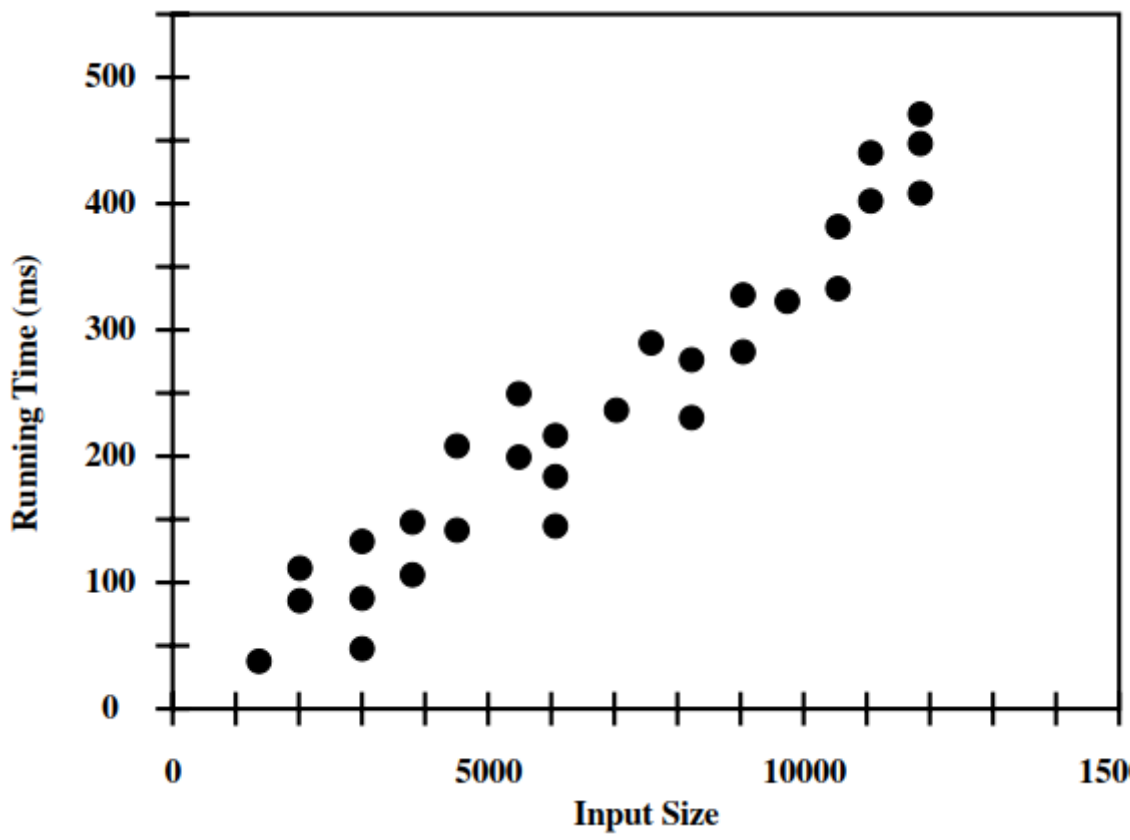
Введение в анализ сложности алгоритмов

Результат экспериментального определения продолжительности исполенения алгорима для различного объема входных данных:



Сложности экспериментального анализа эффективности алгоритма:

- Эксперементальное время работы двух алгоритмов сложно напрямую сравнить до тех пор, пока эксперименты не будут проведены на одинковом аппаратном обеспечении и в одинаковом программном окружении.
- Эксперементы могут быть вопленены только для ограниченного набора тестовых входных данных. Таким образом они не учитывают время работы на входных данных, которые не были включены в набор тестовых входных данных, а эти результаты могут быть важны.
- Для эксперементальной оценки производительности алгоритма он должен быть полностью реализован на одном из языков программирования.

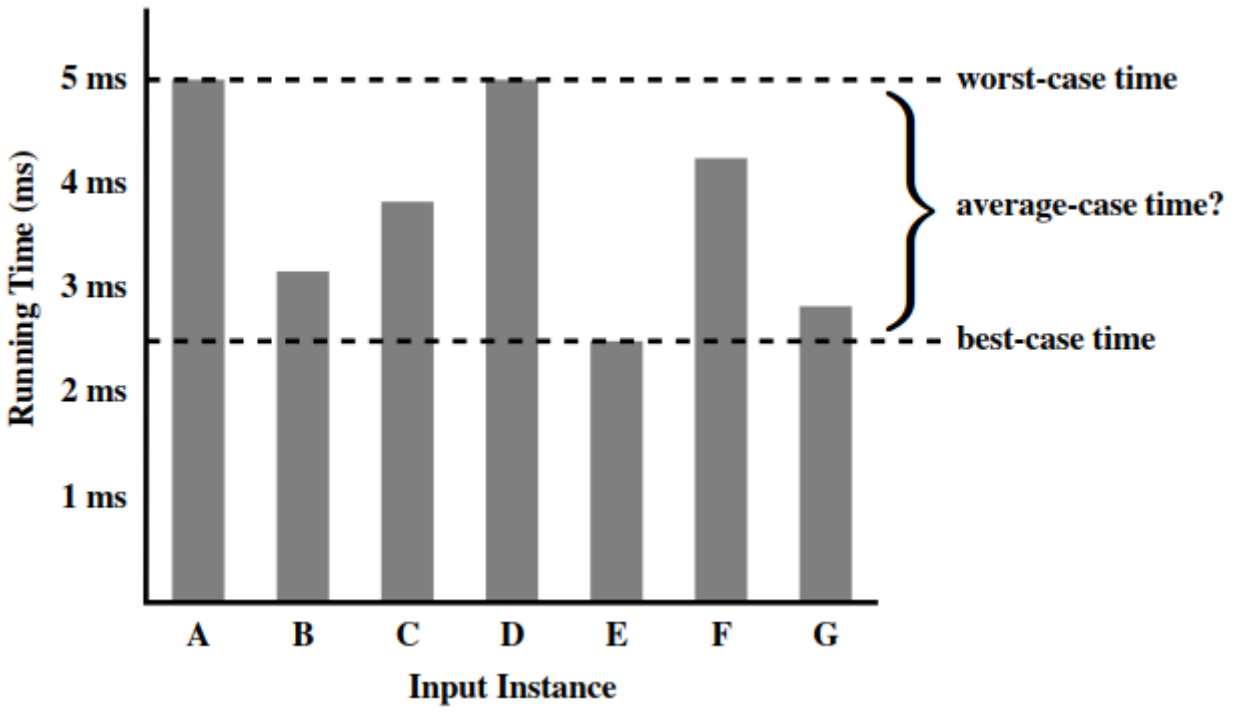
Подсчет базовых операций

Для анализа времени исполнения алгоритма без его эксперементального выполнения мы можем выполнить непосредственный аналз на базе высокоуровневого описания алгоритма (в виде псевдо-кода или фрагмента кода на настоящем языке программирования).

Набор рассматриваемых примитивных операций:

- Присовение значения идентификатору.
- Выполнение арифметической операции (например, сложения двух чисел).
- Сравнение двух чисел.
- Получение одного элемента из контейнера (например списка) по индексу.
- Выполнение вызова функции (операции выполняемые внутри функции должны учитываться дополнительно).
- Возврат из функции.

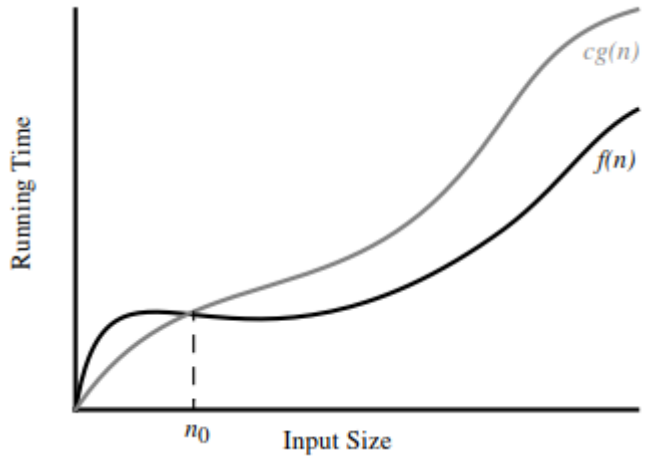
Разница между лучшим и худшим результатом работы алгоритма (длительностью выполнения). Каждый столбец представляет время выполнения алгоритма на различных входных значениях:



При анализе сложности алгоритмов принято концентрироваться на наиболее худшем результате работы алгоритма.

