Введение в структуры данных

Для вопросов по курсу:

- Почта дисциплины - alg@moevm.info

Префикс в теме письма [ALG_53XX]

Определение структуры данных

- **Структура данных** — это способ организации информации для более эффективного использования. В программировании **структурой** обычно называют набор **данных**, связанных определённым образом. Например, массив — это **структура**. **Структуры** нужны, чтобы упорядочивать, искать, анализировать и использовать **данные** с применением алгоритмов программирования.

Абстрактный тип данных

- **Абстрактный тип данных (АТД)** — это математическая модель для типов данных, где тип данных определяется поведением с точки зрения *пользователя* данных, а именно в терминах возможных значений, возможных операций над данными этого типа и поведения этих операций.

Структура данных vs. Абстрактный тип данных

- **Структура данных** определяет расположение данных в памяти
- **Абстрактный тип данных** предоставляет интерфейс для взаимодействия с данными. Реализация может быть различной.

Структура Данных	Абстрактный тип данных
Массив, связный список, бинарное дерево...	Стек, очередь, дэк...

Массив

- **Массив** – структура данных, которая хранит набор однородных данных, доступ к которым осуществляется по индексам.
- Требует выделения непрерывного участка памяти.

Индекс	0	1	2	3	4
Значение	‘H’	‘E’	‘L’	‘L’	‘O’

Массив. Вставка

- Вставка в начало массива
- Необходимо выделить новый участок памяти, вставить новый элемент, а затем скопировать элементы исходного массива.
- Сложность в О-нотации: O(n)

```
Function insert_to_begin(oldArray, element):
    size ← oldArray.size
    newArray ← array[size+1]
    newArray[0] ← element
    For i ← 0 to size-1:
        newArray[i+1] ← oldArray[i]
    Return newArray
```

Массив. Вставка

- Вставка в середину массива
- Необходимо выделить новый участок памяти, скопировать до заданного индекса элементы исходного массива, вставить новый элемент, а затем скопировать оставшиеся элементы исходного массива.
- Сложность в О-нотации: O(n)

```
Function insert_to_middle(oldArray, element, position):
    size ← oldArray.size
    newArray ← array[size+1]
    For i ← 0 to position-1:
        newArray[i] ← oldArray[i]
    newArray[position] ← element
    For i ← position+1 to size
        newArray[i] ← oldArray[i-1]
    Return newArray
```

Массив. Вставка

- Вставка в конец массива
- Для того, чтобы добавить элемент в конец массива, необходимо присвоить значение *array[size]* элементу (Обычно будет требовать перевыделение памяти)
- Сложность в О-нотации: O(1), т.к. обращение по индексу в массиве за константное время

	В начало	В середину	В конец
Вставка элемента	O(n)	O(n)	O(1)

Массив. Удаление

- Удаление из начала массива
- Выделяем участок памяти меньшего размера, проходим по массиву, начиная с первого элемента, записываем эти элементы в новый массив.
- Сложность в О-нотации: O(n)

```
1  delete_from_beginning(array):
2      size = size of array
3      newArray = create a new array with size-1
4      for i = 1 to size-1:
5          newArray[i-1] = array[i]
6      array = newArray
7      return array
```

Массив. Удаление

- Удаление из середины массива
- Выделяем участок памяти меньшего размера, копируем элементы до заданного индекса, а затем копируем элементы, начиная со следующего после заданного индекса
- Сложность в О-нотации: O(n)

```
delete_from_middle(array, index):
    size = size of array
    newArray = create a new array with size-1
    for i = 0 to index-1:
        newArray[i] = array[i]
    for i = index+1 to size-1:
        newArray[i-1] = array[i]
    array = newArray
    return arrays
```

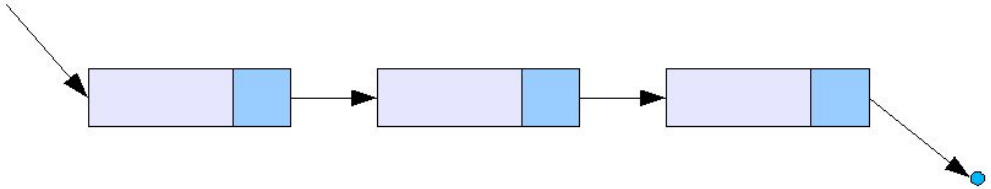
Массив. Удаление

- Удаление из конца массива
- Начинаем считать, что массив размера *size-1*. Всё.
- Сложность в О-нотации: O(1)

	Из начала	Из середины	Из конца
Удаление элемента	O(n)	O(n)	O(1)

