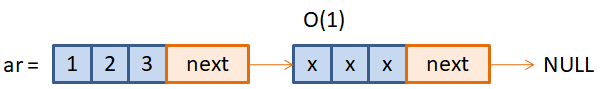
# Односвязный список. Структура и основные операции

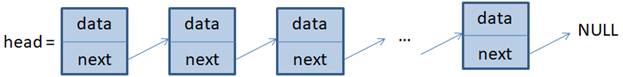
<https://proproprogs.ru/structure_data/std-odnosvyaznyy-spisok-struktura-i-osnovnye-operacii>

Мы продолжаем курс по структурам данных. На прошлых занятиях мы подробно ознакомились со статическими и динамическими массивами. Пришло время сделать следующий шаг и рассмотреть новую структуру – **односвязный список**.

Если при увеличении физического размера динамического массива мы просто выделим еще один такой же кусок где-то в памяти, то есть, данные уже не будут идти строго друг за другом, а находиться в разных областях. Очевидно, при таком подходе нам не пришлось бы копировать уже имеющиеся данные в новый массив, и не требовалось бы искать все большую и большую непрерывную область памяти. Мало того, скорость добавления нового элемента происходила бы за константное время O(1) – значительно быстрее, чем в классическом динамическом массиве. Осталось только выяснить, как нам осуществить переход от последнего элемента первого массива к первому элементу второго массива? Самое простое и очевидное что можно сделать – это хранить в последнем элементе ссылку на следующий блок данных. А тот, второй блок, должен хранить ссылку на следующий и так далее. Получаем связанную структуру из отдельных блоков памяти. Словно паровозик: во главе стоит первый блок, затем цепляется второй, потом третий и так до конца.



А теперь возведем такую структуру в некий абсолют: вообще откажемся от массивов, а вместо них определим отдельные элементы, которые будут ссылаться друг на друга, пока последний элемент не будет иметь ссылку на NULL, или None, или какое-либо другое предопределенное значение, которое означает конец последовательности.

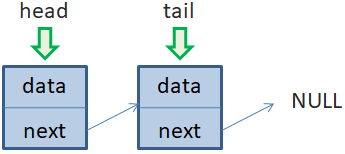


Такая структура данных, в которой элементы ссылаются друг на друга по порядку в одном направлении, начиная от первого и до последнего, получила название **односвязный список**.

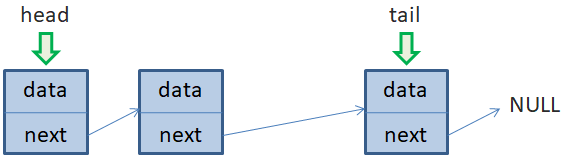
## Добавление элементов в конец и начало односвязного списка

Давайте теперь посмотрим, какие основные операции можно выполнять с этой структурой данных. Начнем с операций добавления новых элементов в конец и начало списка.

Предположим, что в списке всего два объекта. На первый элемент (объект) всегда ссылается переменная head, а на последний – переменная tail:

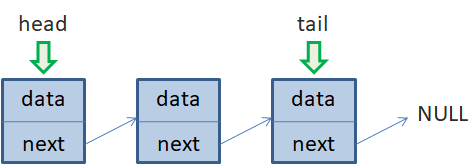


Мы хотим добавить в конец еще один элемент. Для этого, сначала нужно создать сам объект нового элемента списка (пусть на него ссылается временная переменная node). Затем, по ссылке tail выбираем последний элемент списка (который скоро станет предпоследним) и ссылку next меняем со значения NULL на значение адреса добавляемого объекта node. В свою очередь, у объекта node ссылку next устанавливаем равную NULL. И последнее, переменной tail присваиваем значение node так, чтобы tail указывала на последний элемент списка.