__ Асимптотическая оцнка сложности алгоритма__

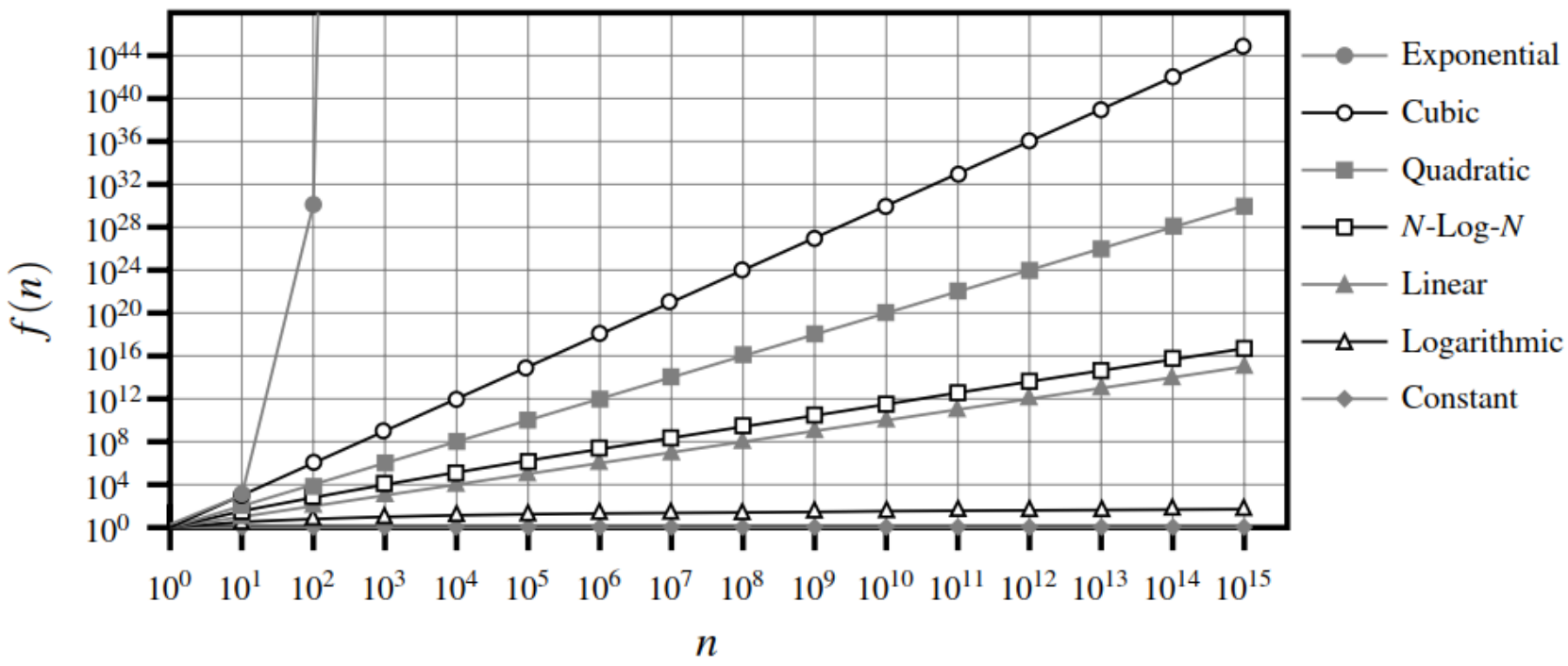
Путь функии $f(n)$ и $g(n)$ отображают положительные целые числа на положительные действительные числа. Тогда $f(n)$ является $\boldsymbol{O}(g(n))$ если существует действительная константа $c > 0$ и цела константа $n_0 \geq 1$ такая, что $f(n) \leq cg(n)$, для $n \geq n_0$.



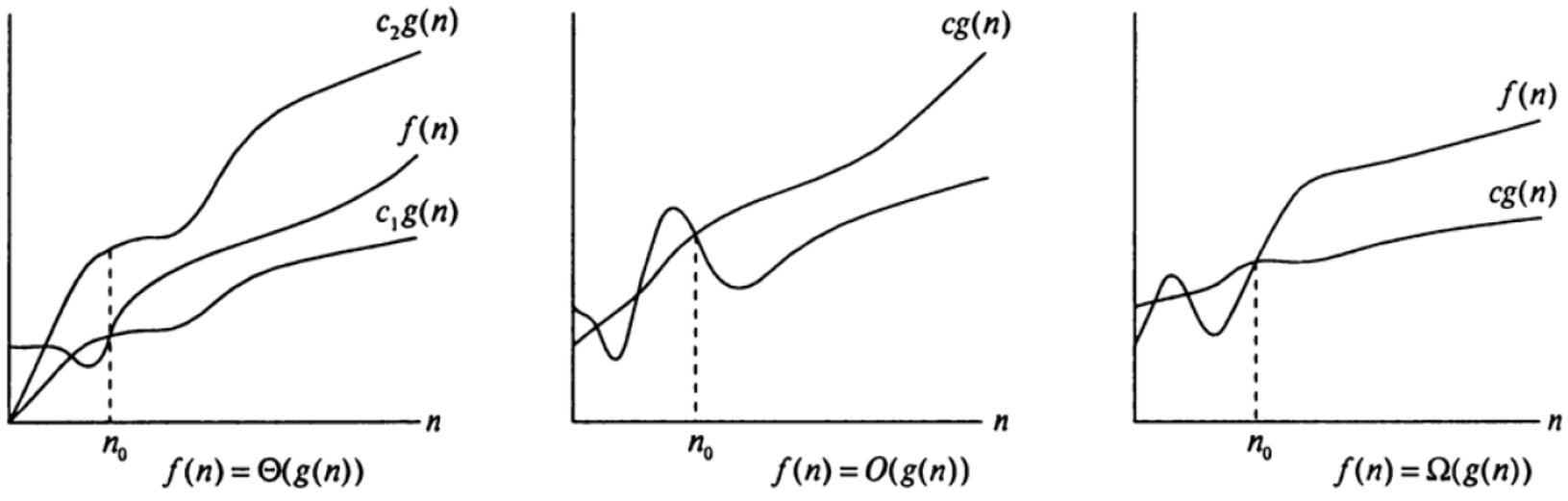
Функции которые часто используются для асимптотических оценок сложности алгоритмов:

constant	logarithm	linear	*n*-log-*n*	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

Графики этих функций:

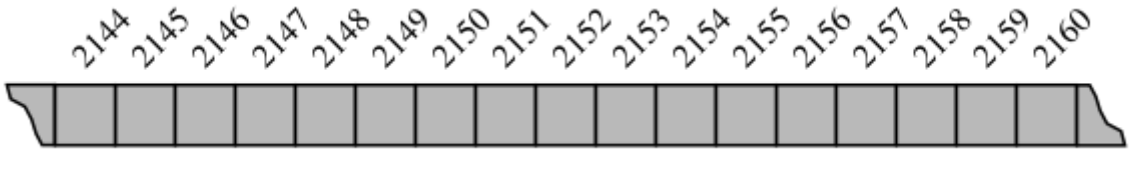


Другие виды асимптотических оценок.