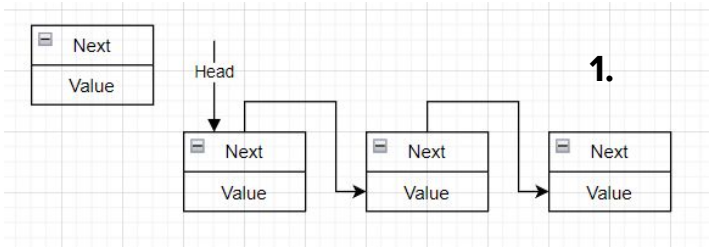
Связный список

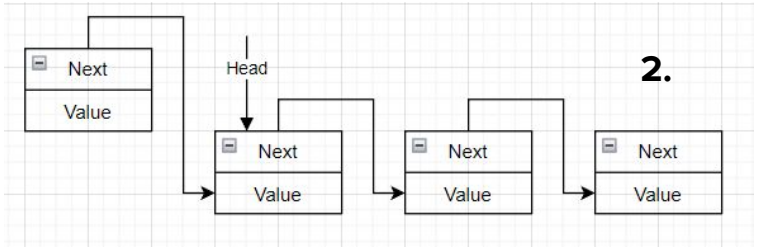
- **Связный список** — базовая динамическая структура данных в информатике, состоящая из узлов, содержащих данные и ссылки («связки») на следующий и/или предыдущий узел списка. Принципиальным преимуществом перед массивом является структурная гибкость: порядок элементов связного списка может не совпадать с порядком расположения элементов данных в памяти компьютера

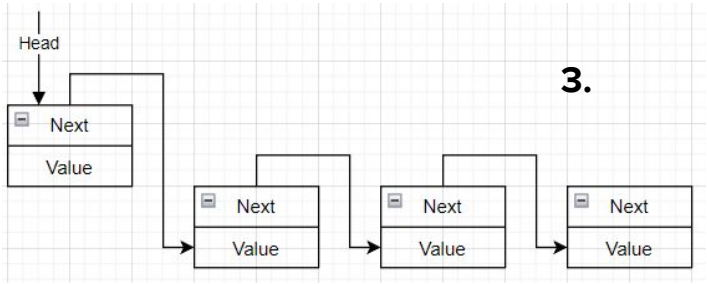


Односвязный список. Вставка

- **Вставка в начало:** Создаем новую ноду, в указатель на следующий элемент записываем текущий head, в head записываем указатель на новую ноду.

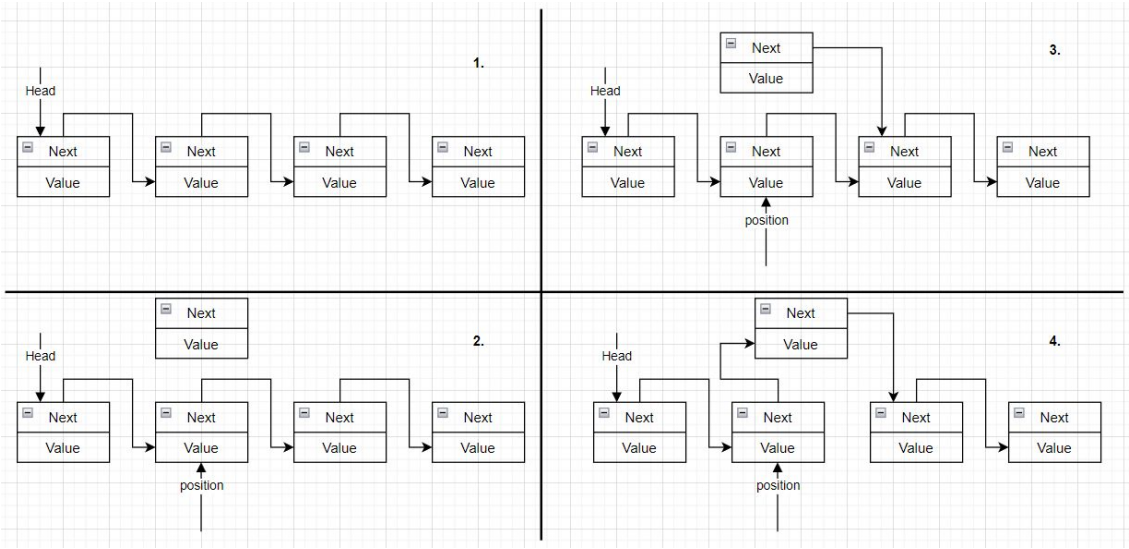






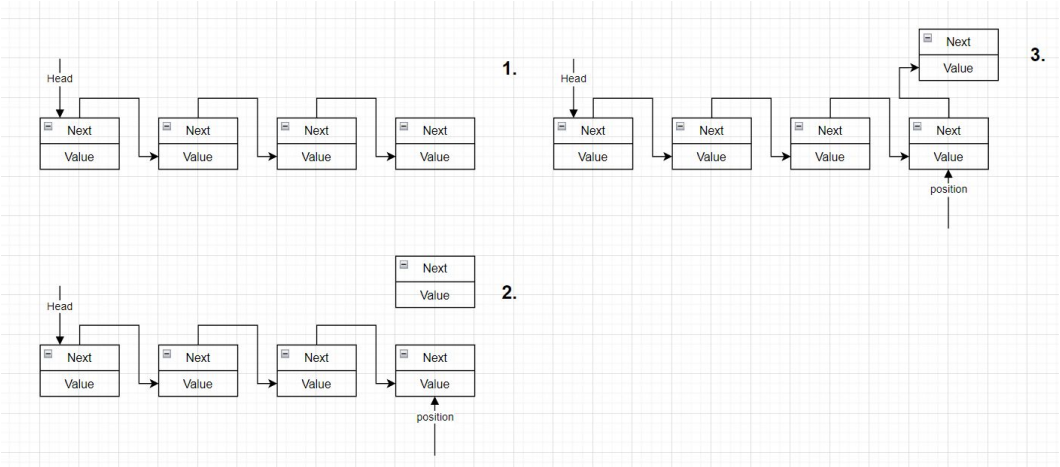
Односвязный список. Вставка

- **Вставка в середину:** Создаем новую ноду, доходим до n-1 элемента списка, указатель на следующий элемент копируем в новую ноду, затем в текущую ноду записываем указатель на новую



