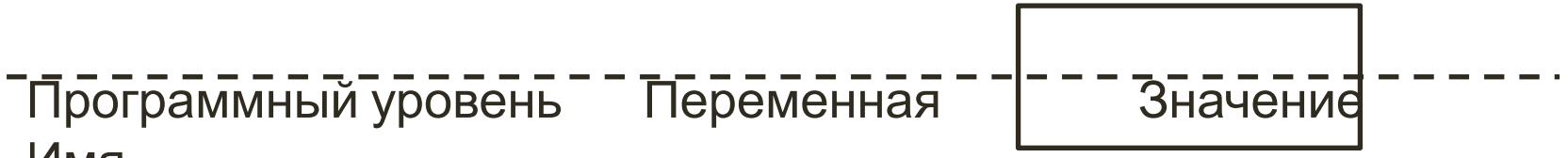
Динамические структуры данных

Адреса и указатели

До сих пор **в программах** мы использовали **переменные** – объекты программы, имеющие имя и значение.

С точки зрения машинной реализации:

имя переменной соответствует адресу участка памяти, который для нее выделен, а **значение переменной** – содержимому этого участка.



Адреса – целочисленные шестнадцатеричные беззнаковые значения, их можно обрабатывать как целые числа.

Пример: char ch;

Для удобства работы с адресами в языках программирования введены переменные типа «**указатель**».

Мы будем рассматривать типизированные указатели, которые могут хранить адреса только объектов определенного типа.

Определение *Указатель* – переменная, значением которой является адрес объекта конкретного типа.

Значение указателя может быть не равно никакому адресу. Это значение принимается за нулевой адрес. Для обозначения нулевого адреса используются специальные константы (**NULL**).

Пусть указатель **p** содержит адрес объекта **X** типа **tData**.
Графически будем изображать следующим образом:

Основные операции с указателями

Операция	Си	Псевдокод
1. Описание указателя	(tData X,Y) tData *p, *q	–
2. Получение адреса	p = &x	p := &x
3. Проверка на равенство, присваивание	p == q, p != q, p = q	p = q, p ≠ q, p := q
4.Доступ по адресу	(X=Y) *p = Y, *p =*q, X =*q	*p = Y, *p = *q, X =*q
5. Доступ к отдельной компоненте	(X.comp) p → comp (*p) . comp	p → comp
6. Отсутствие адреса	NULL	NULL

Динамически распределяемая память

Динамически распределяемая память – память, которая выделяется и освобождается по запросам программы в процессе работы программы. В качестве такой памяти обычно используется вся свободная память компьютера.

Статическая память выделяется на этапе компиляции при запуске программы и освобождается при завершении работы программы.

Две основные процедуры для работы с динамической памятью: **выделение** и **освобождение** памяти.

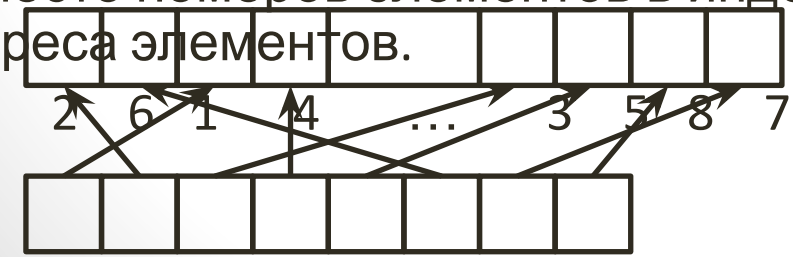
Пример. struct tData { … };
tData *p;

C++ : p = **new** tData; **delete** p;

C : p = (struct tData*) **malloc** (sizeof (struct tData)); **free** (p);

Индексация через массив указателей:

Вместо номеров элементов в индексном массиве записывают адреса элементов.



Построение индексного массива адресов

1) В массив b записываются адреса элементов массива a:

$b = (\&a_1, \&a_2, \&a_3, \ldots, \&a_n)$

2) Производится сортировка любым методом, причем

а) **при сравнении** элементы массива a адресуются через b:

$a_i < a_{i-1}$ => $a[b_i] < a[b_{i-1}]$ => $\mathbf{*b_i < *b_{i-1}}$

б) **перестановки** делаются только в массиве b:

$a_i <-> a_{i-1}$ => $b_i <-> b_{i-1}$ => $\mathbf{b_i <-> b_{i-1}}$

Достоинство метода: исходные данные могут располагаться не только в массиве, а произвольно в динамической памяти.