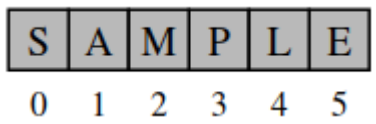
Массивы

Область памяти (RAM):



In [4]:
```
s1 = 'SAMPLE'
```

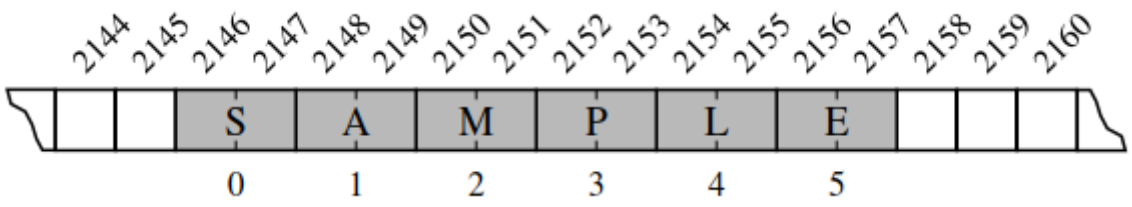
Высокоуровневая абстракция строки:



In [5]:
```
s1[0],s1[3]
```

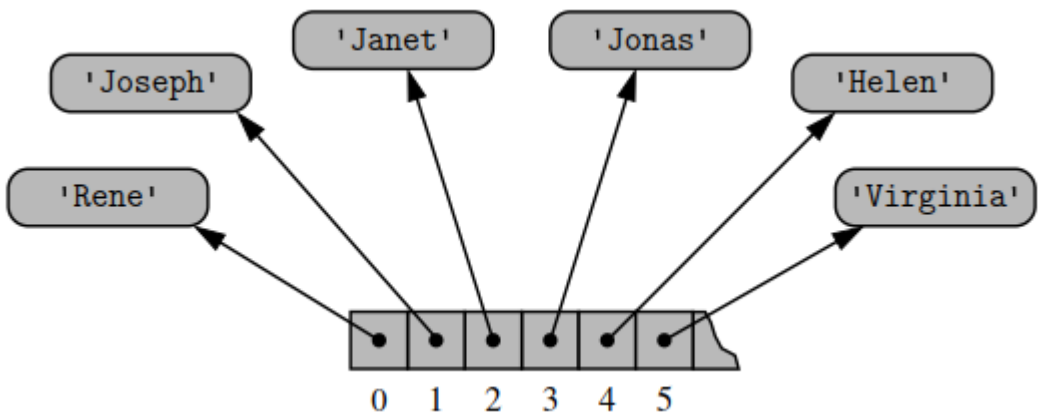
Out[5]:
```
('S', 'P')
```

Реальное расположение строки в памяти (RAM):



In [3]:
```
ra1 = ['Rene', 'Joseph', 'Janet', 'Jonas', 'Helen', 'Virginia', 'Mary' ]
```

Работа массива ссылок:



In [6]:
```
s1 = 'SAMPLE' # в действительности хранятся ссылки на буквы!
```

In [2]:
```
import array # модуль для работы с массивами в Python
```

https://docs.python.org/3/library/array.html (https://docs.python.org/3/library/array.html)

Массив arryay хранит непосредственно объекты (а не ссылки на них), что обеспечивает более быстрый доступ и значительно меньший объем памяти для хранения тех же данных (на типе int экономия памяти может составлять 4-5 раз!). Минус: массивы, так же как и в других языках могут хранить только объекты одного типа.

In [3]:
```
ar1 = array.array('i' , [2, 3, 5, 7, 11, 13, 17, 19, 21, 25])
len(ar1)
```

Out[3]:
```
10
```

In [4]:
```
ar1[0], ar1[4]
```

Out[4]:
```
(2, 11)
```

In [5]:
```
import sys
```

In [6]:
```
sys.getsizeof(ar1)
```

Out[6]:
```
104
```

In [8]:
```
ls2 = [2, 3, 5, 7, 11, 13, 17, 19, 21, 25]
sys.getsizeof(ls2), len(ls2)
```

Out[8]:
```
(144, 10)
```

In [9]:
```
sys.getsizeof(2), sys.getsizeof(777)
```

Out[9]:
```
(28, 28)
```

In [10]:
```
sys.getsizeof(ls2) + len(ls2) * sys.getsizeof(2)
```

Out[10]:
```
424
```

Класс array.array(TypeCode [, инициализатор]) - новый массив, элементы которого ограничены TypeCode, и инициализатор, который должен быть списком, объектом, поддерживающий интерфейс буфера, или итерируемый объект.

array.typecode - TypeCode символ, использованный при создании массива.

array.itemsize - размер в байтах одного элемента в массиве.

array.append(x) - добавление элемента в конец массива.

array.buffer_info() - кортеж (ячейка памяти, длина). Полезно для низкоуровневых операций.

array.byteswap() - изменить порядок следования байтов в каждом элементе массива. Полезно при чтении данных из файла, написанного на машине с другим порядком байтов.

array.count(x) - возвращает количество вхождений x в массив.

array.extend(iter) - добавление элементов из объекта в массив.

array.frombytes(b) - делает массив array из массива байт. Количество байт должно быть кратно размеру одного элемента в массиве.

array.fromfile(F, N) - читает N элементов из файла и добавляет их в конец массива. Файл должен быть открыт на бинарное чтение. Если доступно меньше N элементов, генерируется исключение EOFError , но элементы, которые были доступны, добавляются в массив.

array.fromlist(список) - добавление элементов из списка.

array.index(x) - номер первого вхождения x в массив.

array.insert(n, x) - включить новый пункт со значением x в массиве перед номером n. Отрицательные значения рассматриваются относительно конца массива.

array.pop(i) - удаляет i-ый элемент из массива и возвращает его. По умолчанию удаляется последний элемент.

array.remove(x) - удалить первое вхождение x из массива.

array.reverse() - обратный порядок элементов в массиве.

array.tobytes() - преобразование к байтам.

array.tofile(f) - запись массива в открытый файл.

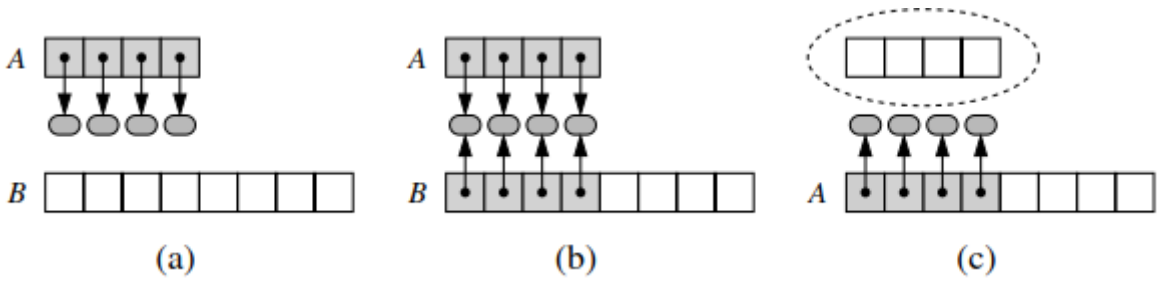
array.tolist() - преобразование массива в список.

Динамические массивы

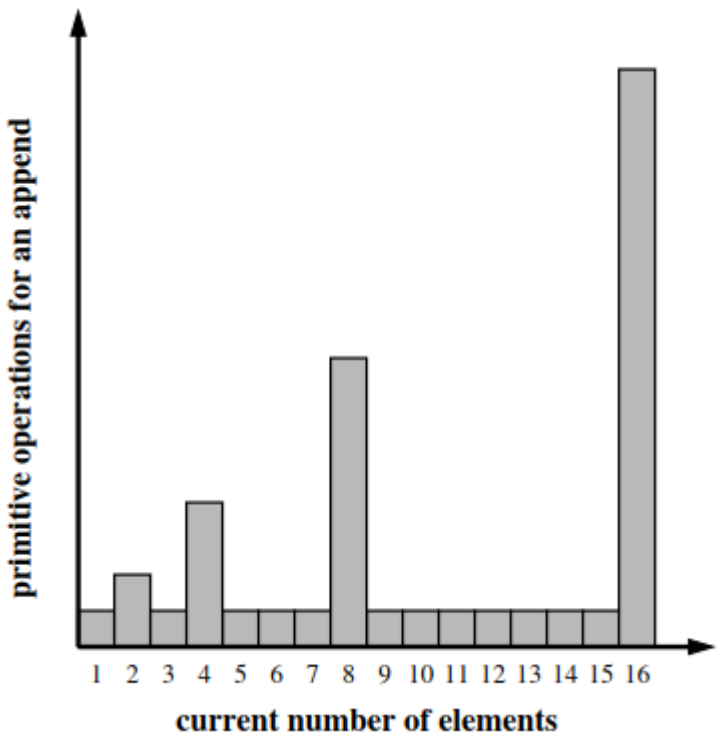
In [12]:
```
data = []
n = 26
for k in range(n):
    a = len(data) # количество элементов в списке
    b = sys.getsizeof(data) # текущий размер списка в байтах
    print('Length: {0:3d}; Size in bytes: {1:4d}'.format(a, b))
    data.append(None) # добавляем элемент в список
```

```
Length:   0; Size in bytes:   64
Length:   1; Size in bytes:   96
Length:   2; Size in bytes:   96
Length:   3; Size in bytes:   96
Length:   4; Size in bytes:   96
Length:   5; Size in bytes:  128
Length:   6; Size in bytes:  128
Length:   7; Size in bytes:  128
Length:   8; Size in bytes:  128
Length:   9; Size in bytes:  192
Length:  10; Size in bytes:  192
Length:  11; Size in bytes:  192
Length:  12; Size in bytes:  192
Length:  13; Size in bytes:  192
Length:  14; Size in bytes:  192
Length:  15; Size in bytes:  192
Length:  16; Size in bytes:  192
Length:  17; Size in bytes:  264
Length:  18; Size in bytes:  264
```