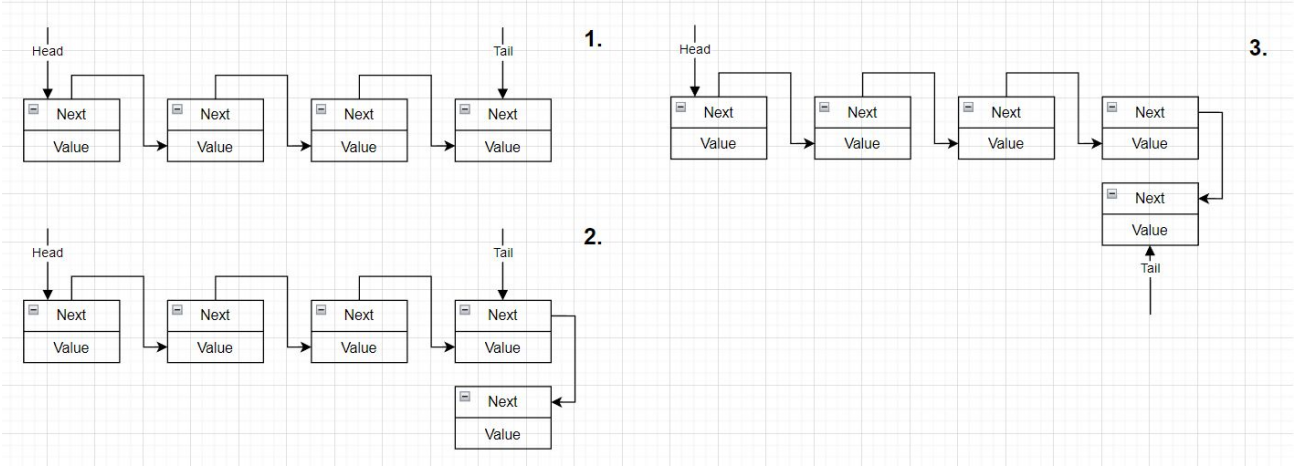
Односвязный список. Вставка

- **Вставка в конец:** Создаем новую ноду, доходим до последней, в указатель на следующий элемент записываем указатель на новую ноду. (Если храним указатель на хвост списка, то сразу записываем указатель на новую ноду в *next* хвоста, указатель на хвост двигаем на новую ноду).



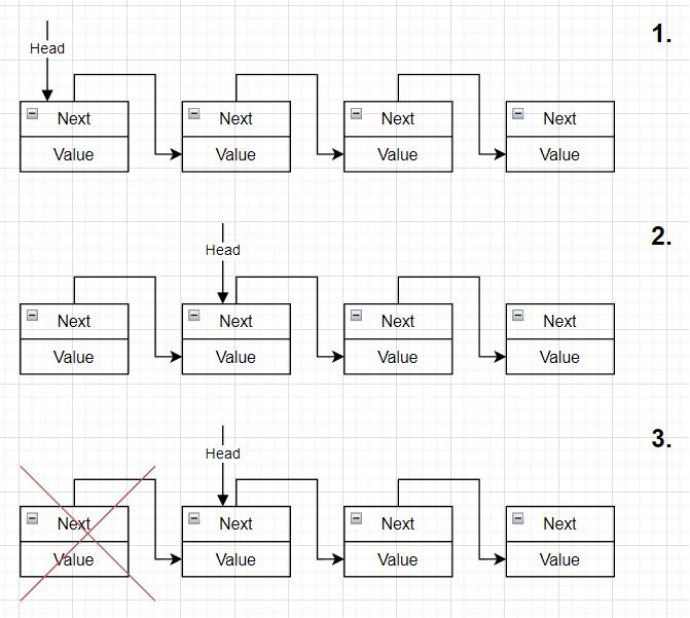
Односвязный список. Вставка

- **Вставка в конец:** Создаем новую ноду, доходим до последней, в указатель на следующий элемент записываем указатель на новую ноду. (Если храним указатель на хвост списка, то сразу записываем указатель на новую ноду в *next* хвоста, указатель на хвост двигаем на новую ноду).