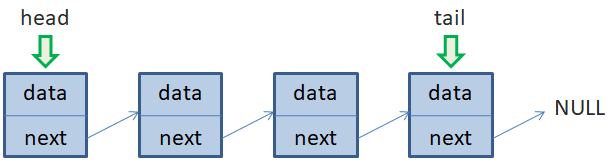


Как видите, все достаточно просто. Скорость выполнения этой операции с точки зрения О большого составляет O(1). Часто в языках программирования эту операцию реализуют методом с названием **push\_back()**.

Давайте теперь посмотрим, как выполняется добавление элемента в начало односвязного списка. Пусть у нас имеются три элемента. На первый, по-прежнему, ссылается переменная head, а на последний – переменная tail:



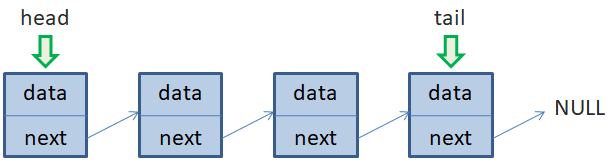
Вначале также создаем новый объект, на который ссылается временная переменная node. Затем, значение ссылки next добавляемого элемента приравниваем значению head – адресу первого элемента списка, после чего head становится вторым. Поэтому присваиваем его значению переменной node так, чтобы head ссылалась на первый элемент списка:



Скорость выполнения этой операции также O(1), то есть, происходит за константное время и не зависит от размера списка (числа его элементов). Часто в языках программирования эту операцию реализуют методом под названием **push\_front()**.

## Доступ к произвольному элементу односвязного списка

Давайте теперь посмотрим, как можно обратиться к произвольному элементу односвязного списка. Предположим, у нас имеется список из четырех объектов:



Очевидно, для получения доступа к данным первого или последнего элемента достаточно воспользоваться ссылками head и tail. Например:

head.data – доступ к данным первого элемента  
tail.data – доступ к данным последнего элемента

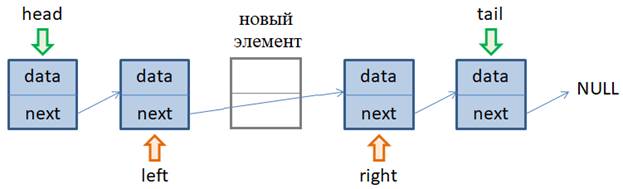
Но как нам обратиться ко 2-му или 3-му элементу этого списка? Для этого придется по цепочке двигаться от первого элемента до заданного k-го элемента, используя ссылки next. Обычно для этого создают временную переменную (ссылку на объект), допустим, с именем node. Изначально node равен head, то есть, ссылается на первый элемент. Чтобы перейти к следующему элементу нужно node присвоить адрес следующего объекта. А этот адрес хранится в переменной next текущего объекта. То есть, нам нужно выполнить команду:

node = node.next

И указатель node будет ссылаться на второй элемент односвязного списка. Далее, можем действовать по циклу, перебирая все элементы от начала и до конца, пока значение next не будет равно NULL (или любому другому терминальному значению). Это значит, для доступа к произвольному k-му элементу, мы должны последовательно перебрать все предыдущие элементы. В итоге, сложность этой операции с точки зрения О большого, составляет O(n), где n – общее число элементов в односвязном списке. Как видите, в отличие от массивов, где доступ к произвольным элементам составляет O(1), здесь все несколько сложнее. Это та цена, которую мы платим за возможность быстрого добавления элементов в односвязный список, а также возможность хранить данные в разных областях памяти компьютера.

## Вставка элемента в односвязный список

Постепенно мы с вами дошли до вставок новых элементов в связный список. Как это делается и насколько вычислительно сложно? Чтобы все было предельно просто и понятно, я возьму небольшой односвязный список из четырех элементов. Наша задача вставить пятый новый элемент после второго.