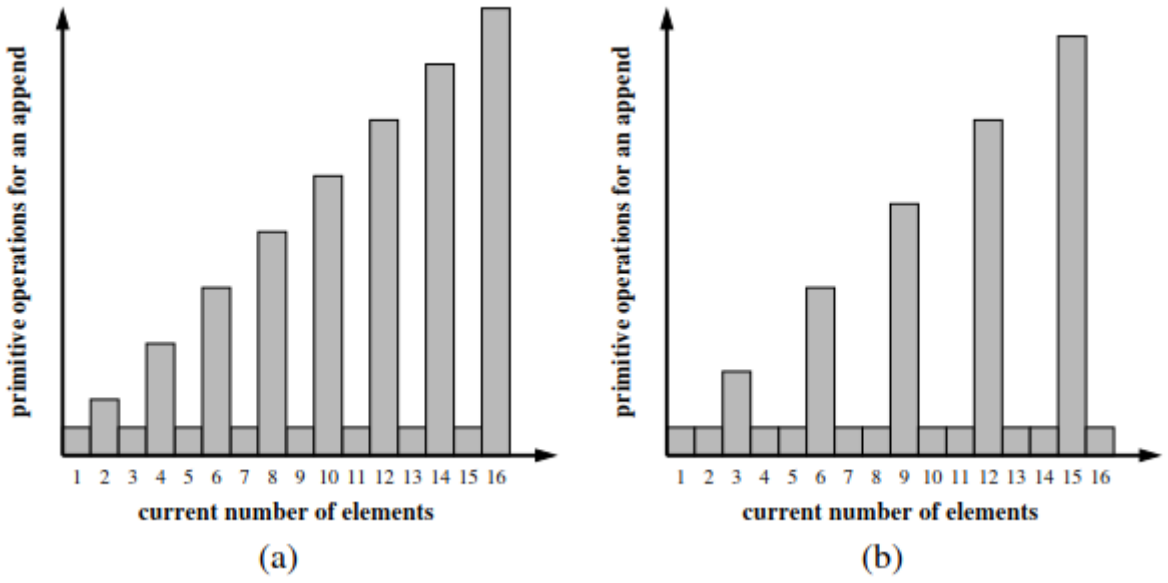
Процес роста динамического массива:



Длительность вставки элемента в конеец динамичекого массива:



Длительность вставки элемента в конеец динамичекого массива при арифметической прогрессии длинны массива:



Асимптотическая эффективность операций для списка (кортежа) (list/tuple) не меняющих содержимое структуры данных.

Идентификаторы объектов: data или data1 и data2, и соответственно их длины: n, n1 и n2. k обозначает индекс первого вхождения (слева) искомого элемента, в случае отсутствия элемента k=n. При сравнении двух последовательностей k показывает самый левый элемент на котром последовательности не согласуются или k = min(n1 , n2).

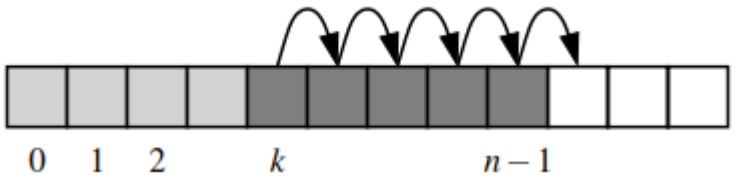
Operation	Running Time
len(data)	$O(1)$
data[j]	$O(1)$
data.count(value)	$O(n)$
data.index(value)	$O(k+1)$
value **in** data	$O(k+1)$
data1 == data2 (similarly !=, <, <=, >, >=)	$O(k+1)$
data[j:k]	$O(k-j+1)$
data1 + data2	$O(n_1+n_2)$
c * data	$O(cn)$

Асимптотическая эффективность операций для списка (list) меняющих содержимое структуры данных.

Operation	Running Time
data[j] = val	$O(1)$
data.append(value)	$O(1)^*$
data.insert(k, value)	$O(n-k+1)^*$
data.pop( )	$O(1)^*$
data.pop(k) **del** data[k]	$O(n-k)^*$
data.remove(value)	$O(n)^*$
data1.extend(data2) data1 += data2	$O(n_2)^*$
data.reverse( )	$O(n)$
data.sort( )	$O(n \log n)$

*amortized

Освобождение места для вставки элемента в список (list):

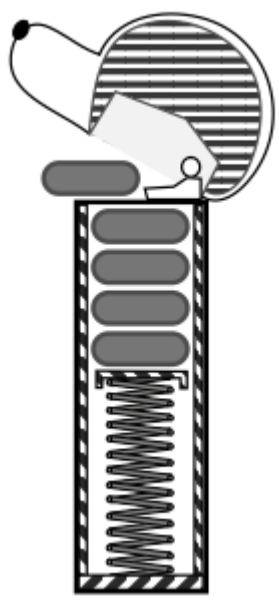


Среднее време операции insert(k, val), измеренное в микросекундах (N - длина списка):

	N				
	100	1,000	10,000	100,000	1,000,000
k = 0	0.482	0.765	4.014	36.643	351.590
k = n // 2	0.451	0.577	2.191	17.873	175.383
k = n	0.420	0.422	0.395	0.389	0.397

Стек

Пример физической реализации стека:



Стек реализует принцип LIFO - last-in, first-out.

Абстрактная структура данных Стек:

Базовые операции для работы со стеком:

S.push(e): добавление элемента e на вершину стека S.

S.pop(): удаляет и возвращает верхний элемент стека S. Если стек пуст, то возникает ошибка.

In [23]:
```
from collections import deque
```

In [28]:
```
class Stack(deque):
    def push(self, a):
        self.append(a)
```

In [32]:
```
s1 = Stack()
s1
```

Out[32]:
```
Stack([])
```

In [33]:
```
s1.push(5)
s1
```

Out[33]:
```
Stack([5])
```

In [34]:
```
s1.push(3)
s1
```

Out[34]:
```
Stack([5, 3])
```

In [35]:
```
s1.push(4)
s1
```

Out[35]:
```
Stack([5, 3, 4])
```

In [36]:
```
s1.pop()
```

Out[36]:
```
4
```

```
In [37]: s1
Out[37]: Stack([5, 3])

In [38]: s1.push(7)
         s1
Out[38]: Stack([5, 3, 7])

In [39]: s1.pop()
Out[39]: 7

In [41]: s1
Out[41]: Stack([5])

In [40]: s1.pop()
Out[40]: 3

In [42]: s1
Out[42]: Stack([5])

In [43]: s1.pop()
Out[43]: 5

In [44]: s1.pop()
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-44-da63ff21a00b> in <module>()
----> 1 s1.pop()

IndexError: pop from an empty deque
```

Дополнительные операции для работы со стеком:

S.top(): возвращает верхний элемент стека S не удаляя его. Если стек пуст, то возникает ошибка.

S.is_empty( ): возвращает True если стек S не содержит ни одного элемента.

len(S): возвращает количество элементов в стееаке S.