Односвязный список. Удаление

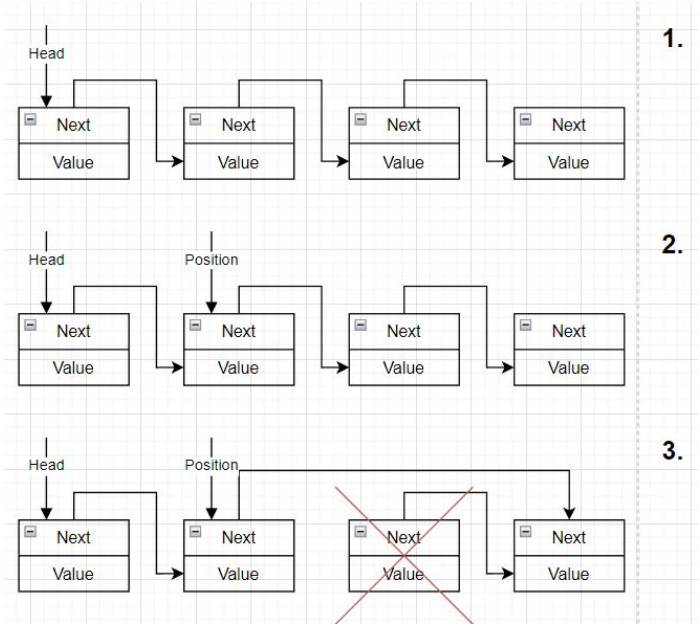
- **Удаление из начала:** Сохраняем указатель на голову списка, сдвигаем Head на следующий элемент, очищаем старый Head.




```
def remove_head(self):
    if self.head is not None:
        self.head = self.head.next
```

Односвязный список. Удаление

- **Удаление из середины:** изменим ссылку на следующий элемент на следующий за удаляемым, затем удалим нужный объект




```python
def delete(self, key):

    if self.head is None:
        print('Deletion Error: The list is empty.')
        return

    if self.head.value == key:
        self.head = self.head.next
        return

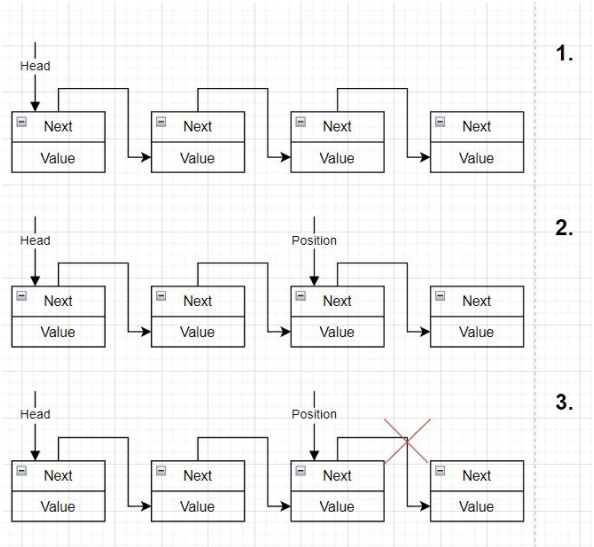
    current = self.head

    while current:
        if current.value == key:
            break
        previous = current
        current = current.next

    if current is None:
        print('Deletion Error: Key not found.')
    else:
        previous.next = current.next
```

Односвязный список. Удаление

- **Удаление из конца:** Доходим до предпоследнего элемента списка, удаляем ссылку на последний элемент. (Если память не управляется автоматически, очищаем последний узел)




```
def delete_tail(self):
    if not self.head:
        print("List is empty")
        return

    cur = self.head
    while cur.next.next:
        cur = cur.next

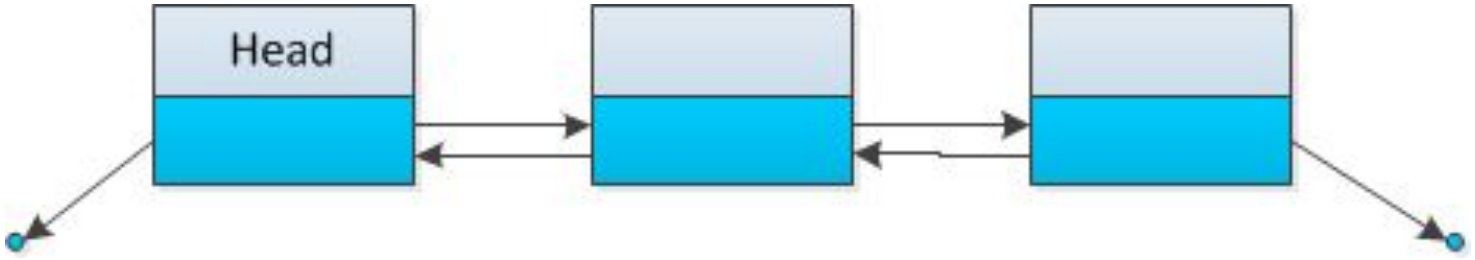
    cur.next = None
```

Односвязный список. Сложность операций

	Начало	Середина	Конец
Вставка элемента	O(1)	O(n)	O(1) или O(n)
Удаление элемента	O(1)	O(n)	O(n)

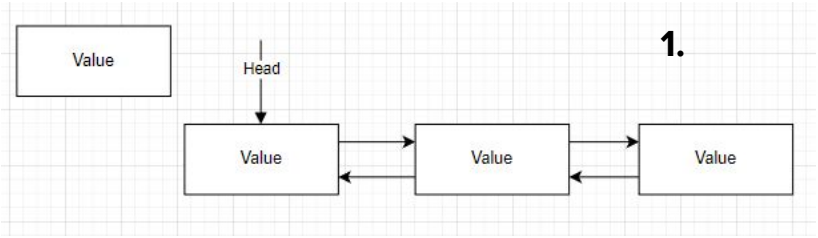
Двусвязный список

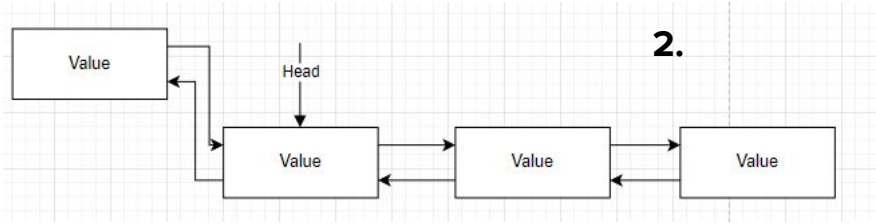
- Также хранится указатель на предыдущий элемент списка, благодаря чему становится проще удалять и переставлять элементы.