Линейные списки

Словарь **list** – список (простой)

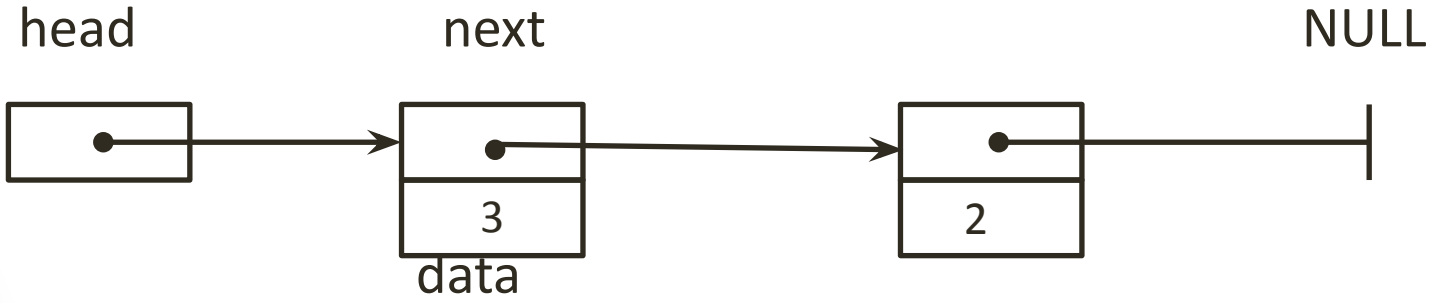
queue – очередь

next – следующий

head – голова

tail – хвост

Определение *Списком* называется последовательность однотипных элементов, связанных между собой указателями.



Пример. Пусть **tLE** - тип элемента списка:

```
struct   tLE {     tLE *next;     int   data;     } *head;
```

Поле **Next** может занимать произвольное место в структуре элементов списка. Однако, если оно является первым элементом структуры, то его адрес совпадает с адресом элемента списка, и это позволяет **оптимизировать** многие операции со списками.

Рассмотрим два вида списков: ***стек*** и ***очередь***. Их отличия в способе и порядке добавления элементов.

Стек (простой список) - новый элемент добавляется в начало последовательности, а удаляться может только первый элемент списка.

Стек реализует дисциплину обслуживания **LIFO** (**L**ast **I**nput, **F**irst **O**utput).

Очередь - новый элемент добавляется в конец последовательности, удаляется первый элемент последовательности.

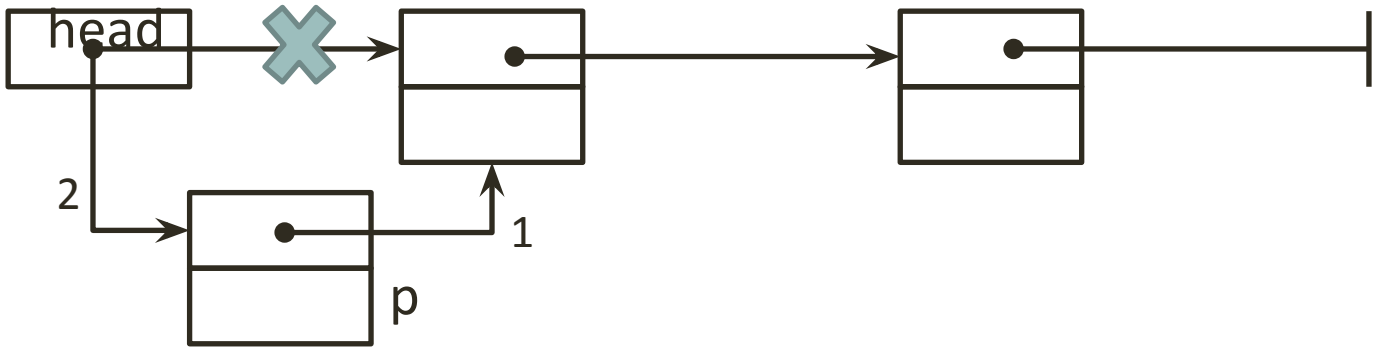
Основные операции со стеком

1) Добавление элементов в начало стека.

Предварительно должны быть сделаны операции:

<выделение памяти по адресу p>

p ->data := <данные>

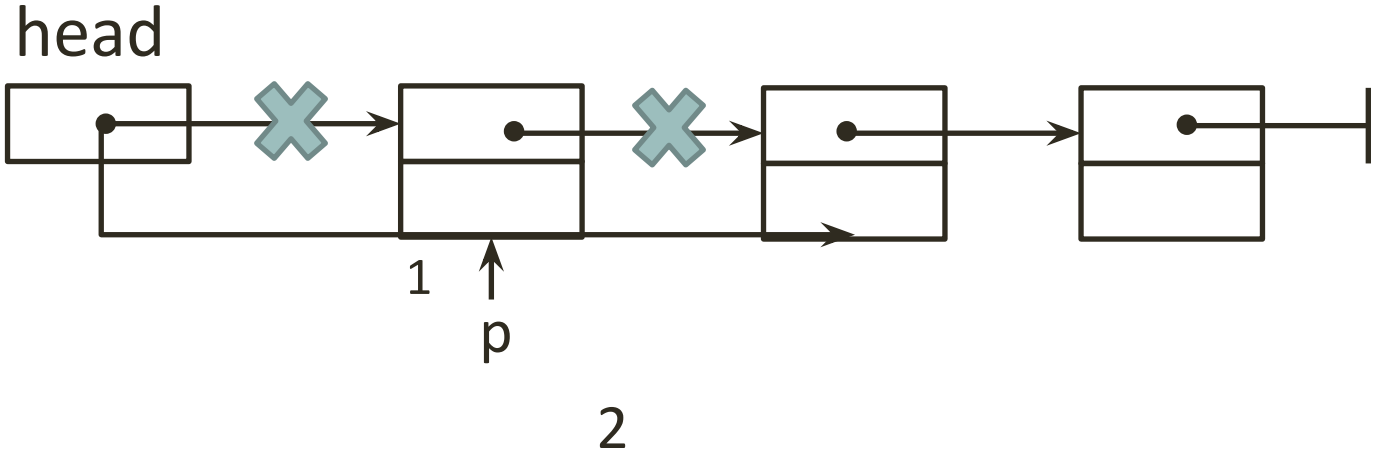


1) p->next := head
2) head := p

Основные операции со стеком

2) Исключение первого элемента из списка

Операция имеет смысл, если список не пустой (head≠NULL).