Реализация стека на базе списка (list):

Stack Method	*Realization with Python list*
S.push(e)	L.append(e)
S.pop( )	L.pop( )
S.top( )	L[−1]
S.is_empty( )	len(L) == 0
len(S)	len(L)

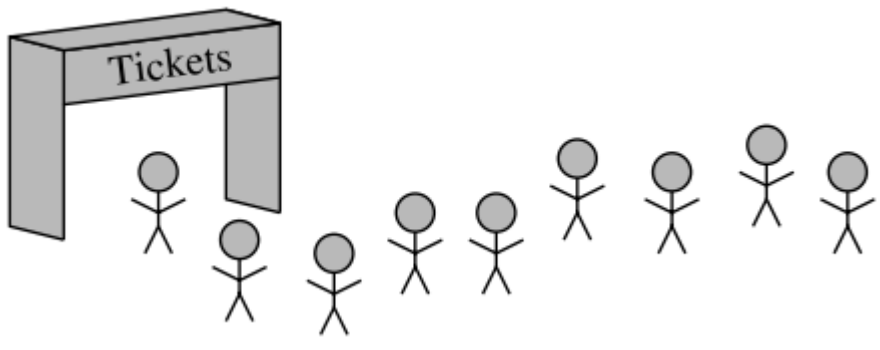
Производительность реализации стека на базе динамического массива:

Operation	Running Time
S.push(e)	$O(1)^*$
S.pop( )	$O(1)^*$
S.top( )	$O(1)$
S.is_empty( )	$O(1)$
len(S)	$O(1)$

*amortized

Очередь

Пример физической реализации очереди:



Очередь реализует принцип FIFO - first-in, first-out.

Абстрактная структура данных Очередь:

Базовые операции для работы с очередью:

Q.enqueue(e): добавление элемента e в конец очереди Q.

Q.dequeue(): удаляет и возвращает первый элемент очереди Q. Если очередь пуста, то возникает ошибка.

In [53]:
```
class Queue(deque):
    def enqueue(self, a):
        self.append(a)

    def dequeue(self):
        return self.popleft()
```

In [60]:
```
q1 = Queue()
q1
```

Out[60]: Queue([])

In [61]:
```
q1.enqueue(5)
q1
```

Out[61]: Queue([5])

In [62]:
```
q1.enqueue(3)
q1
```

Out[62]: Queue([5, 3])

In [63]:
```
q1.enqueue(4)
q1
```

Out[63]: Queue([5, 3, 4])

In [64]:
```
q1.dequeue()
```

Out[64]: 5

In [65]:
```
q1
```

Out[65]: Queue([3, 4])

In [66]:
```
q1.enqueue(7)
q1
```

Out[66]: Queue([3, 4, 7])

In [ ]:
```

```

In [67]:
```
q1.dequeue()
```

Out[67]: 3

In [68]:
```
q1
```

Out[68]: Queue([4, 7])

In [69]:
```
q1.dequeue()
```

Out[69]: 4

In [70]:
```
q1
```

Out[70]: Queue([7])

In [71]:
```
q1.dequeue()
```

Out[71]: 7

In [72]:
```
q1
```

Out[72]: Queue([])

In [73]:
```
q1.dequeue()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-73-accce75173e7> in <module>()
----> 1 q1.dequeue()

<ipython-input-53-d94ceb61bf91> in dequeue(self)
      4
      5     def dequeue(self):
----> 6         return self.popleft()

IndexError: pop from an empty deque
```

Дополнительные операции для работы с очередью:

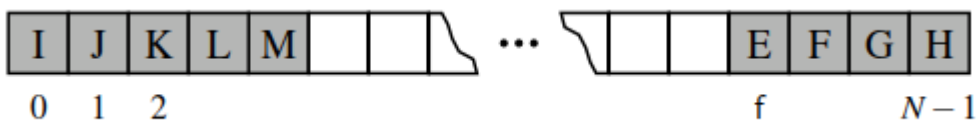
Q.first(): возвращает первый элемент очереди Q не удаляя его. Если очередь пуста, то возникает ошибка.

Q.is empty(): возвращает True если очередь Q не содержит ни одного элемента.

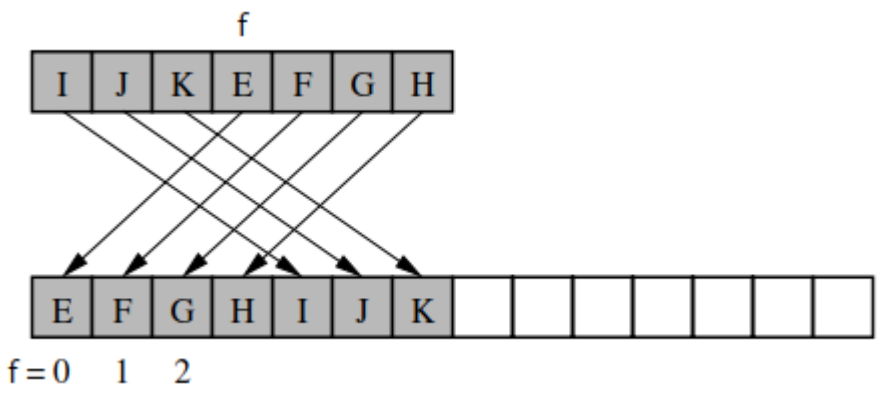
len(Q): возвращает количество элементов в очереди Q.

Реализация очереди на базе массива: позволяем началу очереди смещаться относительно начала массива:

Реализация очереди на базе закольцованного массива:



Изменение размера закольцованного массива, на базе которого реализована очередь:



Производительность реализации очереди на базе динамического массива:

Operation	Running Time
Q.enqueue(e)	$O(1)^*$
Q.dequeue( )	$O(1)^*$
Q.first( )	$O(1)$
Q.is_empty( )	$O(1)$
len(Q)	$O(1)$

*amortized

Реализация deque в модуле collections: https://docs.python.org/3.1/library/collections.html#collections.deque (https://docs.python.org/3.1/library/collections.html#collections.deque) deque позволяет эффективно добавлять и удалять значения из начала/конца очереди.

Быстрые операции с концом (правым концом) очереди в deque (аналогичны операциям для списка):

append(x) Add x to the right side of the deque.

pop() Remove and return an element from the right side of the deque. If no elements are present, raises an IndexError.

Быстрые операции с началом (левым концом) очереди в deque (в списке нет аналогов этих операций):

appendleft(x) Add x to the left side of the deque.

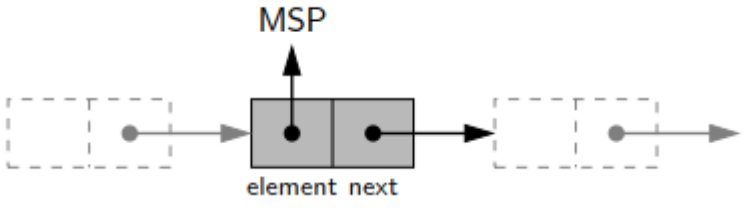
popleft() Remove and return an element from the left side of the deque. If no elements are present, raises an IndexError.

Реализация deque основана на двунаправленных связанных списках массивов фиксированной длины. deque поддерживает итерацию, операции **len(d)**, **reversed(d)**, проверку вхождения с помощью оператора **in** . Операции получения элемента по индексу (такие как `d[0]`, `d[-1]`) быстрые (имеют сложность **O(1)** для двух концов очереди, но длительные (сложность **O(n)**) для элментов в середине списка. Т.е. для быстрого произвольного доступа к элементам списка нужно использовать **list** вместо **deque**.