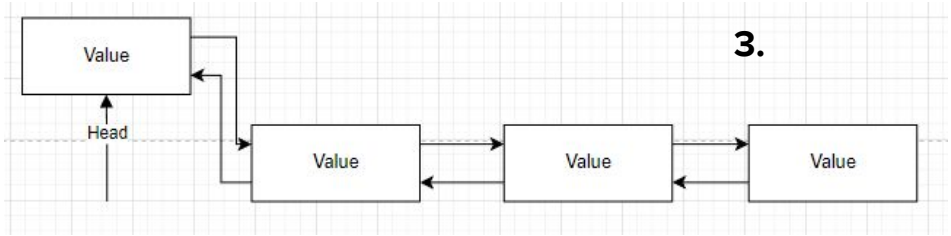


Двусвязный список. Вставка

- **Вставка в начало:** Создаем новую ноду, в указатель на следующий элемент записываем текущий head, в указатель на предыдущий элемент в head записываем новый узел, обновляем указатель на голову.

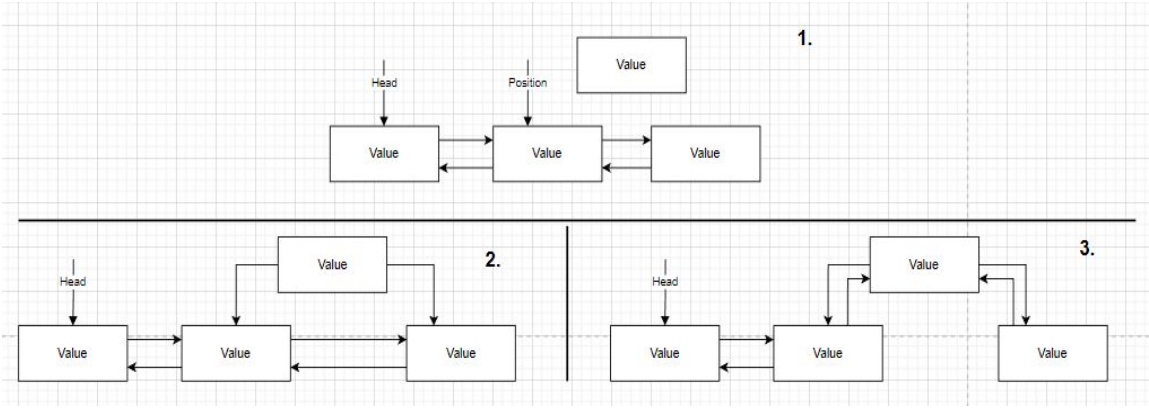






Двусвязный список. Вставка

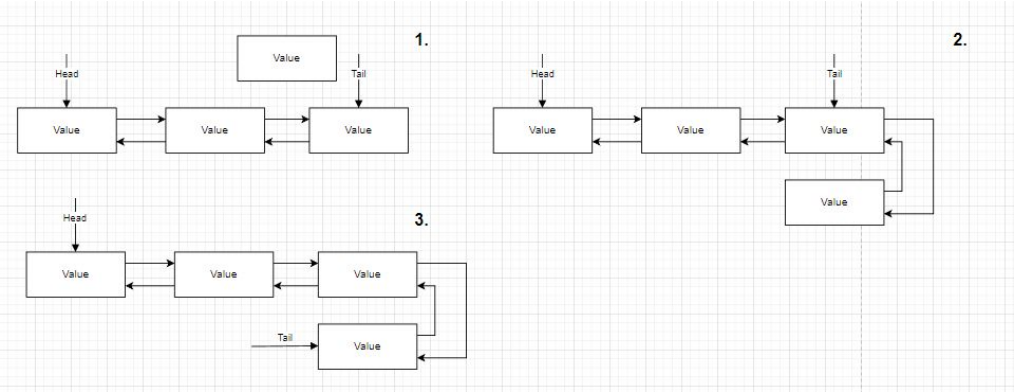
- **Вставка в середину:** Доходим до узла, после которого необходимо вставить новый узел, устанавливаем в новой ноде ссылки на текущий и на следующий элементы, обновляем указатель на следующий элемент в текущем узле и на предыдущий в следующем




```
def insert(prev_node, new_val):
    if prev_node is None:
        return
    new_node = Node(new_val)
    new_node.next = prev_node.next
    prev_node.next = new_node
    new_node.prev = prev_node
    if new_node.next is not None:
        new_node.next.prev = new_node
```

Двусвязный список. Вставка

- **Вставка в конец:** Вставка в конец в случае, когда не сохраняем указатель на конец списка, рассмотрена на предыдущем слайде. В случае, когда храним указатель на конец списка, необходимо обновить указатель на следующий элемент в tail, в указатель на предыдущий элемент нового узла записать указатель на tail, обновить tail

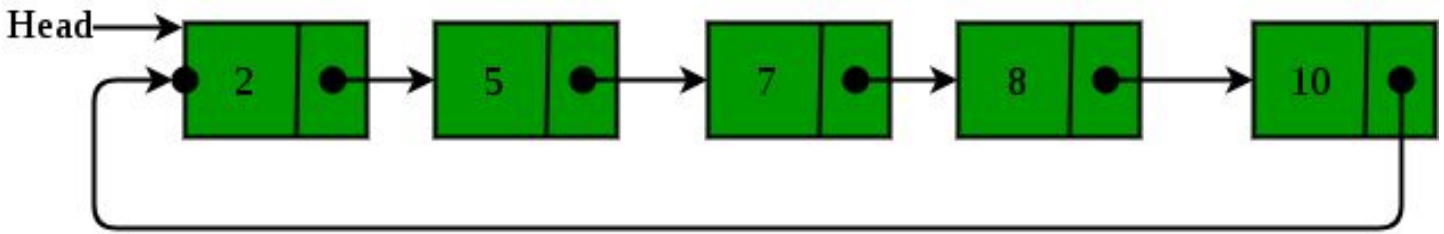


Двусвязный список. Сложность операций

	В начало	В середину	В конец
Вставка элемента	O(1)	O(1) или O(n)	O(1) или O(n)
Удаление элемента	O(1)	O(1) или O(n)	O(1) или O(n)

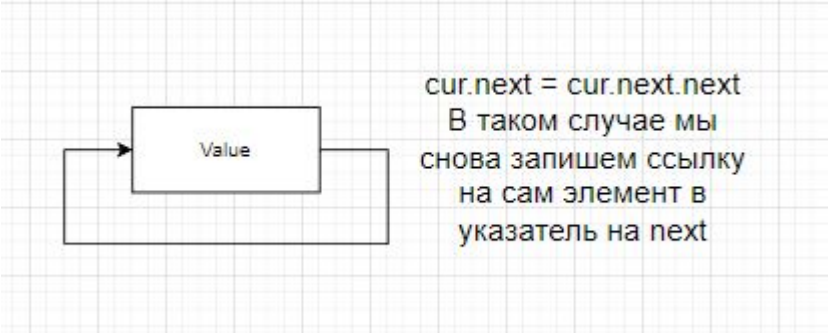
Циклический связный список

- **Круговой связный список** - это связный список, в котором все узлы соединены, образуя цикл. В круговом связном списке первый узел и последний узел соединены друг с другом, что образует цикл. В конце нет NULL.



Циклический связный список.

- Операции вставки/удаления работают аналогично линейным спискам, при этом пропадает необходимость проверять обращение к NULL. Но необходимо проверять, что список не состоит из 1 элемента (при удалении)



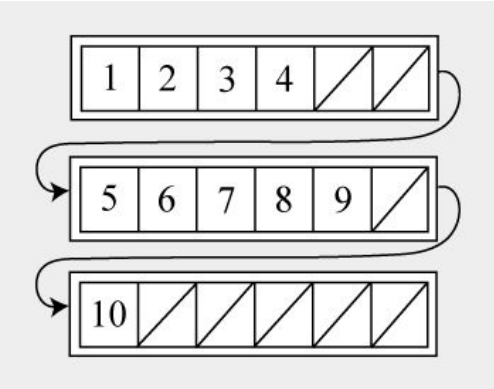
Циклический связный список.

- Циклический список может быть естественным вариантом для представления структур, которые являются циклическими по своей сути, например, углов многоугольника, пула буферов, которые используются и освобождаются в порядке FIFO ("первый вошел, первый вышел").
- При круговом списке указатель на последний узел обеспечивает легкий доступ также и к первому узлу, перейдя по одной ссылке.

- Напротив, использование null для обозначения пустого линейного списка является более естественным и часто создает меньше особых случаев.

Развернутый связный список

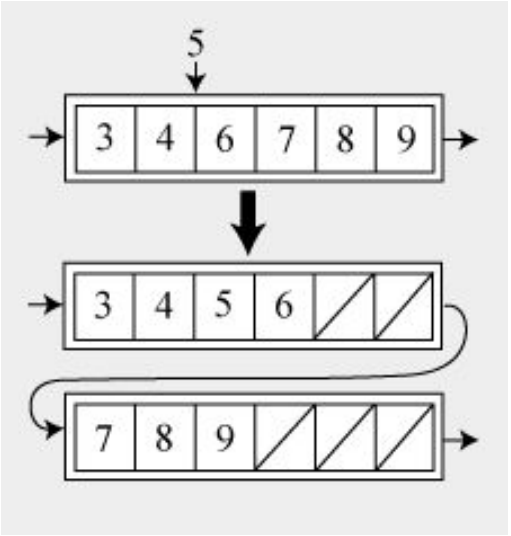
- Попросту говоря, **развернутый связный список** — связный(двусвязный) список массивов длины n. То есть каждая нода содержит в себе не одно значение, а массив значений. При этом массив является достаточно маленьким, чтобы операции вставки и удаления были очень быстрыми и фрагментация памяти меньше влияла на возможность создания ноды



Развернутый связный список. Вставка

Рассмотрим добавление в конец узла развернутого списка

- В случае наличия свободных ячеек в узле, новый элемент просто добавляется в конец массива узла.
- Если узел заполнен, то мы создаём новую ноду за текущей и переносим в нее правую половину массива, создавая тем самым место для вставки новых значений. Новый элемент в таком случае будет вставлен в конец новой ноды. То же можно применить к вставке в середину массива.



Пример вставки в середину

Развернутый связный список. Поиск

- Как и в связном списке поиск начинается с головы или хвоста, откуда ближе, отличие в том, что итерироваться по массивам нет нужды — необходимо просто суммировать значения полей, содержащих количество элементов в ноде. Таким образом поиск нужной ноды происходит существенно быстрее, чем в связном списке, хотя асимптотика тут по-прежнему O(N).

Развернутый связный список. Алгоритм вставки

- Проверяем, заполнен ли узел
- Если заполнен, создаем новый узел
- Переносим половину элементов узла в созданный узел
- Добавляем новый элемент в конец узла
- Иначе, добавляем элемент в конец узла

Развернутый связный список. Алгоритм получения элемента по индексу

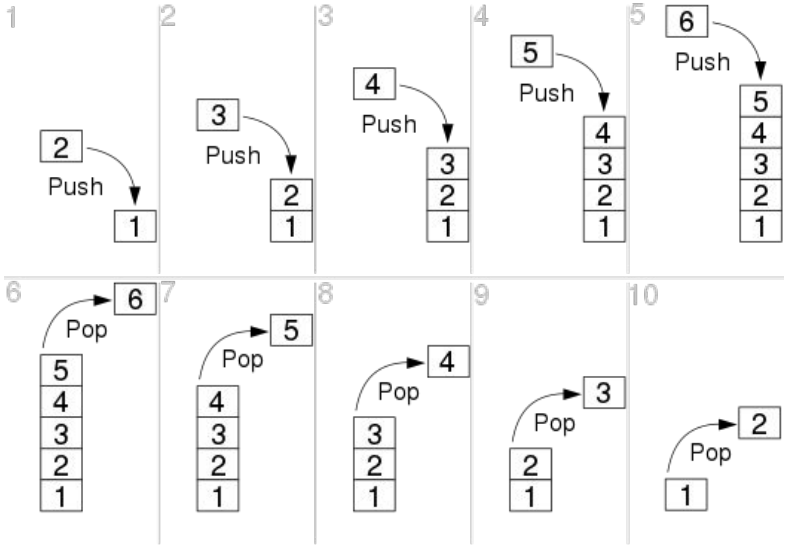
- Проверяем, превышает ли индекс общую длину списка
- Если превышает - возвращаем ошибку
- Иначе, заводим счетчик элементов и флаг, сигнализирующий, найден ли элемент
- В цикле пока элемент не найден:
- Если счетчик + длина текущей ноды > индекса и счетчик <= индекса - возвращаем элемент из текущей ноды под индексом index-count
- Иначе добавляем к счетчику длину текущей ноды и переходим к следующему узлу

Развернутый связный список. Алгоритм удаления элемента

- С помощью алгоритма поиска находим нужный элемент
- Удаляем его из текущей ноды
- Производим балансировку списка
- Удаляем пустые узлы

Стек

- **Стек** – абстрактный тип данных, представляющий собой список элементов, организованных по принципу *LIFO* (англ. *last in — first out*, «последним пришёл — первым вышел»).

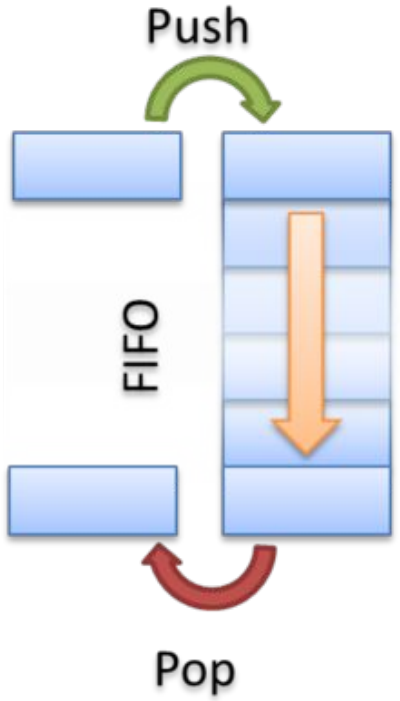


Стек

- Возможны реализации на различных структурах данных (например, стандартными являются реализации на основе массива и линейного списка)
- Стандартные операции со стеком:
  - Вставка элемента
  - Удаление элемента
  - Чтение элемента из вершины стека
- Стандартные операции со стеком выполняются за **O(1)**, т.к. мы всегда храним указатель на вершину стека

Очередь

- **Очередь** – это структура данных, добавление и удаление элементов в которой происходит путём операций push и pop соответственно. Притом первым из очереди удаляется элемент, который был помещен туда первым, то есть в очереди реализуется принцип «первым вошел — первым вышел» (англ. *first-in, first-out — FIFO*).
- У очереди имеется **голова** (англ. *head*) и **хвост** (англ. *tail*). Когда элемент ставится в очередь, он занимает место в её хвосте. Из очереди всегда выводится элемент, который находится в ее голове.