Очевидно, нам здесь понадобятся ссылки left и right на предыдущий и следующий элементы относительно добавляемого. Как их получить вы уже знаете: мы перебираем все элементы с самого начала и фиксируем два соседних, между которыми вставляется новый объект. Далее, создаем новый элемент в памяти компьютера, на который ссылается временная переменная node. Осталось только настроить связи ссылок next у объектов left и node. Из рисунка хорошо видно, что ссылка next объекта left должна вести на node, поэтому делаем присваивание:

left.next = node

А ссылка next объекта node должна вести на объект right, имеем:

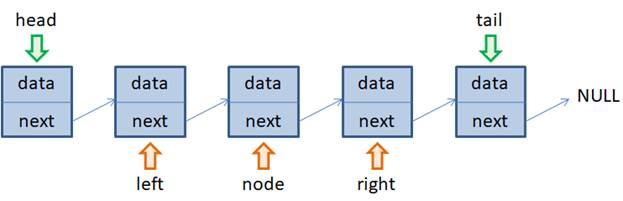
node.next = right

Все, новый объект вставлен в односвязный список. Часто в языках программирования эта операция реализуется командой **insert()**. Самое сложное в вычислительном плане здесь – это получить ссылку left объекта, после которого планируется вставлять новый элемент. В общем случае для этого требуется O(n) операций, что не мало (n – число элементов в списке). Однако если эта ссылка у нас уже есть, то сама операция вставки занимает время O(1). А это уже значительно быстрее, чем у массивов. Поэтому сложность зависит от конкретного использования односвязного списка. Если у нас в распоряжении будут уже готовые ссылки на объекты, после которых предполагаются вставки, то скорость выполнения операций будет высока. В общем же случае, когда нужно сначала найти объект, а потом вставить новый, то сложность становится линейной O(n).

## Удаление элементов из односвязного списка

В заключение этого занятия рассмотрим обратные операции по удалению элементов из односвязного списка. Начнем с удаления промежуточных объектов.

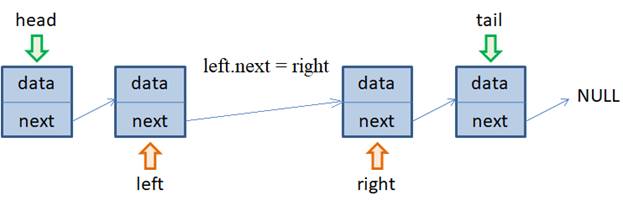
Давайте предположим, что в списке пять элементов и нам нужно удалить объект node:



Для этого достаточно у предыдущего объекта left переопределить ссылку next на объект right:

left.next = right

и освободить память, занимаемую объектом node:

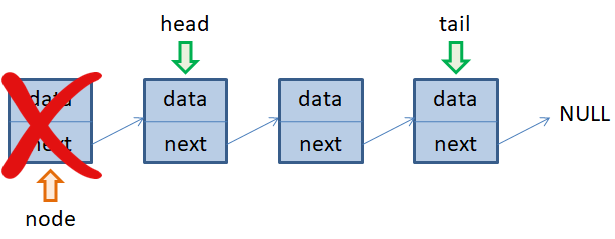


Скорость самой операции удаления элемента node из односвязного списка выполняется за константное время O(1). Но для получения ссылок left и right требуется O(n) операций. Поэтому, в общем случае, имеем объем вычислений, равный O(n), где n – число элементов в списке.

В языках программирования эта операция часто реализуется командой **erase()**.

Последнее, что мы рассмотрим на этом занятии – это удаление первого и последнего элементов односвязного списка.

Первый элемент удаляется относительно просто. Нам достаточно переместить указатель head на следующий элемент и освободить память, занимаемую элементом node:



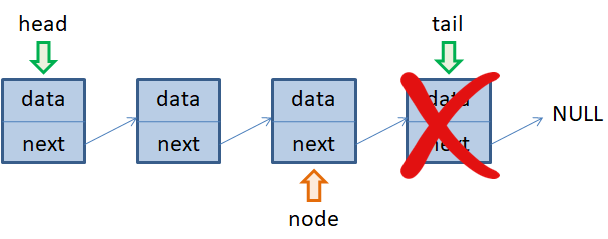
Как видите, все предельно просто. Скорость этой операции с точки зрения О большого O(1) и часто реализуется командой **pop\_front()**.