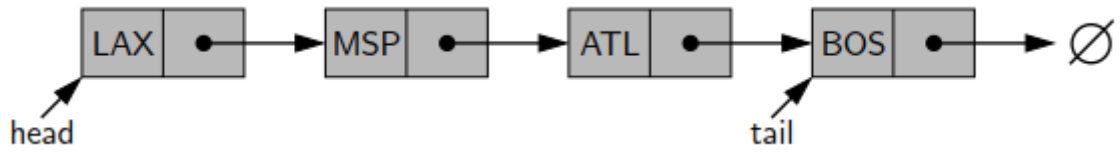
Связанные списки

Однонаправленные связанные списки

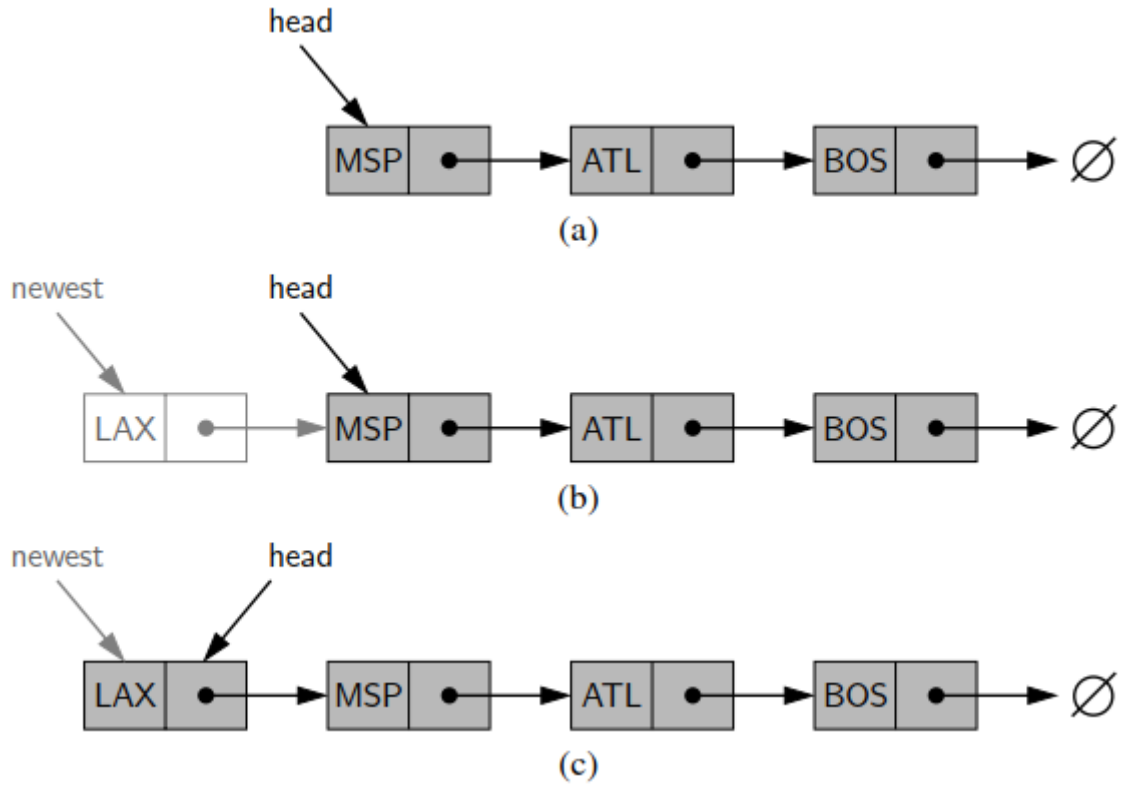
Пример узла, являющегося частью однонаправленного связного списка. Узел ссылается на объект, который является элементом последовательности и на следующий узел списка (или хранит значение None, если в списке больше нет узлов):



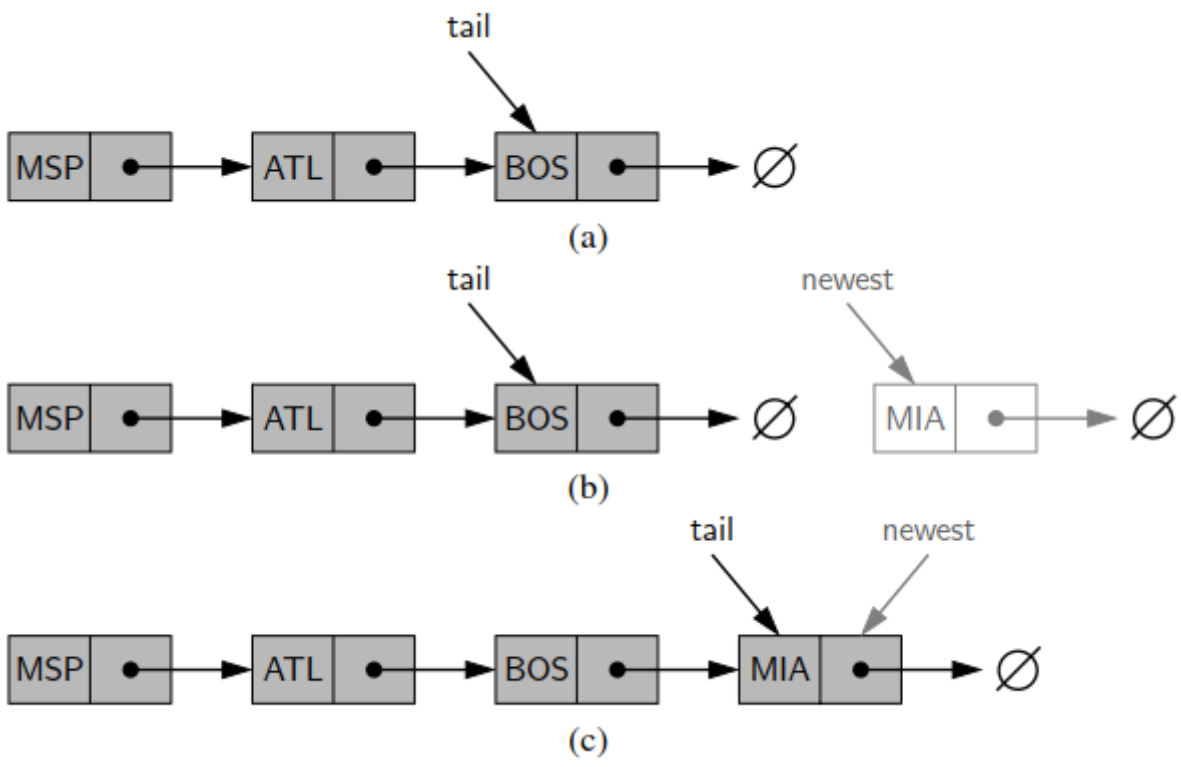
Упрощенная илюстрация однонаправленного связаного списка (для упрощения, объекты, являющиеся элементами последовательности, представленны встроенными в узлы, а не внешниими объектами, на которые ссылаются узлы)



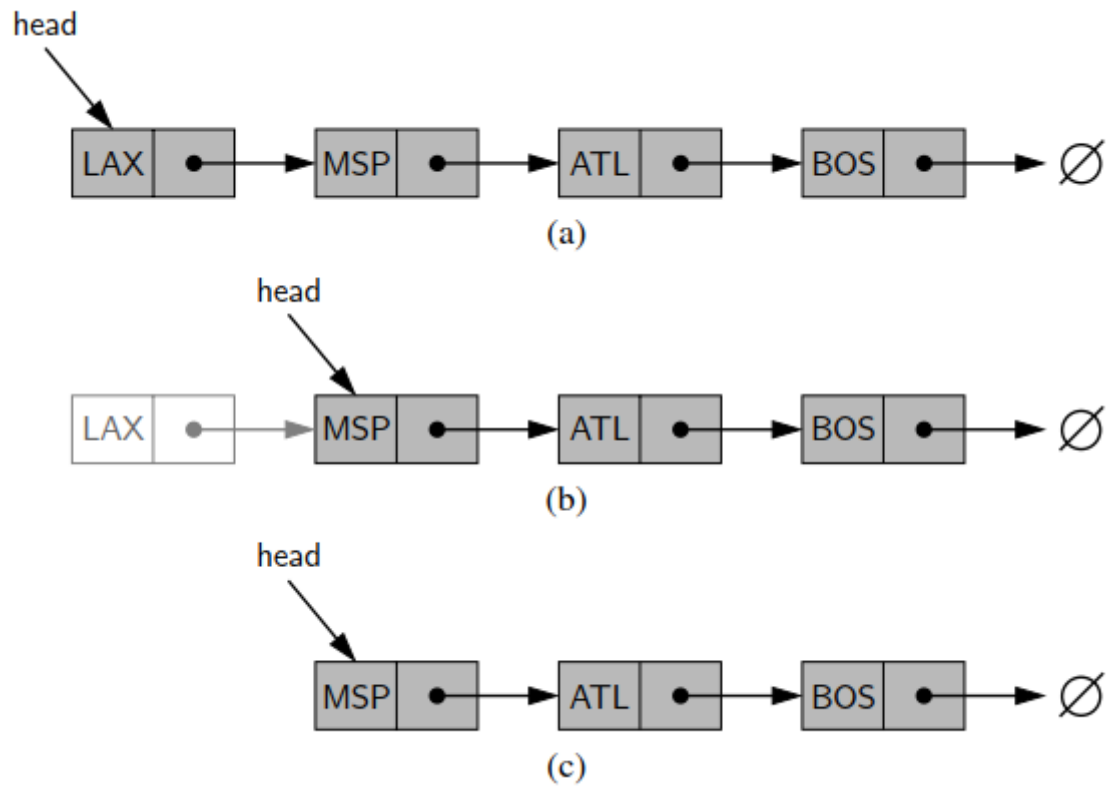
Процедура вставки узла в начало (head) однонаправленного связного списка (a - до вставки, b - после создания нового элемента, c - после переприсвоения значения ссылки на первый узел).