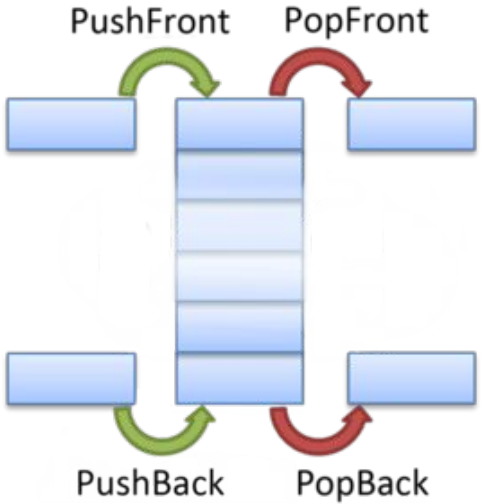


Очередь

- Возможны реализации на различных структурах данных (например, стандартными являются реализации на основе массива и линейного списка)
- Стандартные операции с очередью:
  - Вставка элемента
  - Удаление элемента
  - Чтение элемента из головы очереди
- Стандартные операции с очередью выполняются за O(1), т.к. мы всегда храним указатели на начало и конец. Это дает возможность выполнять операции за константное время

Дек

- **Двусвязная очередь** (*дек* от англ. *deque* — *double ended queue*; двусторонняя очередь) — абстрактный тип данных, в котором элементы можно добавлять и удалять как в начало, так и в конец. Может быть реализована при помощи двусвязного списка.



Дек

- Возможны различные реализации деки (на двусвязном списке, на двух стеках, циклическая дека).
- Стандартные операции:
  - Вставка в начало
  - Вставка в конец
  - Удаление из начала
  - Удаление из конца
- Амортизированная сложность операций - O(1)

Кольцевой буфер

Структура данных фиксированного размера, которая использует один массив* и работает по принципу "кольца". Когда указатели доходят до конца массива, они "перематываются" в начало.

“Кольцевая” логика:

- Вместо сдвига элементов при удалении/добавлении используются индексы:
  - head — индекс первого элемента (начала деки).
  - tail — индекс последнего элемента (конца деки).
  - N — размер (вместимость) массива
- Переход к следующей ячейке:
  - next=(index+1)%N
- Переход к предыдущей:
  - prev=(index−1+N)%N

Главный плюс: операции добавления и удаления с обоих концов работают за O(1)

- *почему?*

Кольцевой буфер. Состояния

Проблема кольцевого буфера — отличить состояние "пусто" от состояния "заполнено".

- Стандартное решение: хранение счётчика заполненности (текущего размера) size:
- Инициализация:
  - head = 0, tail = 0, size = 0.

Состояние	Условие	Доступные операции
Пусто (Empty)	size() == 0	Нельзя делать pop и front/back (исключение).
Заполнено (Full)	size() == N	push* зависит от флага push_force.

При таком подходе индексы head/tail могут указывать на одну ячейку и в пустом буфере, и в заполненном — всегда сверка с size.

Кольцевой буфер. Операция добавления (push)

Для операций push_* (front/back) используется параметр *push_force*

- Если push_force == False:
  - Проверка full(): если True — исключение, иначе добавление элемента и увеличение size.
- Если push_force == True (Перезапись):
  - push_back(x): запись поверх head .
    - Запись x по индексу head.
    - Сдвиг head вперёд (head = next(head)).
    - Если буфер был полон, сдвиг tail вперёд (т.к. “конец” сместился), иначе просто увеличение size.
  - push_front(x): Запись поверх tail (конца).
    - Запись x по индексу tail.
    - Сдвиг tail назад (tail = prev(tail)).
    - Если буфер был полон — сдвиг head назад (т.к. “начало” сметилось); иначе увеличиваем size.

Т.е. при переполнении старые элементы вытесняются, но указатели корректно замыкают круг.

Кольцевой буфер. Операции удаления (pop) и чтения добавления (front/back)

Для операций pop_* (front/back) дополнительных параметров нет.

- pop_front()
  - Если empty() → исключение.
  - Значение по индексу head в переменную result.
  - Сдвиг head вперёд: head = (head + 1) % N.
  - Уменьшение size на 1.
  - return result
- pop_back()
  - Если empty() → исключение.
  - Значение по индексу tail в result.
  - Сдвиг tail назад: tail = (tail - 1 + N) % N.
  - Уменьшение size на 1.
  - return result.
- front() и back():
  - Аналогично pop, но без изменения индексов и размера, только return array[head] или array[tail].

Кольцевой буфер. Операции удаления (pop) и чтения добавления (front/back)

Для операций pop_* (front/back) дополнительных параметров нет.

- pop_front()
  - Если empty() → исключение.
  - Значение по индексу head в переменную result.
  - Сдвиг head вперёд: head = (head + 1) % N.
  - Уменьшение size на 1.
  - return result
- pop_back()
  - Если empty() → исключение.
  - Значение по индексу tail в result.
  - Сдвиг tail назад: tail = (tail - 1 + N) % N.
  - Уменьшение size на 1.
  - return result.
- front() и back():
  - Аналогично pop, но без изменения индексов и размера, только return array[head] или array[tail].

Вопросы по курсу можно задавать:

Почта дисциплины - alg@moevm.info
Иванов Дмитрий Владимирович - dmitry.ivanov@moevm.info