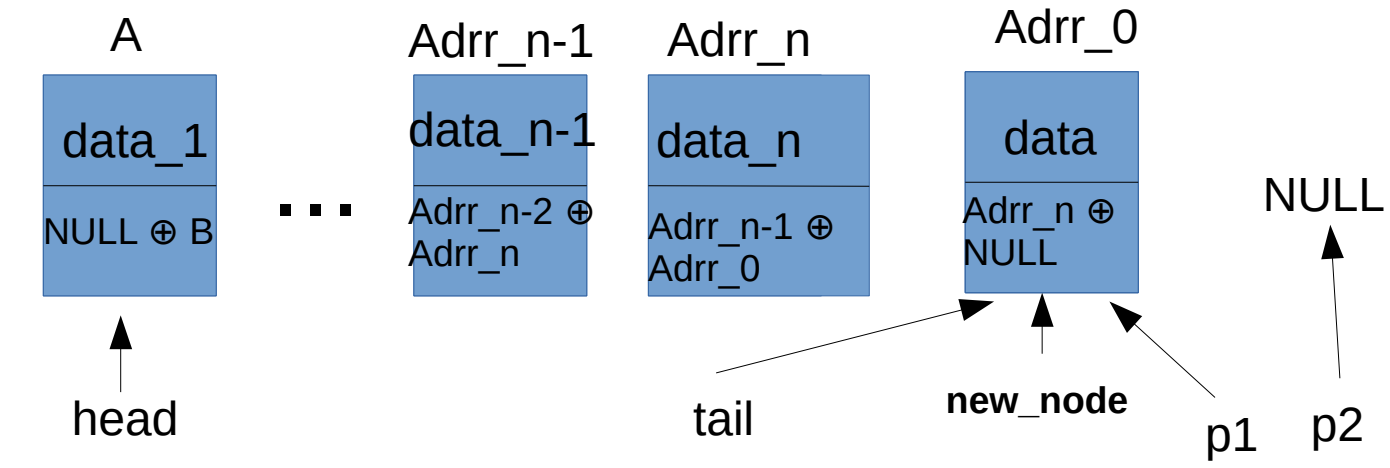
Вставка в xor-список.
Функция insert
Часть 3.

```
/* в конец списка */
else if((pos - 1) == l->size){
        new_node->ptr = xor_ptr(l->tail, NULL);
        l->tail->ptr = xor_ptr(new_node, l->tail->ptr);
        l->tail = new_node;

        l->p1 = l->tail;
        l->p2 = NULL;

        l->curr_pos = l->size;
    }
```

До вставки:



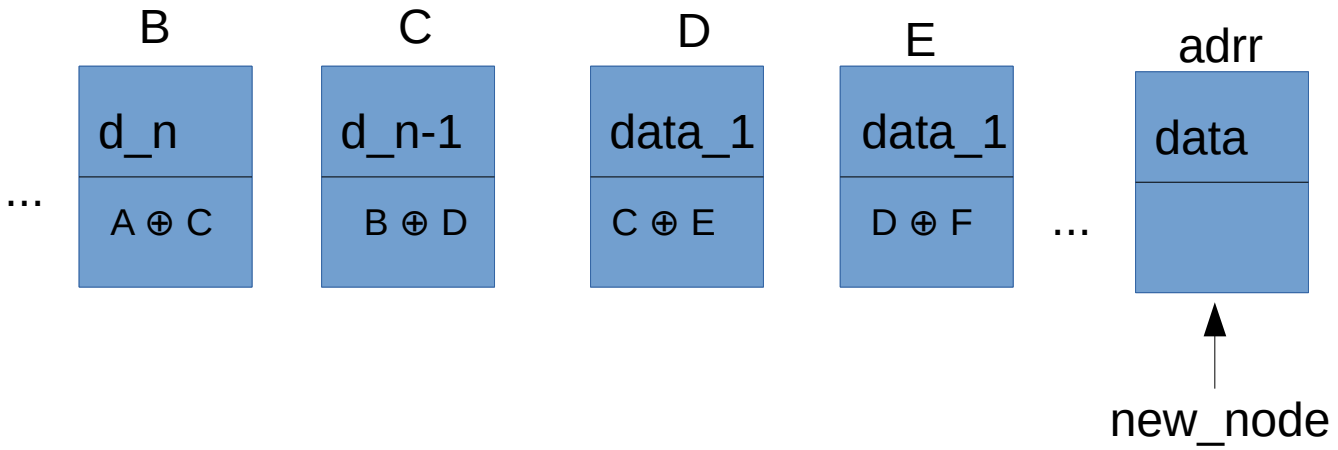
После вставки:

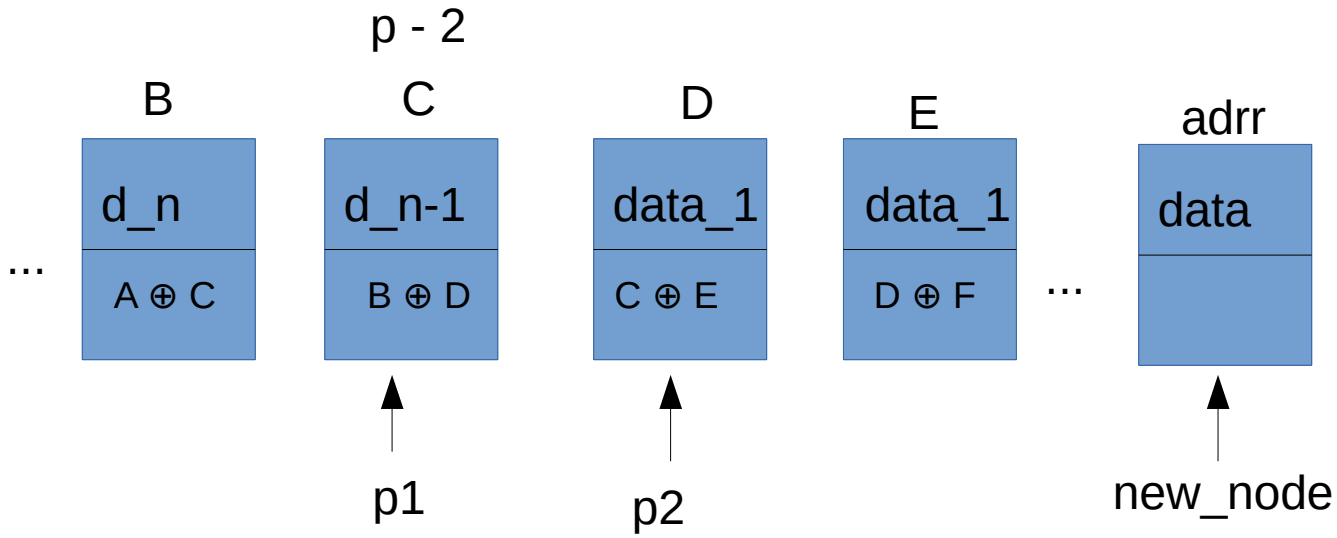


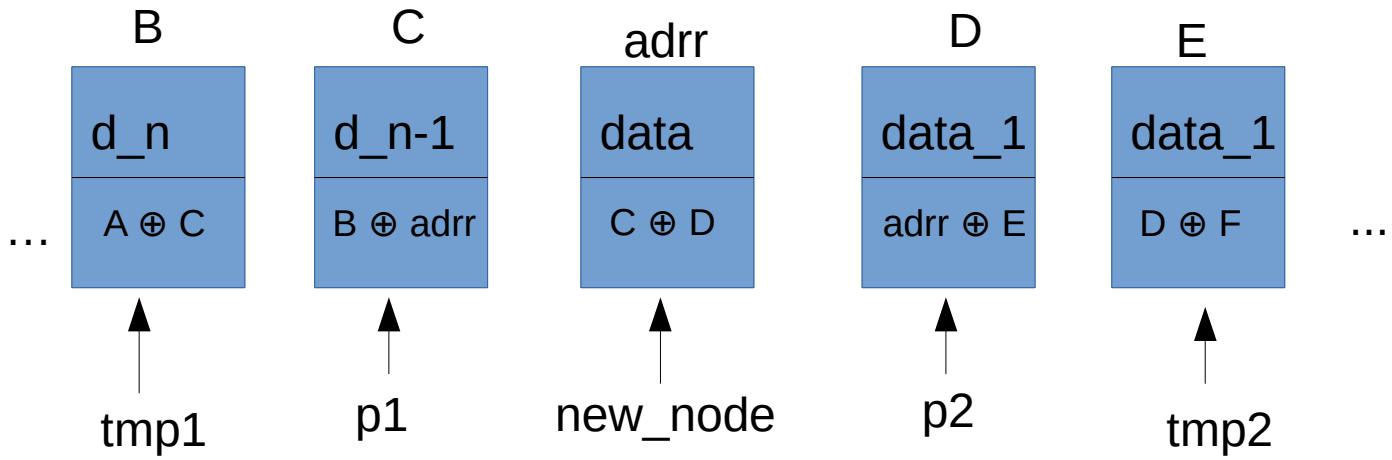
Вставка в xor-список.
Функция insert
Часть 4.

```
else{
        /* insert between p1 and p2 */
        move_to(l, pos - 1);
        new_node->ptr = xor_ptr(l->p1, l->p2);
        /* tmp1 - Node before p1 */
        struct Node *tmp1 = xor_ptr(l->p1->ptr, l->p2);
        /* tmp2 - Node after p2*/
        struct Node *tmp2 = xor_ptr(l->p2->ptr, l->p1);

        l->p1->ptr = xor_ptr(tmp1, new_node);
        l->p2->ptr = xor_ptr(tmp2, new_node);

        l->p1 = new_node;
        l->curr_pos = pos - 1;
    }

    ++(l->size);
    return 0;
}
```

Функция insert. Часть 4.

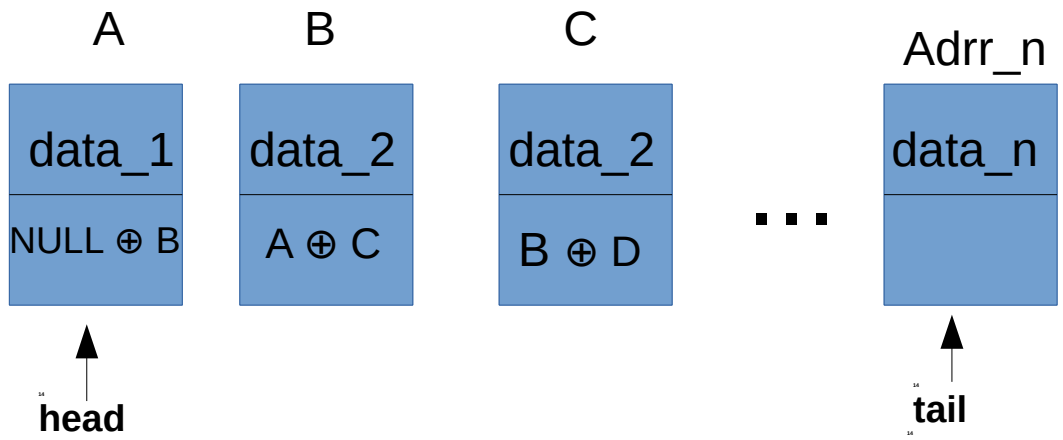


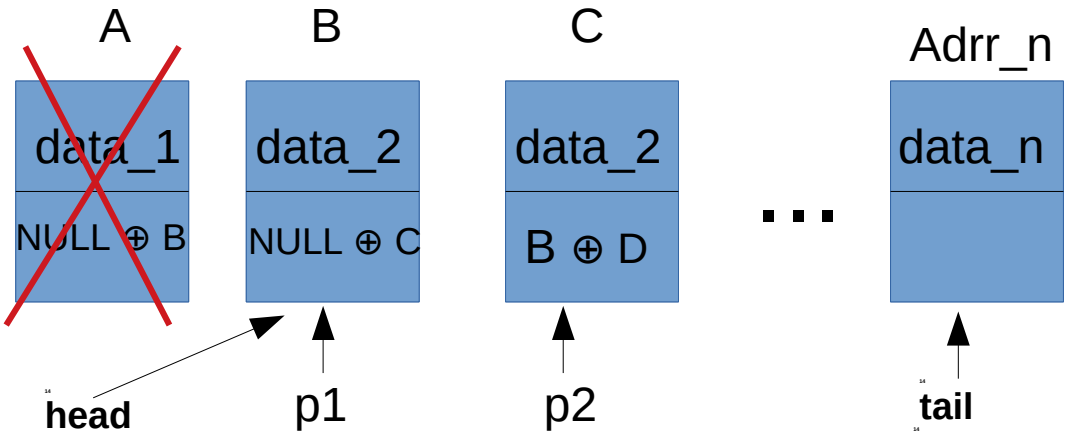




Удаление звена из xor-списока.
Функция erase
Часть 1.

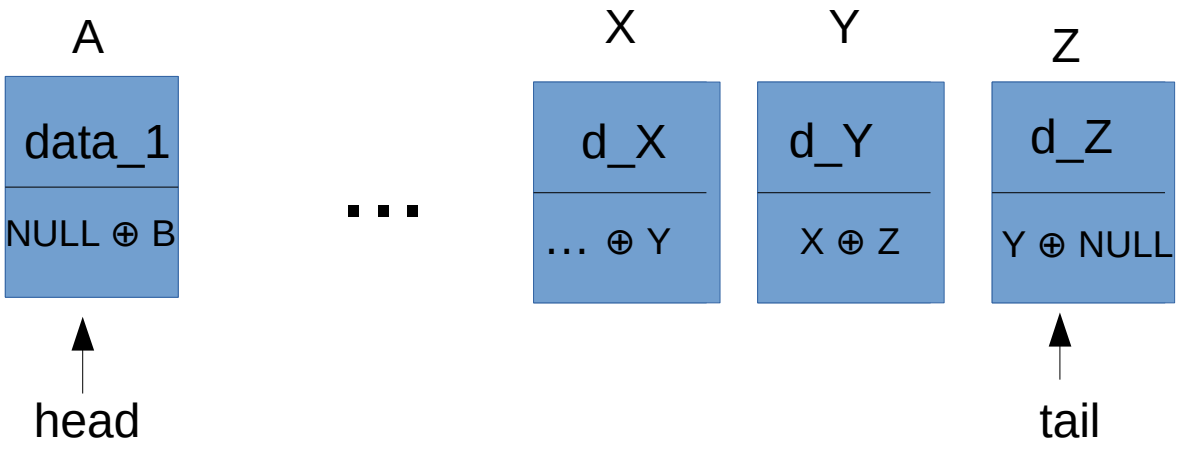
```
if(l->size == 1){
        free(l->head);
        l->p1 = l->p2 = l->head = l->tail = NULL;
        l->curr_pos = -1;
    }
/* из начала  */
else if((pos - 1) == 0){
    l->p1 = l->head->ptr;
    l->p2 = xor_ptr(l->head, l->p1->ptr);
    free(l->head);
    l->p1->ptr = xor_ptr(NULL, l->p2);
    l->head = l->p1;
    l->curr_pos = 0;
}
```

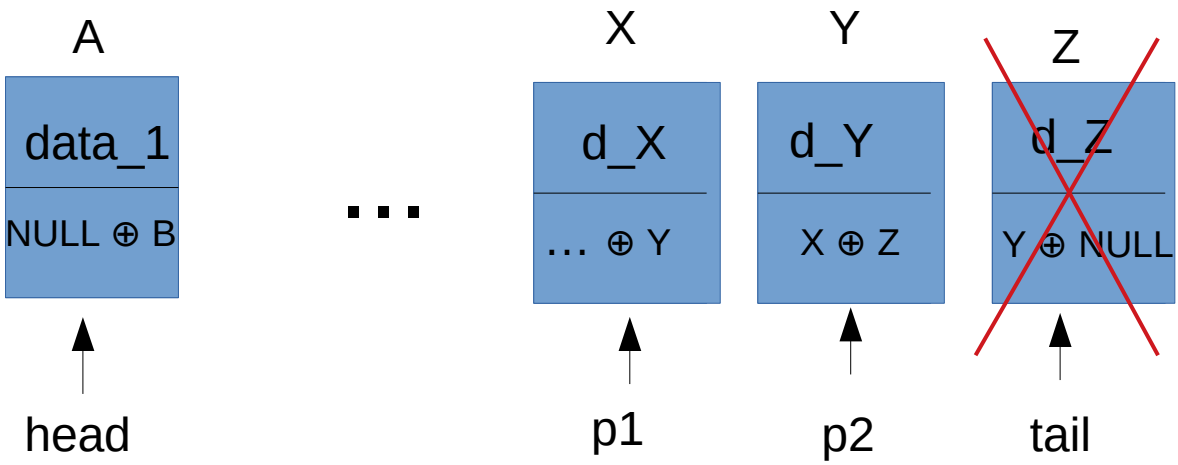




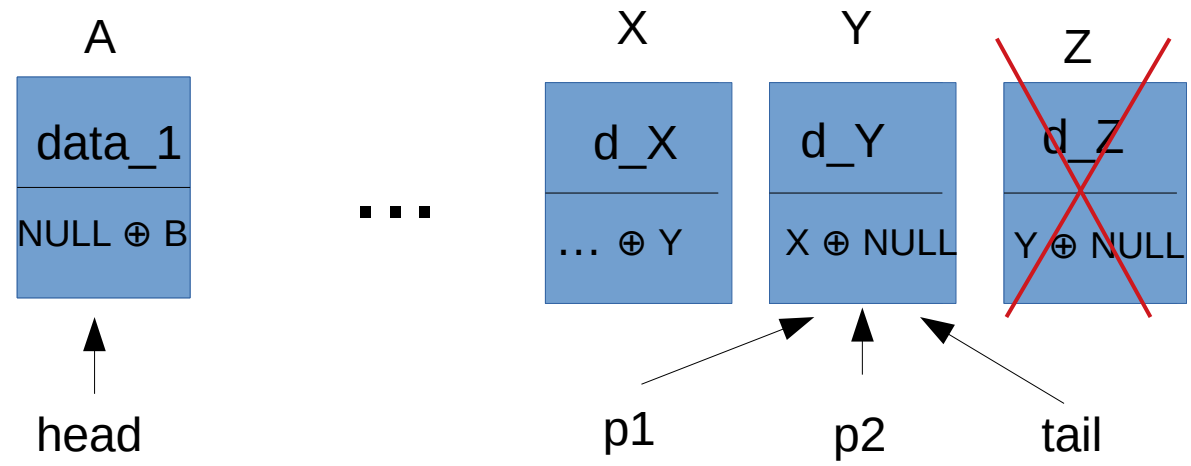
Удаление звена из xor-списока. Функция erase Часть 2.

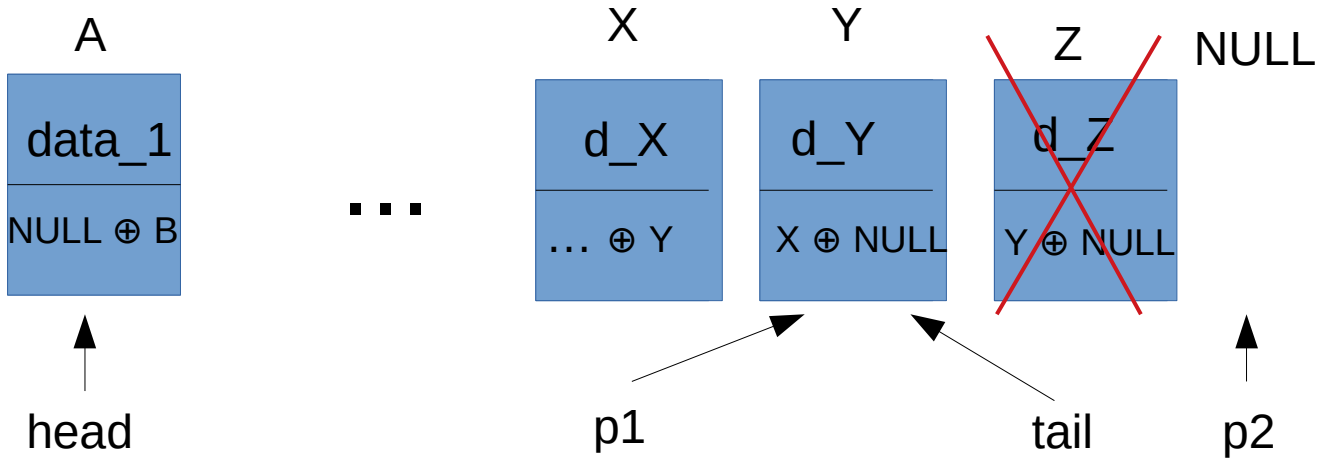
```
/* from the end */
        else if(pos == l->size){
                l->p2 = l->tail->ptr;
                l->p1 = xor_ptr(l->tail, l->p2->ptr);
                free(l->tail);
                l->p2->ptr = xor_ptr(NULL, l->p1);
                l->tail = l->p2;

                l->p1 = l->tail;
                l->p2 = NULL;
                l->curr_pos = l->size - 2;
        }
```





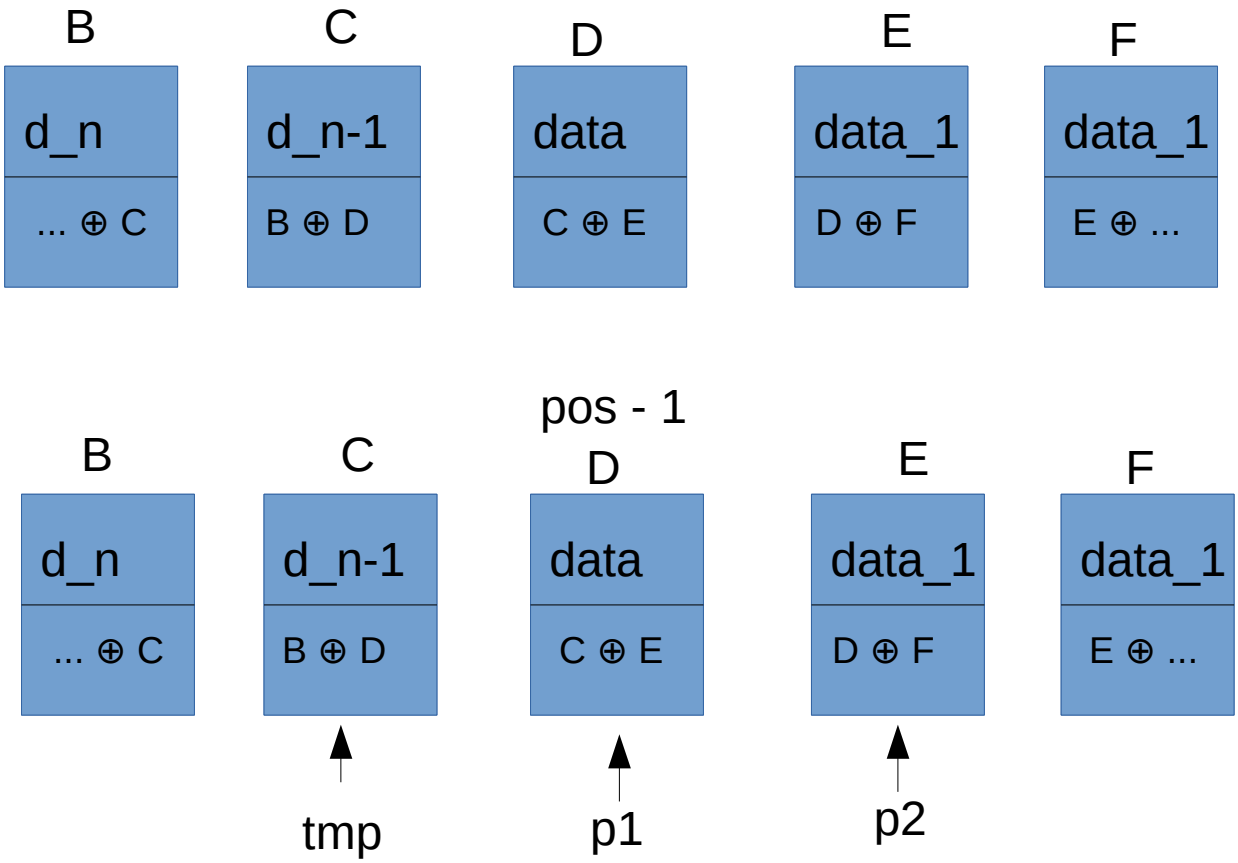
Удаление звена из xor-списока. Функция erase Часть 2.



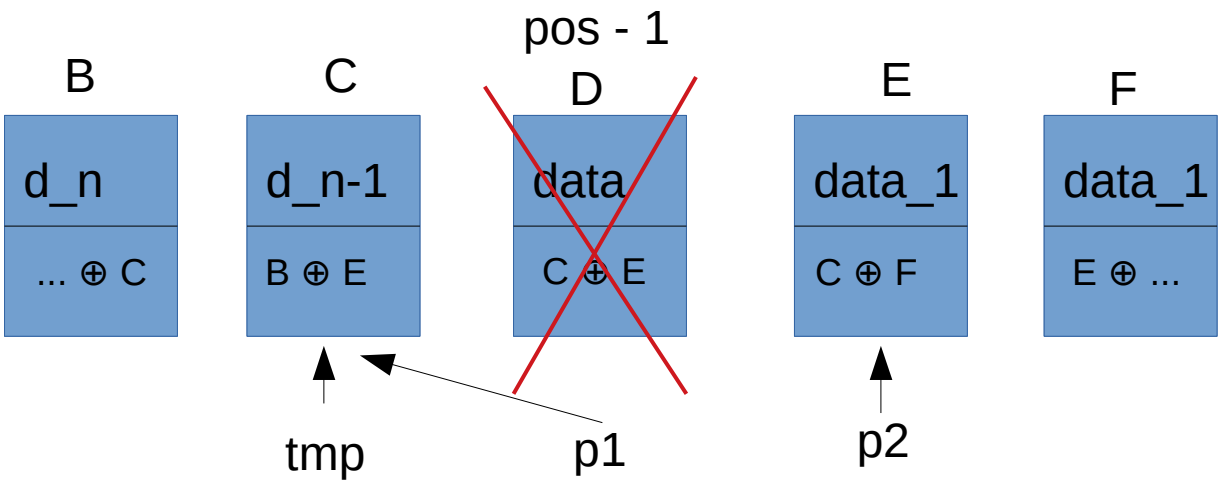


Удаление звена из xor-списока. Функция erase Часть 3.

```
else{
        move_to(l, pos);
        //Node before p1
        struct Node *tmp = xor_ptr(l->p1->ptr, l->p2);
        tmp->ptr = xor_ptr(xor_ptr(tmp->ptr, l->p1), l->p2);
        l->p2->ptr = xor_ptr(xor_ptr(l->p2->ptr, l->p1), tmp);
        free(l->p1);
        l->p1 = tmp;
    }
    --(l->size);
    return 0;
```



Удаление звена из xor-списока. Функция erase Часть 3.



Итог:

плюсы xor-списка:

- содержит функционал двусвязного списка, при этом занимает меньше памяти.

Минусы:

- Более сложная реализация, чем у двусвязного списка.
- Скорость работы уступает двусвязному списку.

структура данных xor-list может быть полезна, если нам необходим функционал двусвязного списка, но нет достаточного количества памяти для него.

Команда:

Кизин, Алкисев, Щербаков, Шелудяков, Фахретдинов.