Процедура вставки узла в конец (tail) однонаправленного связного списка (a - до вставки, b - после создания нового элемента, c - после переприсвоения значения ссылки на последний узел).



Процедура удаления узла из начала (head) однонаправленного связного списка (a - до удаления, b - после изменения ссылки на первый узел списка, c - итоговая конфигурация).

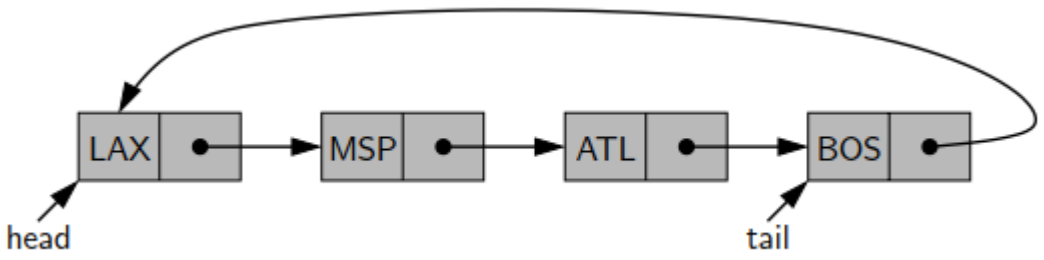


Производительность реализации стэка на основе связанного списка. Все значения определяют консервативные оценки сложности.

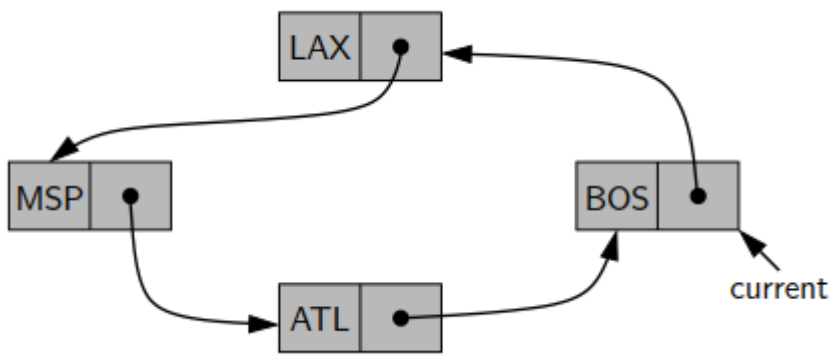
Operation	Running Time
S.push(e)	$O(1)$
S.pop( )	$O(1)$
S.top( )	$O(1)$
len(S)	$O(1)$
S.is_empty( )	$O(1)$

Циклические связные списки

Пример однонаправленного связного списка с циклической структурой.

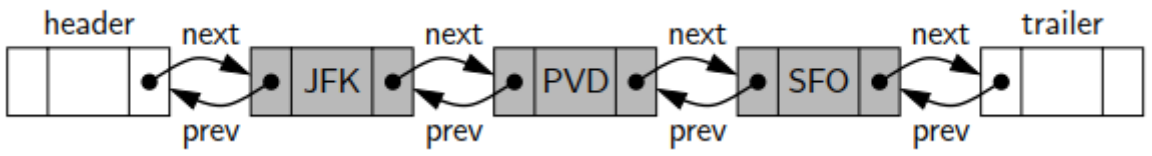


Пример однонаправленного связного списка с циклической структурой. current указывает на выделенный элемент списка.