При удалении последнего элемента в односвязном списке нам необходимо сначала получить ссылку на предпоследний элемент, затем, освободить память последнего элемента tail. Изменить значение tail на node:

tail = node

и поменять значение ссылки next объекта node на NULL:

node.next = NULL



Скорость удаления самого элемента составляет O(1), но переход к предпоследнему элементу O(n). Поэтому общая вычислительная сложность этого алгоритма составляет O(n).

В языках программирования эта операция часто реализуется командой **pop\_back()**.

## Заключение

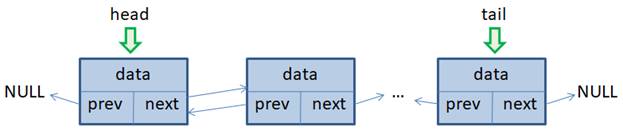
Подведем общие итоги этого занятия. Самые быстрые операции – это добавление элементов в начало и конец списка и удаление с начала списка. Остальные операции имеют сложность O(n).

|  |  |  |
| --- | --- | --- |
|  | Команда | Big O |
| Добавление в начало | push\_front() | O(1) |
| Добавление в конец | push\_back() | O(1) |
| Удаление с конца | pop\_back() | O(n) |
| Удаление с начала | pop\_front() | O(1) |
| Вставка элемента | insert() | O(n) |
| Удаление промежуточных элементов | erase() | O(n) |
| Доступ к элементу | iterator | O(n) |

На следующем занятии мы продолжим эту тему и посмотрим на пример реализации односвязного списка на языке программирования.

**ДВУСВЯЗНЫЕ СПИСКИ**

На этом занятии мы познакомимся с новой структурой данных – **двусвязными списками**. Двусвязный список – это все тот же односвязный, только каждый элемент имеет ссылки next и prev на следующий и предыдущий элементы. Во всем остальном он идентичен односвязному.



Также мы имеем здесь два указателя:

head – на первый элемент списка;

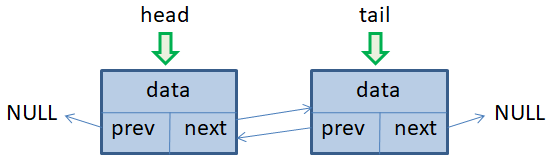
tail – на последний.

В результате, можно очень быстро обрабатывать первые и последние элементы, например, делать добавление элементов в начало списка или вконец, а также удалять их. У граничных объектов указатель prev и next принимают значения NULL. Так можно определять первый и последний элементы списка. (Помимо значения NULL можно использовать любую другую предопределенную величину, например, None если программа пишется на Python).

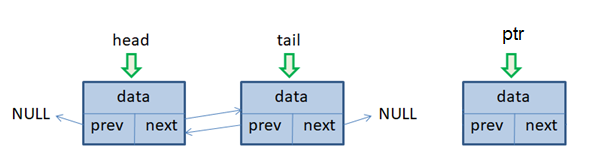
**Добавление элементов в начало и конец двусвязного списка**

Теперь рассмотрим базовые операции работы с этой структурой данных. И начнем с операции добавления элементов в начало и конец списка.

Предположим, что в списке два объекта и мы хотим добавить новый в его конец:

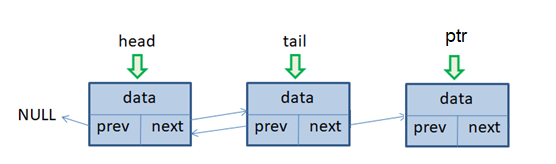


Вначале создается новый элемент в памяти компьютера, на который ведет временная ссылка ptr.



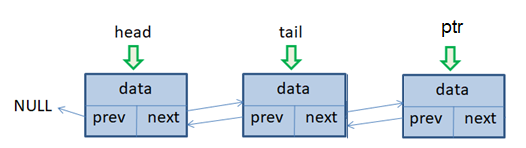
1. Затем, ссылка next последнего объекта списка должна вести на этот новый объект. Для этого достаточно присвоить ссылке адрес нового элемента:

tail.next = ptr



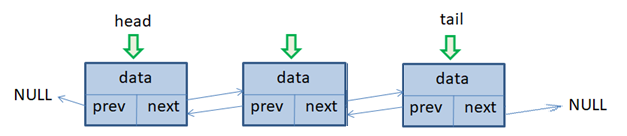
1. Далее необходимо ссылке prev нового элемента присвоить адрес последнего элемента списка:

ptr.prev = tail



И последнее, указатель tail переместить на новый последний элемент списка:

tail = ptr

tail.next = NULL

Скорость работы этой операции составляет O(1) и часто реализуется командой push\_back().

По аналогии выполняется добавление нового элемента в начало двусвязного списка. Создается новый объект, на который ссылается указатель ptr. Затем, ссылка prev первого объекта должна вести на добавляемый элемент:

head.prev = ptr

Ссылка next добавляемого элемента должна вести на объект head:

ptr.next = head

И указатель head перемещаем на объект ptr:

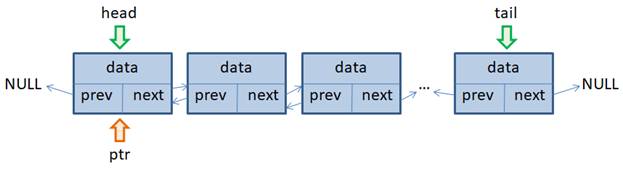
head = ptr

head.prev =NULL

Вот так, достаточно просто выполняется добавление нового элемента в начало двусвязного списка. Скорость выполнения этой операции с точки зрения О большого составляет O(1) и часто реализуется командой push\_front().

## Доступ к произвольному элементу двусвязного списка

Давайте теперь посмотрим, как осуществляется доступ к произвольному элементу в двусвязном списке. Предположим, имеется список из нескольких элементов и указатели head и tail, которые ссылаются на первый и последний элементы этого списка. Затем, создадим еще один временный указатель ptr, который пусть изначально ссылается на первый элемент:



То есть, через указатель ptr мы сейчас обращаемся к первому элементу и можем, например, прочитать его данные командой:

value = ptr.data

Чтобы перейти к следующему второму элементу необходимо переместить указатель ptr на него. Сделать это можно очень просто с помощью указателя next первого объекта:

ptr = ptr.next

Получаем доступ ко второму элементу списка. Можем записывать и считывать из него данные:

ptr.data = new\_value

По аналогии, для перехода к следующему третьему элементу, снова выполняем команду:

ptr = ptr.next

и указатель ptr ссылается на третий элемент. И так можно проходить по всем элементам списка до последнего.

С точки зрения О большого объем вычислений для перехода к произвольному k-му элементу списка составляет O(n), где n – общее число элементов в списке. И в этом двусвязный список проигрывает массивам, у которых скорость доступа к произвольному элементу O(1).

Но у двусвязных списков есть важное преимущество по сравнению с односвязными списками: мы можем перемещаться по элементам в обе стороны (и вправо и влево). Когда выполняется команда:

ptr = ptr.next

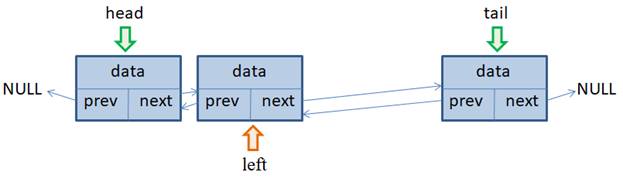
то осуществляется переход вправо. А с помощью команды:

ptr = ptr.prev

влево. В частности, это несколько упрощает некоторые операции, например, удаление промежуточных элементов.

## Вставка элемента в двусвязный список

Но сначала мы рассмотрим вставку нового элемента в произвольную позицию двусвязного списка. Предположим, в списке имеется три элемента и мы хотим вставить новый элемент после второго, на который ссылается указатель left:



Затем, создается новый объект в памяти устройства, на который ссылается временный указатель ptr. Все что нам осталось – это настроить связи между соседними элементами относительно вставляемого. Для этого создадим еще один временный указатель right, который будет ссылаться на следующий элемент после left:

right = left.next

В результате мы имеем ссылки left и right на соседние объекты относительно вставляемого.

Сформируем новые связи между элементами. Первые две связи между объектом left и ptr, очевидно, создаются командами:

left.next = ptr  
ptr.prev = left

А следующие две (между right и ptr) командами:

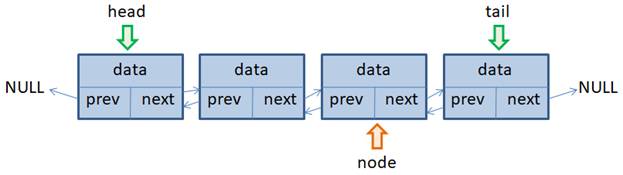
right.prev = ptr  
ptr.next = right

Все, новый элемент вставлен в двусвязный список после объекта left. Разумеется, в реальных программах нужно дополнительно проверять, чтобы объекты left и right существовали, прежде чем обращаться к указателям next и prev этих объектов.

Вычислительная сложность непосредственно вставки нового элемента, составляет O(1). Но здесь нужно учитывать, что прежде необходимо получить ссылку на объект left, после которого вставляется новый элемент. А эта операция выполняется за O(n). Поэтому, в общем случае, вставка нового элемента в двусвязный список имеет сложность O(n). И часто реализуется командой insert().

## Удаление промежуточных элементов

Следующая операция – удаление промежуточных элементов. Предположим, имеется список из четырех элементов и мы хотим удалить третий. На него ведет указатель node:



Первым шагом нам нужно получить ссылки на соседние объекты относительно удаляемого. Это можно сделать командами:

left = node.prev  
right = node.next

Затем, освободить память, занимаемую объектом node. Например, в C++ это делается командой:

delete node

В некоторых других языках, таких как Python, эту операцию берет на себя сборщик мусора, когда на объект не ведет никаких внешних ссылок.

Осталось только изменить связи у объектов left и right (связать их между собой). Делается это очень просто. Ссылка next объекта left должна вести на объект right:

left.next = right

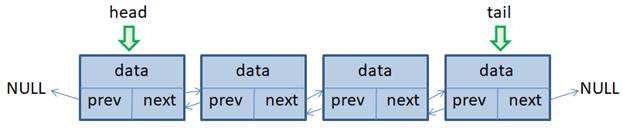
А ссылка prev объекта right на left:

right.prev = left

Все, связи настроены и удаление промежуточного элемента выполнено. Вычислительная сложность самой операции удаления составляет O(1). С учетом поиска удаляемого элемента – O(n). А сама команда часто называется erase().

## Удаление первого и последнего элементов

В заключение этого занятия отдельно рассмотрим алгоритмы удаления первого и последнего элемента двусвязного списка.



Предположим, в списке имеется четыре элемента и мы собираемся удалить первый. Тогда создадим временный указатель ptr, который будет ссылаться на следующий (второй) элемент:

ptr = head.next

И у этого второго элемента указатель prev приравняем значению NULL:

ptr.prev = NULL

Затем, освободим память, занимаемую первым элементом head. В С++ это делается с помощью оператора delete:

delete head

И переместим указатель head на второй элемент, который теперь стал первым:

head = ptr

Вот общий принцип удаления первого элемента в двусвязном списке. Вычислительная сложность этого алгоритма составляет O(1) и часто реализуется командой pop\_front().

По аналогии выполняется удаление последнего элемента. Вначале формируем временный указатель ptr на предыдущий объект списка:

ptr = tail.prev

Затем, указатель next объекта ptr приравниваем значению NULL:

ptr.next = NULL

Освобождаем память из под последнего элемента:

delete tail

И устанавливаем указатель tail на новый последний элемент:

tail = ptr

Вычислительная сложность этого алгоритма составляет O(1) и часто реализуется командой pop\_back().