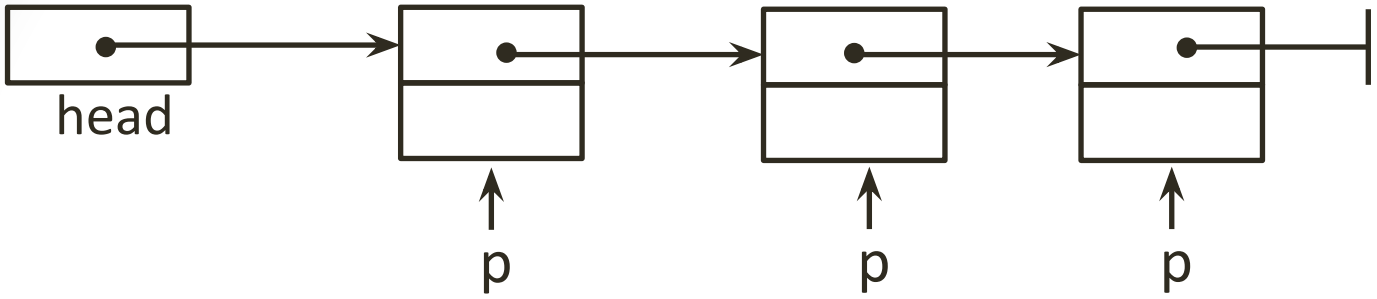
1)p := head

2) head := p ->next

delete p

Основные операции со стеком

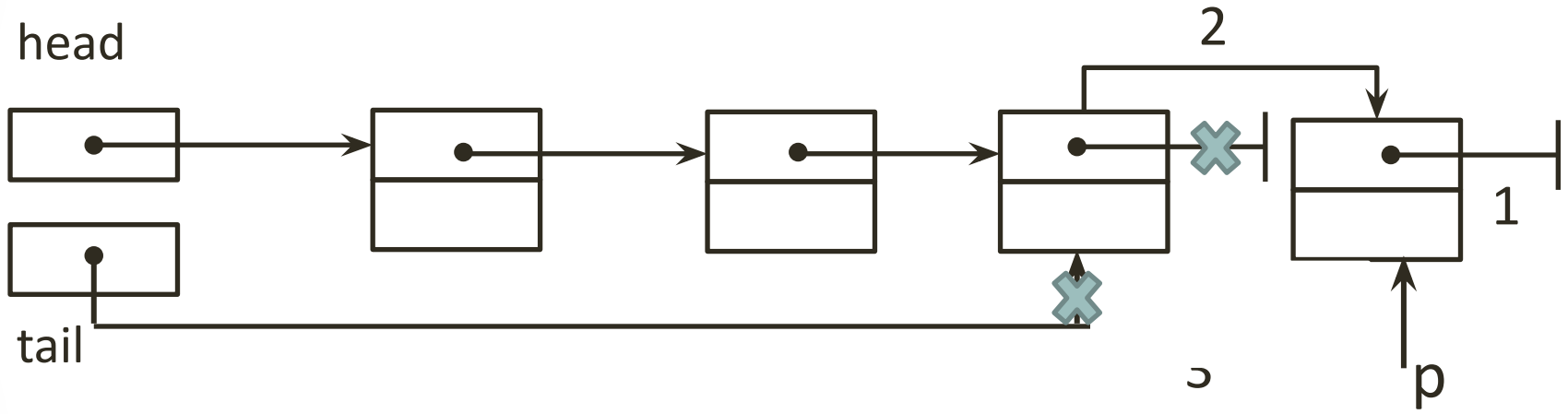
3) Просмотр списка




```
p := head
DO (p ≠ NULL)
    операция (*p)
    p := p->next
OD
```

Основные операции с очередью

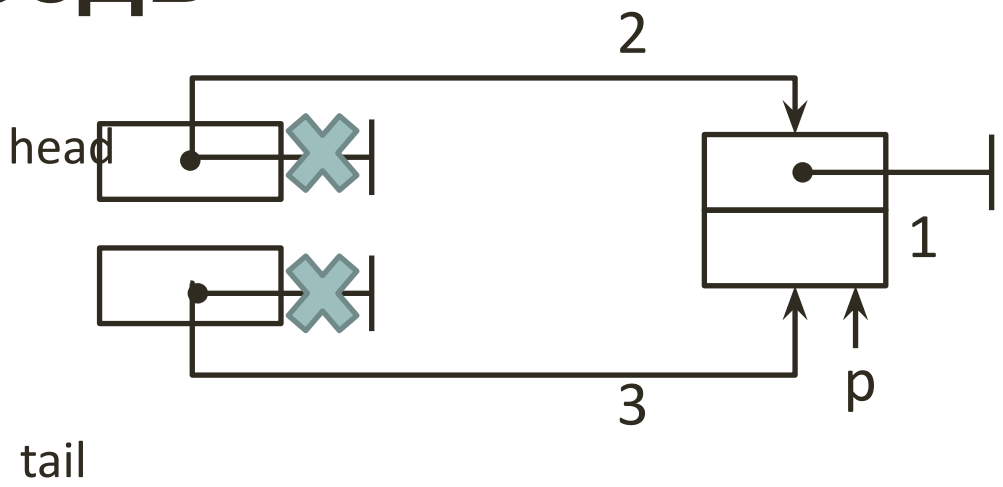
1) а) Добавление элемента в конец очереди (непустой)



1) p ->next := NULL
2) tail ->next := p
3) tail := p

Основные операции с очередью

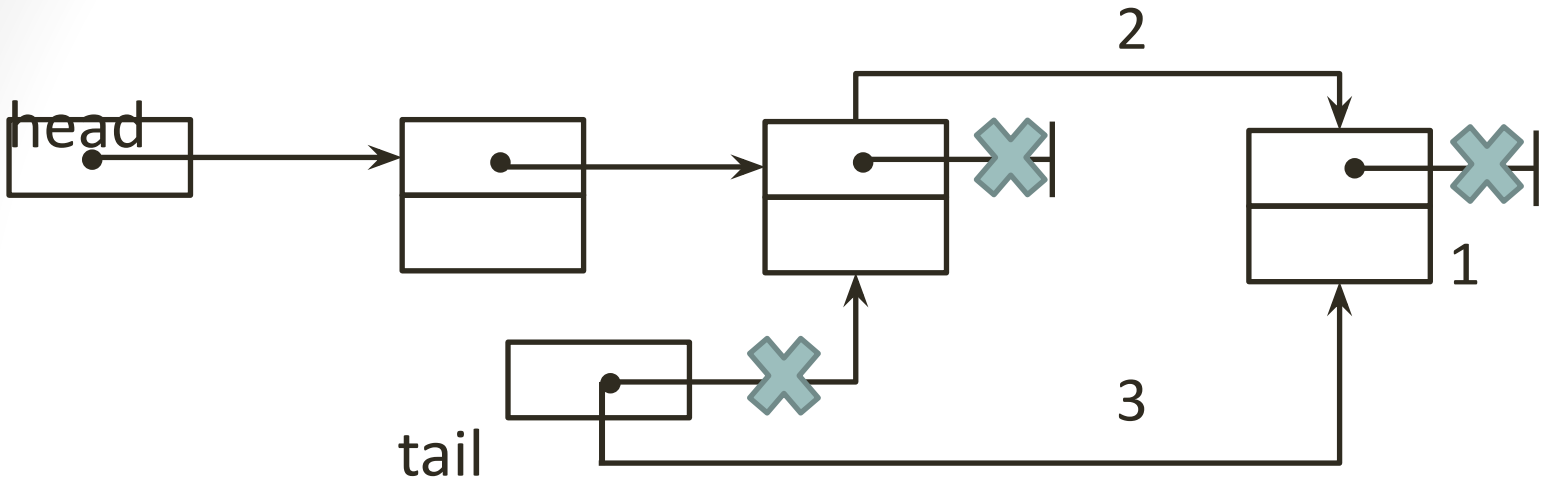
1) б) Добавление в пустую очередь



1) p ->next := NULL
2) head := p
3) tail := p

Основные операции с очередью

1) в) **Добавление элемента по адресу p в очередь**

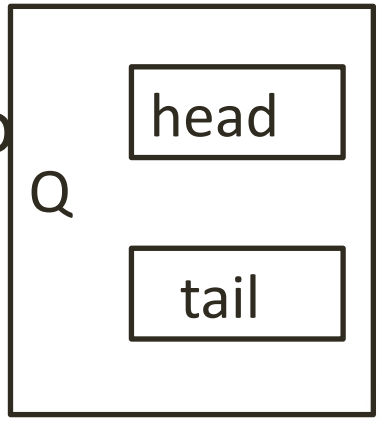


1) p→Next := NULL
2) IF (Head≠NULL) Tail→Next := p
   ELSE Head := p
   FI
3) Tail := p

2) 3) Исключение первого элемента из очереди, просмотр очереди.

Т.к. обработка любого списка производится с начала, то операции исключения первого элемента из очереди и просмотр очереди будут аналогичными стеку.

Иногда удобно рассматривать заголовок очереди как единое целое.



Это удобно, когда используется много очередей.

struct Queue { tLE *head;

tLE *tail; } Q;

Может быть даже использован массив очередей.

Задача сортировки последовательностей

Пусть дана **последовательность** $S = S_1, S_2, S_3, \ldots, S_n$ - совокупность данных с последовательным доступом к элементам.

Пример последовательности: линейный список.

Необходимо переставить элементы так, чтобы выполнялись неравенства:

$$S_1 \le S_2 \le S_3 \le \ldots \le S_n \quad \text{или} \quad S_1 \ge S_2 \ge S_3 \ge \ldots \ge S_n.$$

Последовательный доступ означает, что (**k+1**)-й элемент списка может быть получен путем просмотра предыдущих k элементов, причем просмотр возможен только в одном направлении (слева направо).

Это существенное ограничение последовательного доступа по сравнению с прямым доступом.

Методы сортировки, разработанные для массивов, не годятся для списков.

Рассмотрим операции:

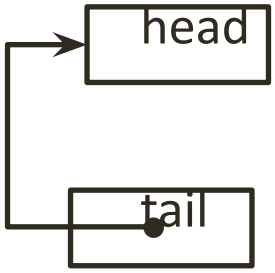
1) Постановка элемента в конец очереди:

Можно использовать алгоритм постановки в очередь, описанный ранее, но рассмотрим **оптимизированную версию**:

а) не пишем NULL в последнем элементе очереди, т.к. его адрес известен из указателя tail

б) сделаем поле next в элементе очереди первой компонентой, тогда его адрес совпадает с адресом элемента списка

в) зададим пустую очередь следующим образом:



Инициализация очереди: **tail := (tLE*) &head**

2) Добавление из стека в очередь

Q.Tail→Next:=List

Q.Tail:=List

List:=List→Next