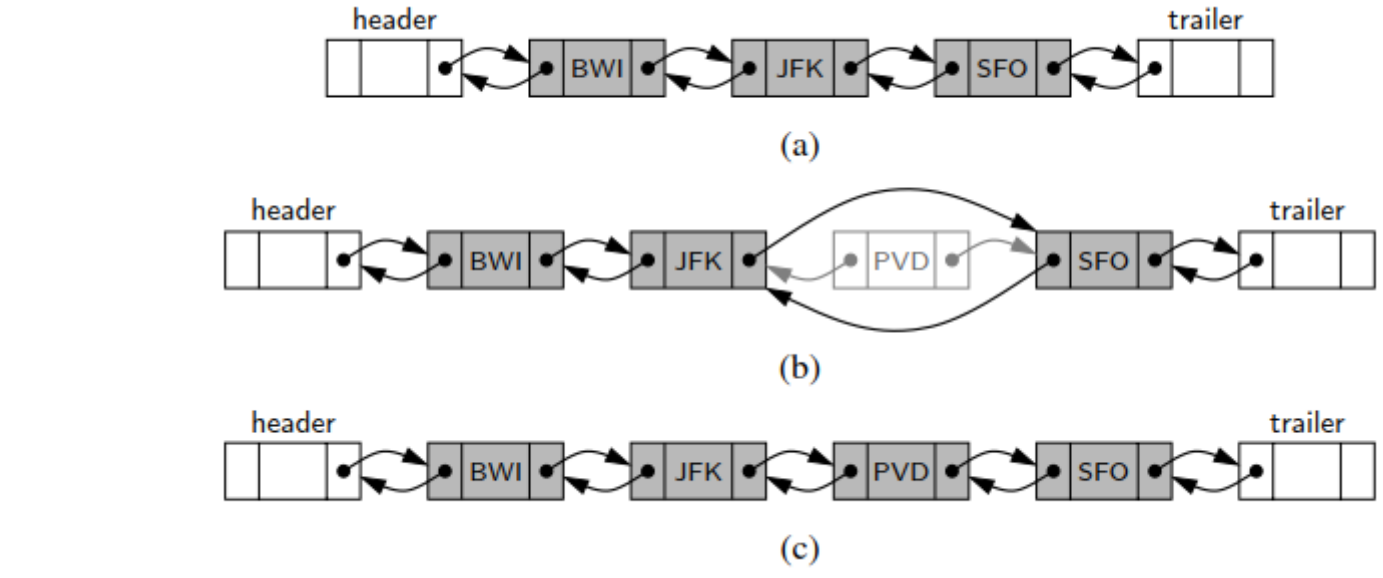


Двунаправленные связные списки

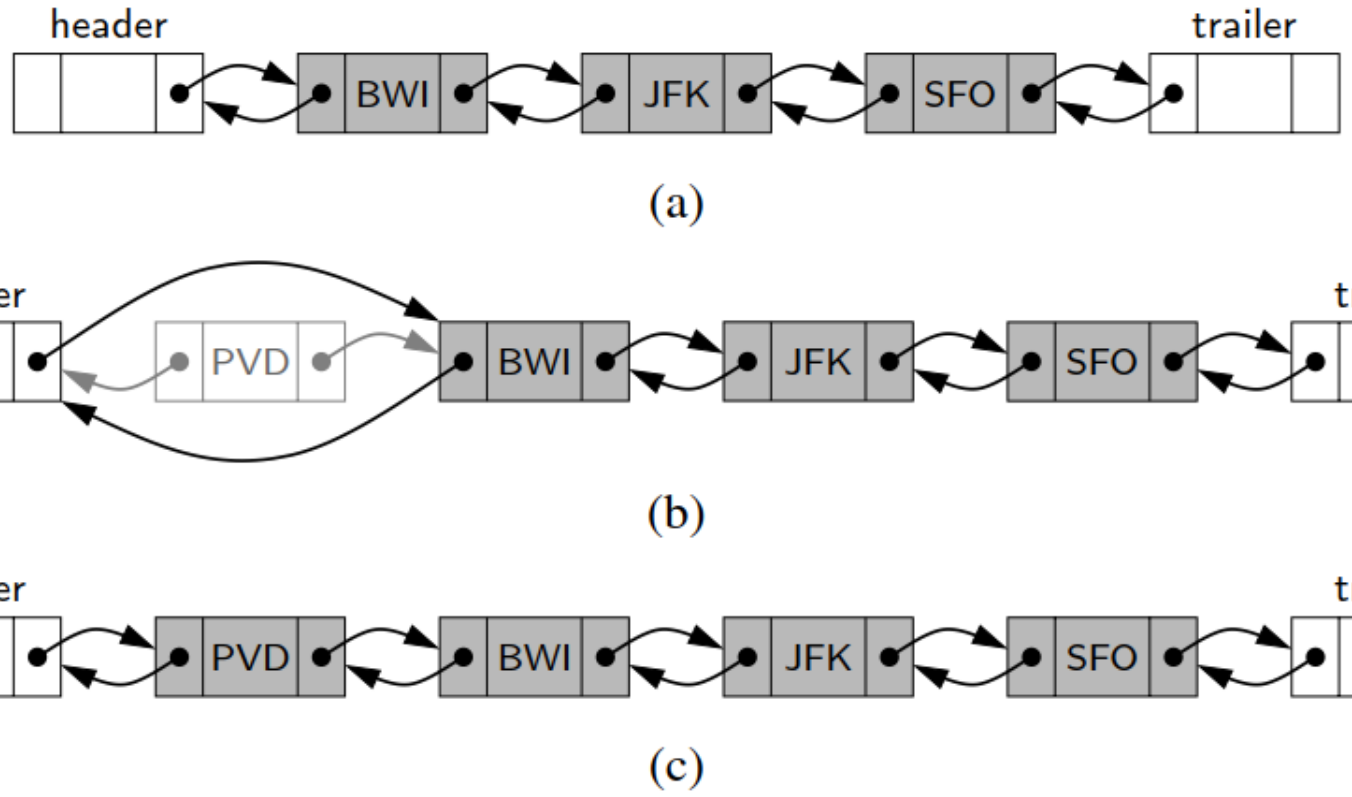
Пример двунаправленного связного списка используюещго специальные узлы-ограничители (header, trailer).



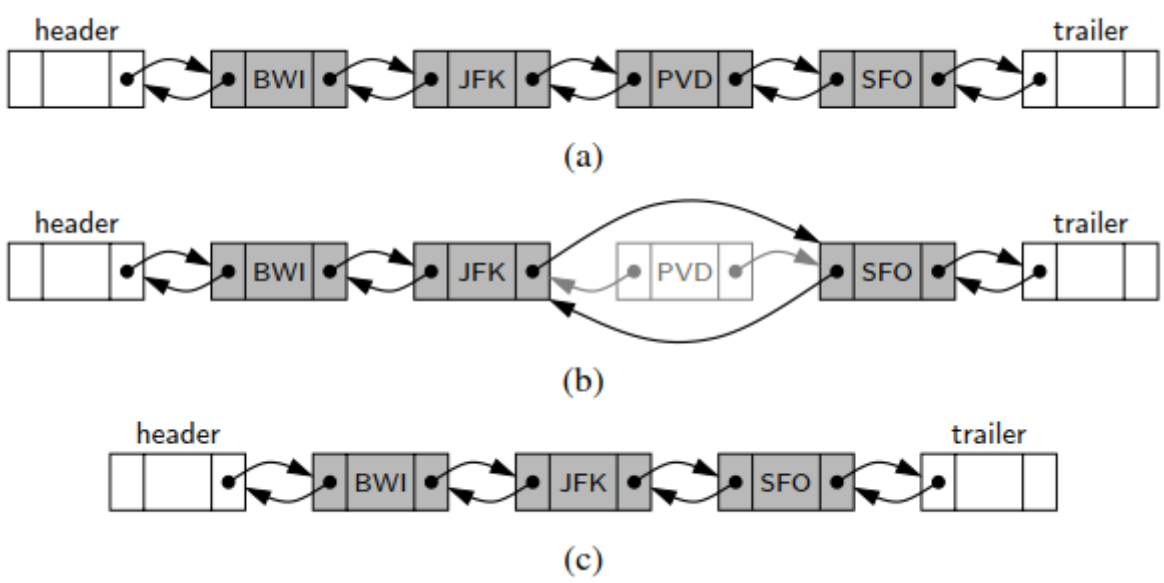
Добавление узла в двунаправленный связаный список с ограничетлями: a - до операции, b - после создания узла, c - после операции.



Добавление узла в начало двунаправленного связаного списка с ограничетлями: a - до операции, b - после создания узла, c - после операции.



Удаление элемента из двунаправленного связаного списка с ограничетлями: a - до операции, b - после удаления связей с узлом, c - после операции.



Сравнение скорости выполнения операций в разных реализациях абстрактных структур данных:

https://en.wikipedia.org/wiki/Linked_list (https://en.wikipedia.org/wiki/Linked_list)

In [ ]:

In [ ]:

Advantages of Array-Based Sequences

• Arrays provide O(1)-time access to an element based on an integer index.

The ability to access the kth element for any k in O(1) time is a hallmark advantage of arrays (see Section 5.2). In contrast, locating the kth element in a linked list requires O(k) time to traverse the list from the beginning, or possibly O(n − k) time, if traversing backward from the end of a doubly linked list.

• Operations with equivalent asymptotic bounds typically run a constant factor more efficiently with an array-based structure versus a linked structure. As an example, consider the typical enqueue operation for a queue. Ignoring the issue of resizing an array, this operation for the ArrayQueue class (see Code Fragment 6.7) involves an arithmetic calculation of the new index, an increment of an integer, and

storing a reference to the element in the array. In contrast, the process for a LinkedQueue (see Code Fragment 7.8) requires the instantiation of a node, appropriate linking of nodes, and an increment of an integer. While this operation completes in O(1) time in either model, the actual number of CPU operations will be more in the linked version, especially given the instantiation of the new node.

• Array-based representations typically use proportionally less memory than linked structures. This advantage may seem counterintuitive, especially given that the length of a dynamic array may be longer than the number of elements that it stores. Both array-based lists and linked lists are referential structures, so the primary memory for storing the actual objects that are elements is the same for either structure. What differs is the auxiliary amounts of memory that are used by the two structures. For an array-based container of n elements, a typical worst case may be that a recently resized dynamic array has allocated memory for 2n object references. With linked lists, memory must be devoted not only to store a reference to each contained object, but also explicit references that link the nodes. So a singly linked list of length n already requires 2n references (an element reference and next reference for each node). With a doubly linked list, there are 3n references.

Advantages of Link-Based Sequences

• Link-based structures provide worst-case time bounds for their operations. This is in contrast to the amortized bounds associated with the expansion or contraction of a dynamic array (see Section 5.3).

When many individual operations are part of a larger computation, and we only care about the total time of that computation, an amortized bound is as good as a worst-case bound precisely because it gives a guarantee on the sum of the time spent on the individual operations.

However, if data structure operations are used in a real-time system that is de- signed to provide more immediate responses (e.g., an operating system, Web server, air traffic control system), a long delay caused by a single (amortized) operation may have an adverse effect.

• Link-based structures support O(1)-time insertions and deletions at arbi- trary positions. The ability to perform a constant-time insertion or deletion with the PositionalList class, by using a Position to efficiently describe the location of the operation, is perhaps the most significant advantage of the linked list.

This is in stark contrast to an array-based sequence. Ignoring the issue of resizing an array, inserting or deleting an element from the end of an array- based list can be done in constant time. However, more general insertions and deletions are expensive. For example, with Python's array-based list class, a call to insert or pop with index k uses O(n – k + 1) time because of the loop to shift all subsequent elements (see Section 5.4).

As an example application, consider a text editor that maintains a document as a sequence of characters. Although users often add characters to the end of the document, it is also possible to use the cursor to insert or delete one or more characters at an arbitrary position within the document. If the charac- ter sequence were stored in an array-based sequence (such as a Python list), each such edit operation may require linearly many characters to be shifted, leading to O(n) performance for each edit operation. With a linked-list representation, an arbitrary edit operation (insertion or deletion of a character at the cursor) can be performed in O(1) worst-case time, assuming we are given a position that represents the location of the cursor.

In [ ]: