Мини-задача #15 (1 балл)

Реализовать собственный динамический массив, в котором поддержаны операции:

1. Доступа к элементу по индексу
2. Добавление элемента в конец
3. Удаления последнего элемента

Операции работают за O(1) или O*(1).

Обрабатывается случаи выхода за границу массива и удаление элемента из пустого массива.

Пользоваться готовыми библиотечными решениями нельзя.

Мини-задача #16 (1 балл)

По заданному связному списку нужно найти цикл в нем и вернуть номер узла, с которого он начался.

Нельзя:

1. Модифицировать элементы списка
2. Пользоваться встроенными коллекциями

Нужно: использовать только O(1) дополнительной памяти.

Решение проверить на leetcode:
https://leetcode.com/problems/linked-list-cycle-ii/

Мини-задача #17 (1 балл)

По заданному связному списку и двум значениям элементов из него: построить новый связный список в котором все элементы между заданных в условии идут в обратном порядке.

Для решения используйте однопроходный алгоритм.

Решение проверить на leetcode:
https://leetcode.com/problems/reverse-linked-list-ii/

Алгоритмы и структуры данных

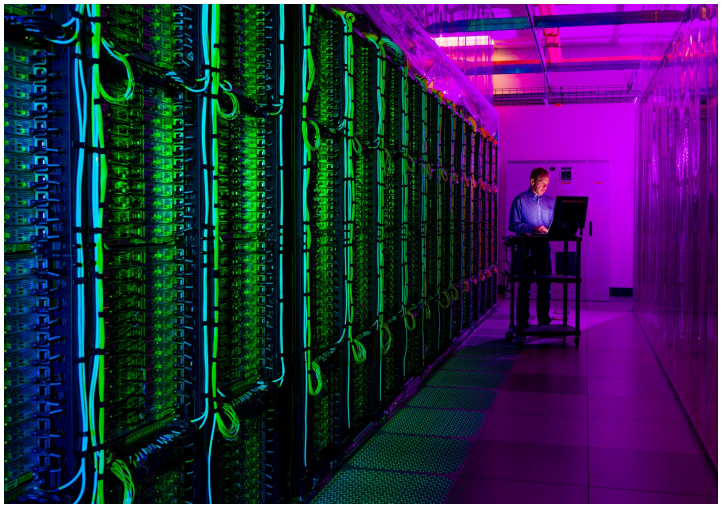
Элементарные структуры данных

Алгоритмы и структуры данных

Элементарные структуры данных

Задача

Реализовать хранилище однотипных данных (например, чисел)



Задача

Реализовать хранилище однотипных данных (например, чисел)

Которое бы поддерживало следующие запросы:

Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

1. Доступ к i-ому элементу

Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

1. Доступ к i-ому элементу

Как реализовывать будем?

Задача

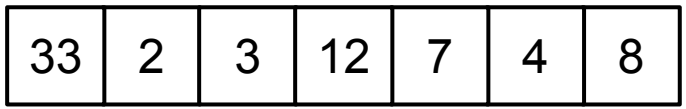
Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

1. Доступ к i-ому элементу

Как реализовывать будем?

Массив!



a[i] = *(a + i)

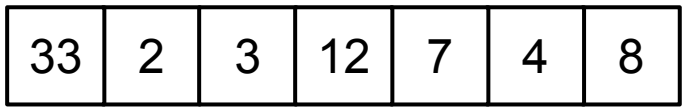
Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

1. Доступ к i-ому элементу

   Сложность?




```
a[i] = *(a + i)
```

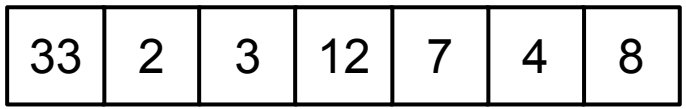
Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

1. Доступ к i-ому элементу

   Сложность? O(1)




```
a[i] = *(a + i)
```

Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

1. Доступ к i-ому элементу

---

2. Добавление элемента в "конец"

Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

1. Доступ к i-ому элементу

---

2. Добавление элемента в "конец"
3. Удаление последнего элемента

Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

1. Доступ к i-ому элементу

---

2. Добавление элемента в "конец"
3. Удаление последнего элемента

Как реализовывать будем?

Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

1. Доступ к i-ому элементу

---

2. Добавление элемента в "конец"
3. Удаление последнего элемента

Как реализовывать будем? Массив!

Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

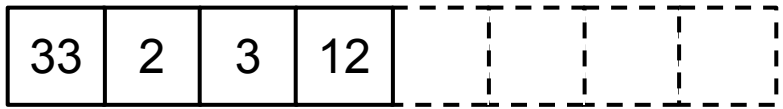
1. Доступ к i-ому элементу

---

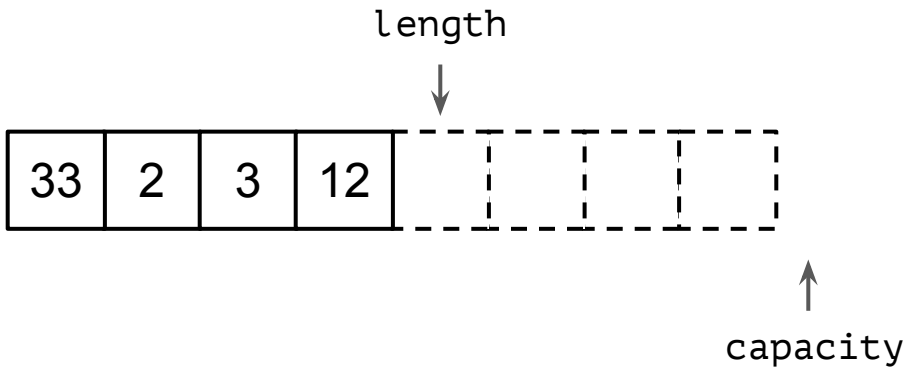
2. Добавление элемента в "конец"
3. Удаление последнего элемента

Как реализовывать будем? Динамический массив!

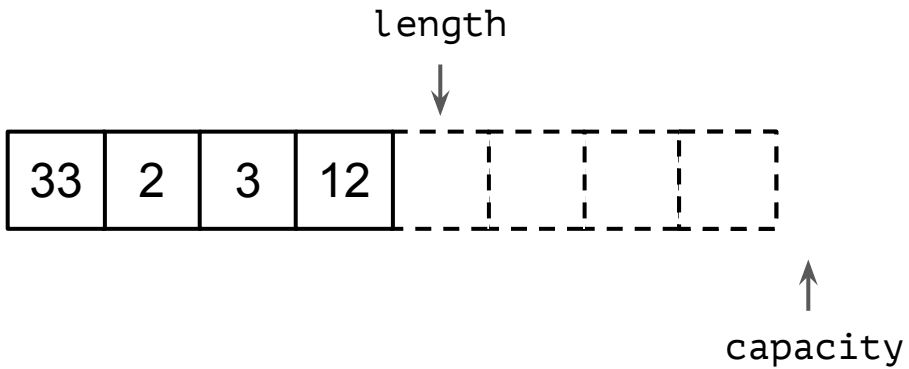
Динамический массив



Динамический массив

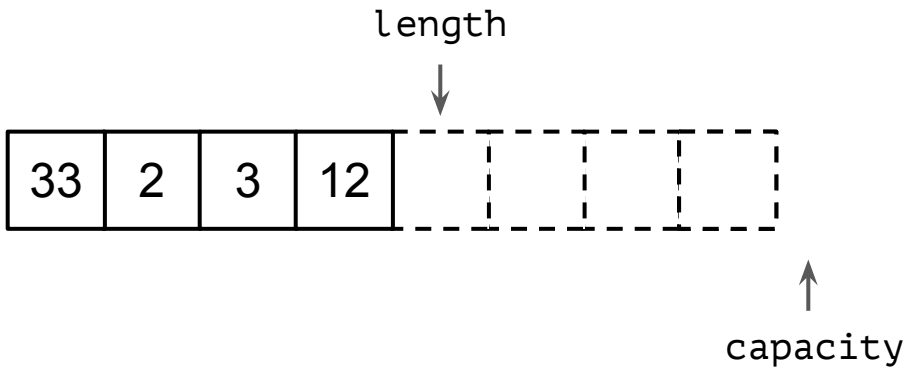


Динамический массив




```
def add(self, value: int):
    if self.length < self.capacity:
        self.data[self.length] = value
        self.length += 1
```

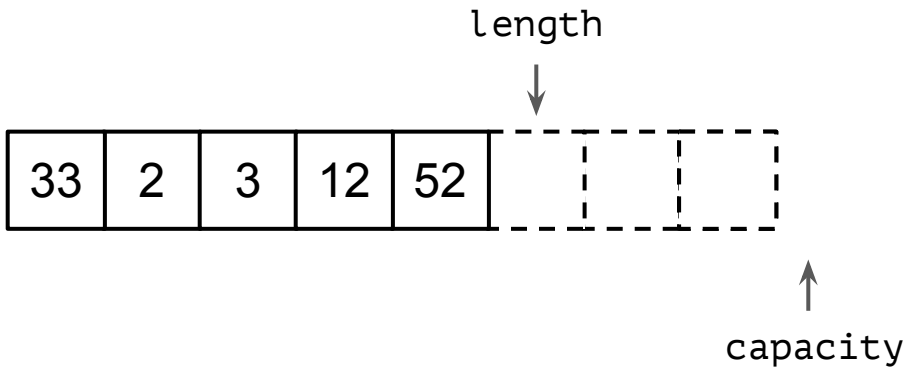
Динамический массив




```
vector = Vector(33, 2, 3, 12)
```

```
def add(self, value: int):
    if self.length < self.capacity:
        self.data[self.length] = value
        self.length += 1
```

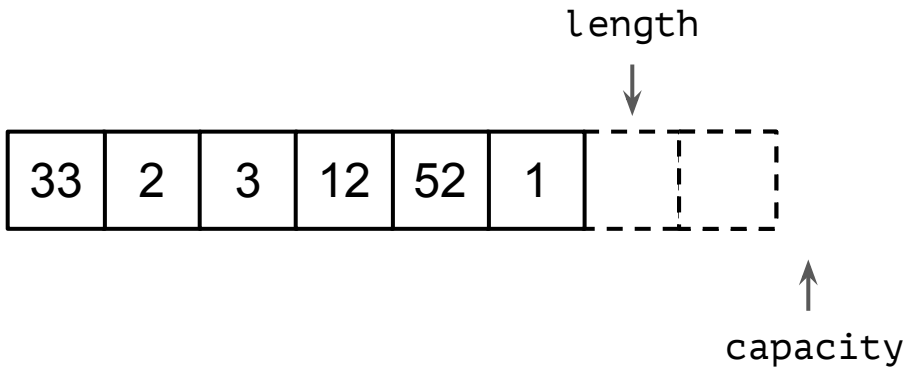
Динамический массив




```
vector = Vector(33, 2, 3, 12)
vector.add(52)
```

```
def add(self, value: int):
    if self.length < self.capacity:
        self.data[self.length] = value
        self.length += 1
```

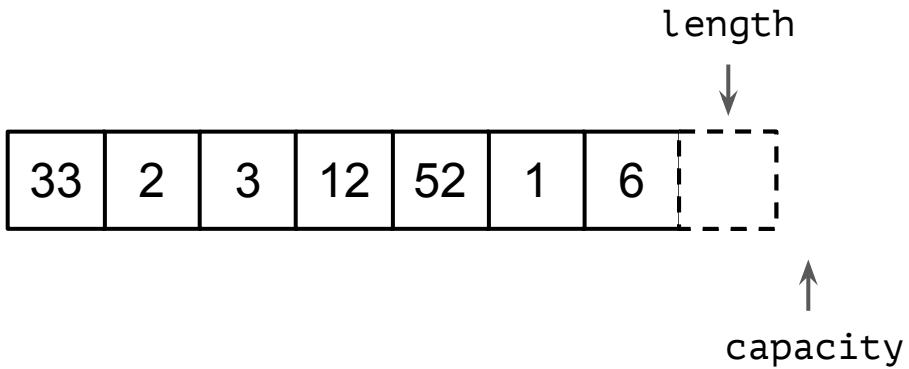
Динамический массив




```
vector = Vector(33, 2, 3, 12)
vector.add(52)
vector.add(1)
```

```
def add(self, value: int):
    if self.length < self.capacity:
        self.data[self.length] = value
        self.length += 1
```

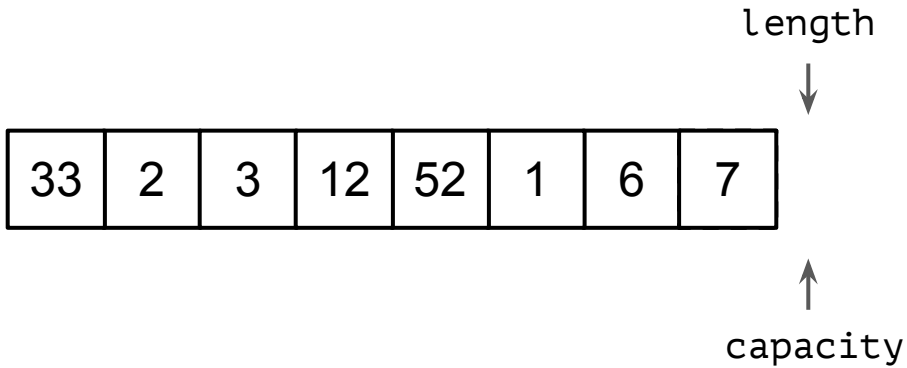
Динамический массив




```
vector = Vector(33, 2, 3, 12)
vector.add(52)
vector.add(1)
vector.add(6)
```

```
def add(self, value: int):
    if self.length < self.capacity:
        self.data[self.length] = value
        self.length += 1
```

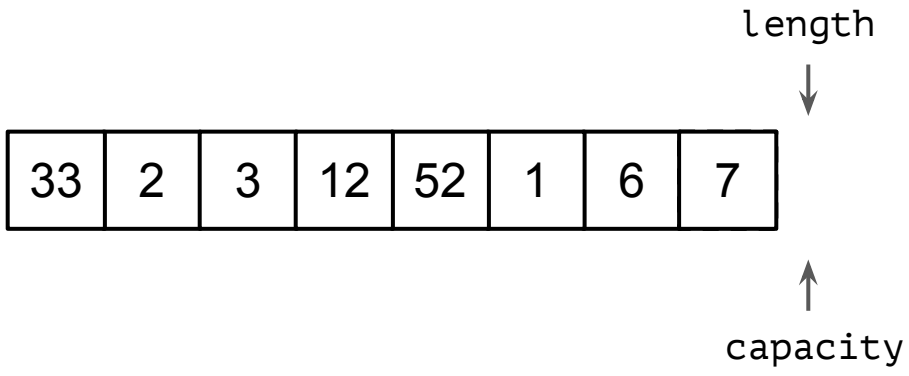
Динамический массив




```
vector = Vector(33, 2, 3, 12)
vector.add(52)
vector.add(1)
vector.add(6)
vector.add(7)
```

```
def add(self, value: int):
    if self.length < self.capacity:
        self.data[self.length] = value
        self.length += 1
```

Динамический массив

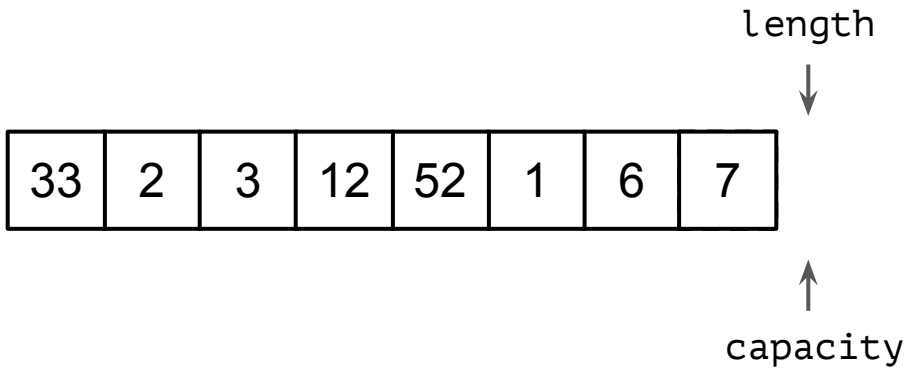



```
vector = Vector(33, 2, 3, 12)
vector.add(52)
vector.add(1)
vector.add(6)
vector.add(7)
vector.add(11)
```

```
def add(self, value: int):
    if self.length < self.capacity:
        self.data[self.length] = value
        self.length += 1
```

Что делать?

Динамический массив



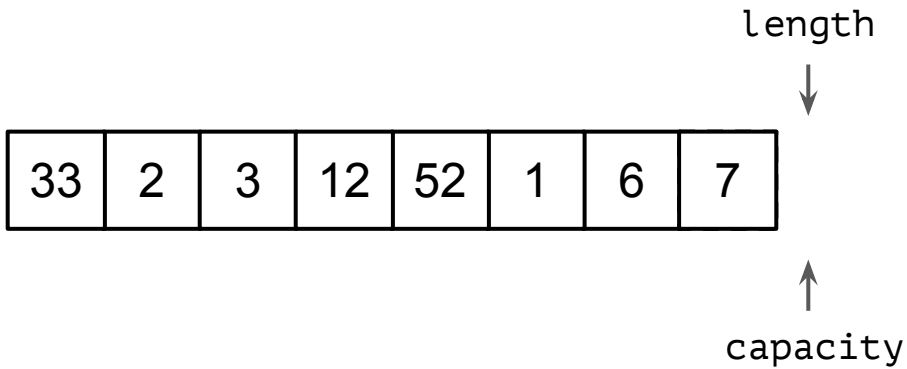

```
vector = Vector(33, 2, 3, 12)
vector.add(52)
vector.add(1)
vector.add(6)
vector.add(7)
vector.add(11)
```

```
def add(self, value: int):
    if self.length < self.capacity:
        self.data[self.length] = value
        self.length += 1
```

Что делать?

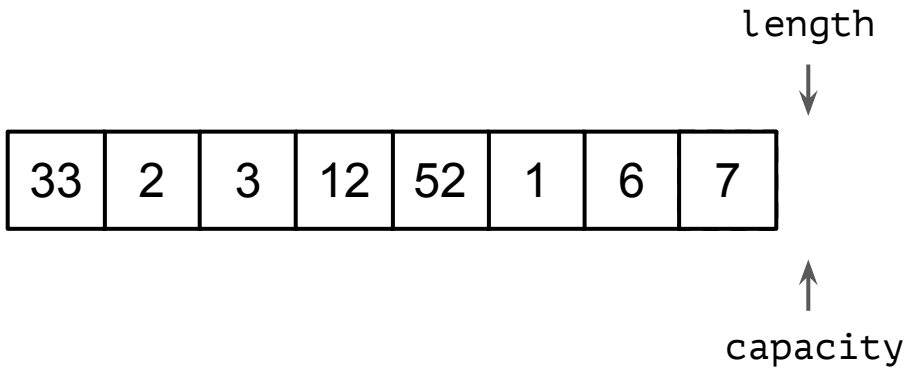
Реаллоцировать!

Динамический массив




```
def add(self, value: int):
    if self.length < self.capacity:
        self.data[self.length] = value
        self.length += 1
```

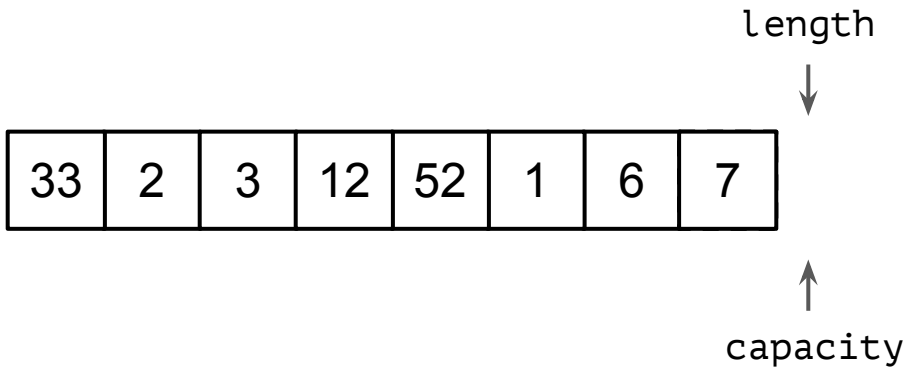
Динамический массив




```
def add(self, value: int):
    if self.length == self.capacity:
        ...
        ...

    self.data[self.length] = value
    self.length += 1
```

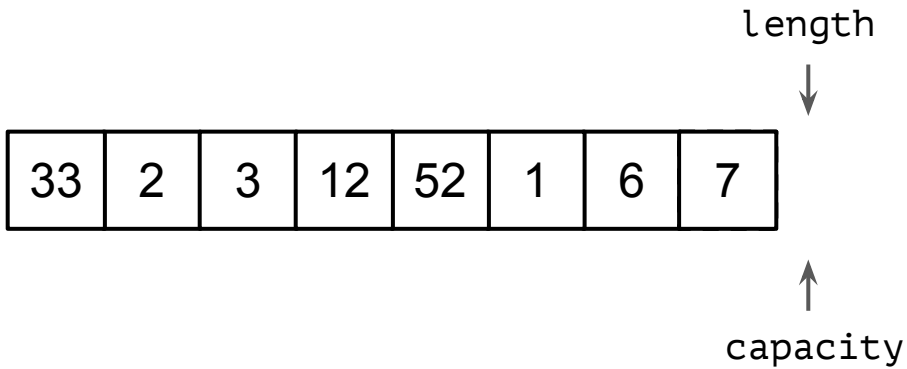
Динамический массив




```
def add(self, value: int):
    if self.length == self.capacity:
        self.capacity = ???
        self.data = reallocate(self.data, ???)

    self.data[self.length] = value
    self.length += 1
```

Динамический массив

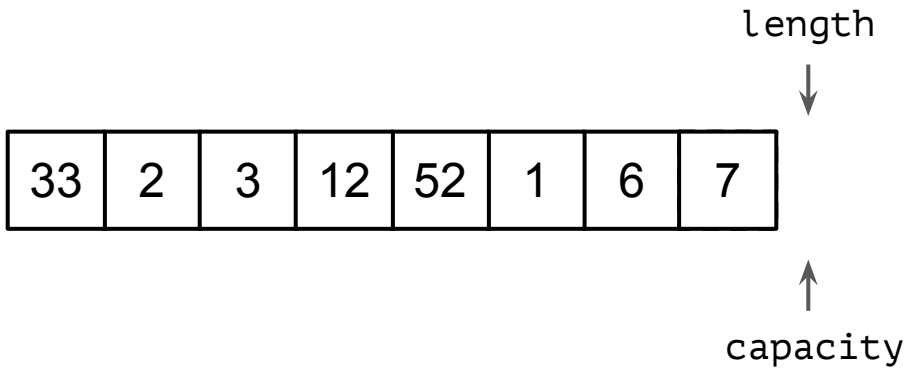


А насколько увеличивать размер?

```
def add(self, value: int):
    if self.length == self.capacity:
        self.capacity = ???
        self.data = reallocate(self.data, ???)

    self.data[self.length] = value
    self.length += 1
```

Динамический массив

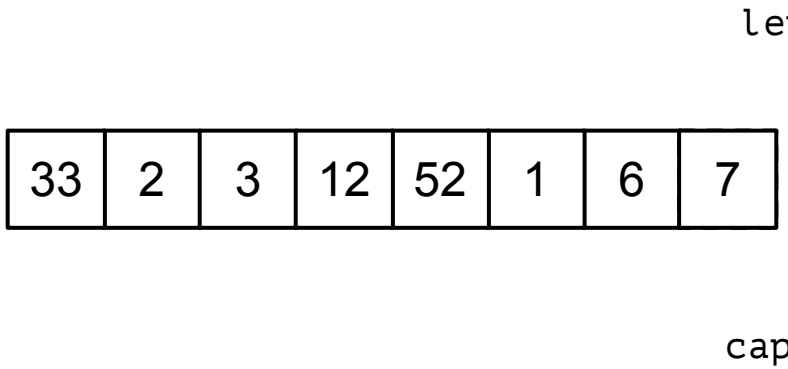


А насколько увеличивать размер?

Допустим, на 1 элемент

```
def add(self, value: int):
    if self.length == self.capacity:
        self.capacity += 1
        self.data = reallocate(self.data, self.capacity)

    self.data[self.length] = value
    self.length += 1
```



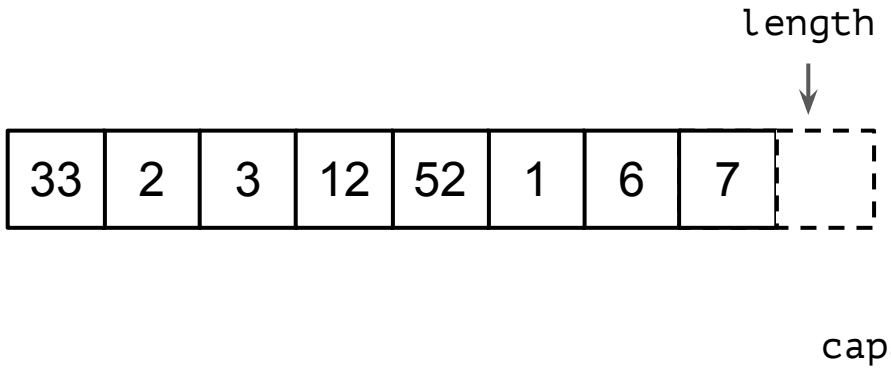
А насколько увеличивать размер?

Допустим, на 1 элемент

```
def add(self, value: int):
    if self.length == self.capacity:
        self.capacity += 1
        self.data = reallocate(self.data, self.capacity)

    self.data[self.length] = value
    self.length += 1

vector.add(4)
```



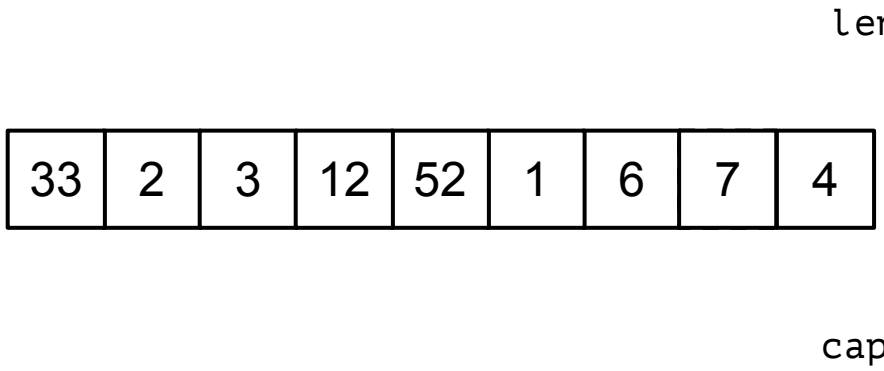
А насколько увеличивать размер?

Допустим, на 1 элемент

```
def add(self, value: int):
    if self.length == self.capacity:
        self.capacity += 1
        self.data = reallocate(self.data, self.capacity)

    self.data[self.length] = value
    self.length += 1

vector.add(4)
```



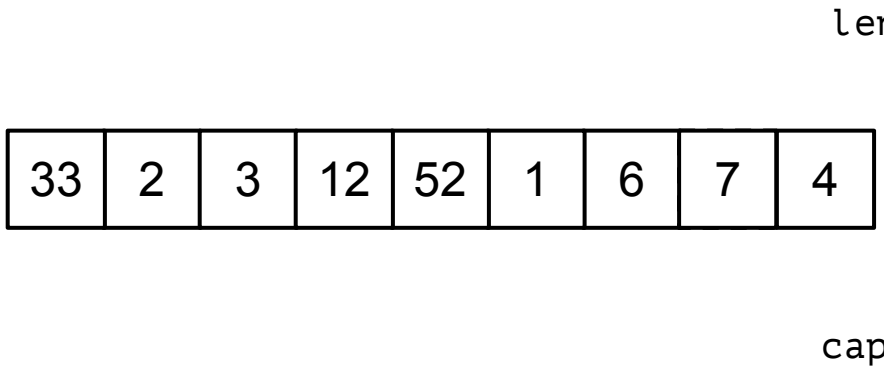
А насколько увеличивать размер?

Допустим, на 1 элемент

```
def add(self, value: int):
    if self.length == self.capacity:
        self.capacity += 1
        self.data = reallocate(self.data, self.capacity)

    self.data[self.length] = value
    self.length += 1

vector.add(4)
```



А насколько увеличивать размер?

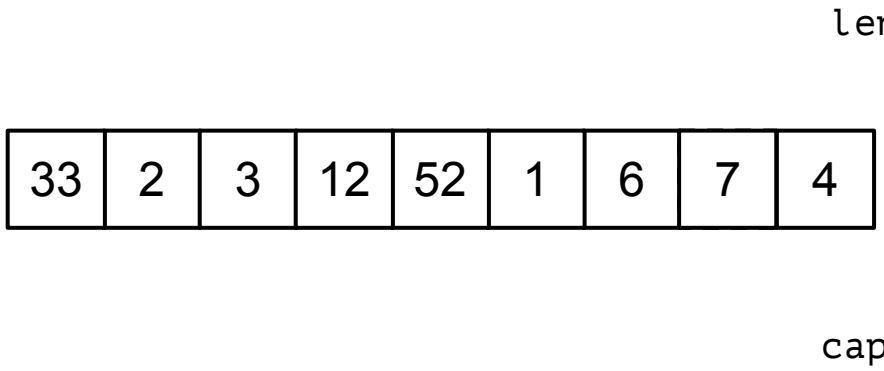
Допустим, на 1 элемент

```
def add(self, value: int):
    if self.length == self.capacity:
        self.capacity += 1
        self.data = reallocate(self.data, self.capacity)

    self.data[self.length] = value
    self.length += 1
```

```
vector.add(4)
```

reallocate - дорогая операция,
подразумевающая копирование всех элементов



А насколько увеличивать размер?

Допустим, на 1 элемент

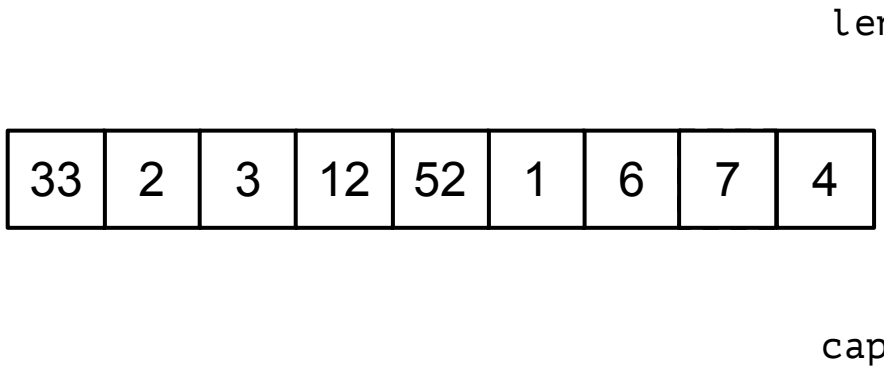
```
def add(self, value: int):
    if self.length == self.capacity:
        self.capacity += 1
        self.data = reallocate(self.data, self.capacity)

    self.data[self.length] = value
    self.length += 1
```

Тогда сложность функции add?

```
vector.add(4)
```

reallocate - дорогая операция, подразумевающая копирование всех элементов



А насколько увеличивать размер?

Допустим, на 1 элемент

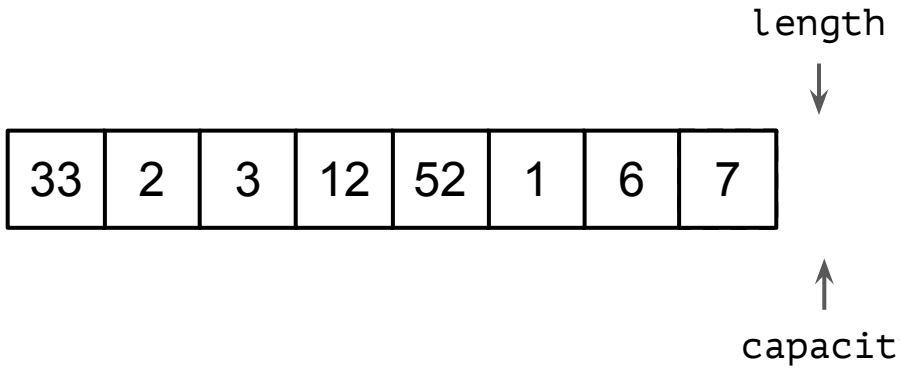
```
def add(self, value: int):
    if self.length == self.capacity:
        self.capacity += 1
        self.data = reallocate(self.data, self.capacity)

    self.data[self.length] = value
    self.length += 1
```

Тогда сложность функции add? O(n)!

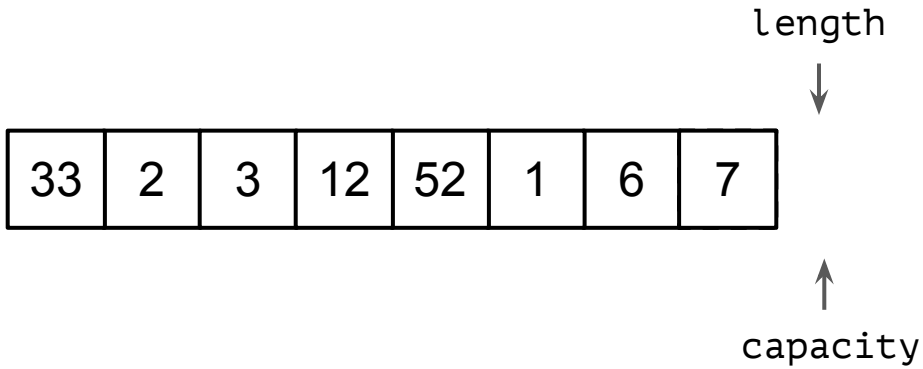
```
vector.add(4)
```

reallocate - дорогая операция, подразумевающая копирование всех элементов



Но при анализе сложности запросов к структурам данных часто не останавливаются на сложности в худшем (или даже в среднем).

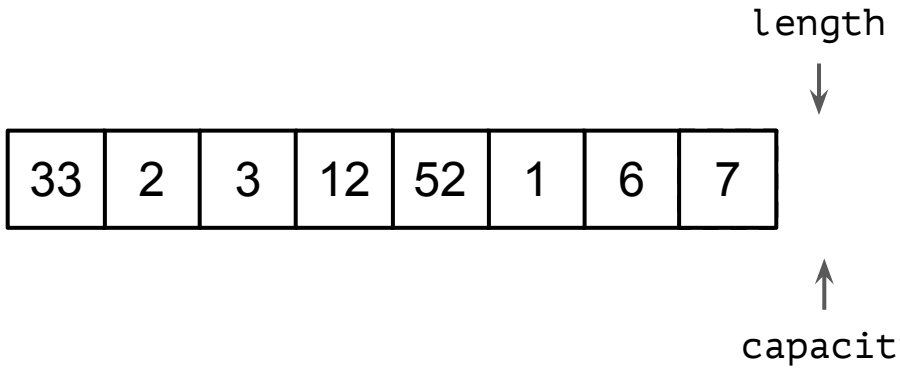
А проводят амортизационный анализ и вычисляют амортизированное (учётное) время работы запроса.



Пусть есть последовательность запросов к динамическому массиву:

```
vector.add(33)
vector.add(2)
...
vector.add(6)
vector.add(7)
```

} N штук

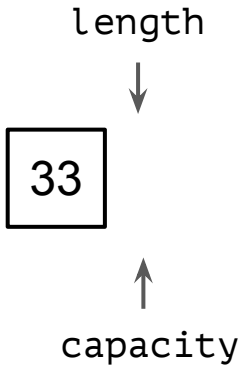


Пусть есть последовательность запросов к динамическому массиву:

```
vector.add(33)
vector.add(2)
...
vector.add(6)
vector.add(7)
```

} N штук

Оцените общее время работы последовательности запросов?



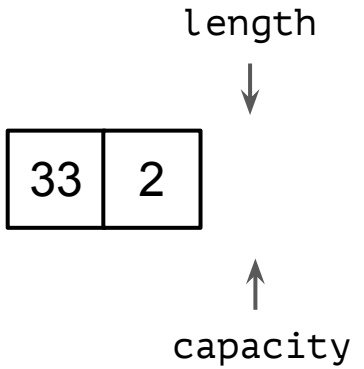
Потратили k операций

Пусть есть последовательность запросов к динамическому массиву:

```
vector.add(33)
vector.add(2)
...
vector.add(6)
vector.add(7)
```

} N штук

Оцените общее время работы последовательности запросов?



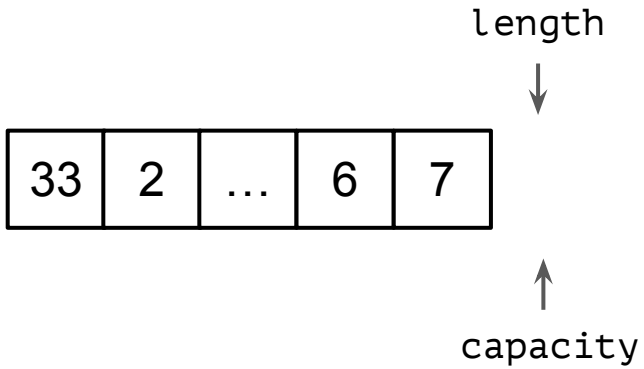
Потратили k + 2*k операций

Пусть есть последовательность запросов к динамическому массиву:

```
vector.add(33)
vector.add(2)
...
vector.add(6)
vector.add(7)
```

} N штук

Оцените общее время работы последовательности запросов?



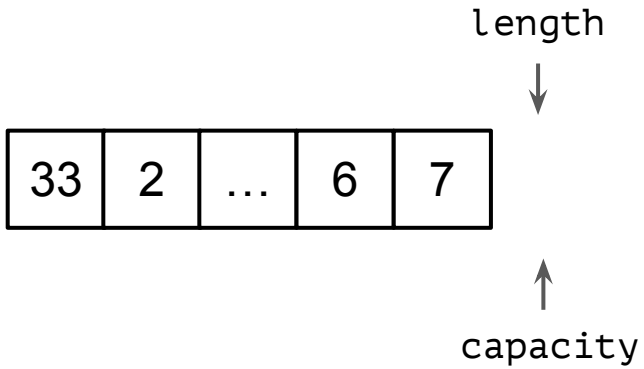
Потратили k + 2*k + … + k*(N - 1) + k*N операций

Пусть есть последовательность запросов к динамическому массиву:

```
vector.add(33)
vector.add(2)
...
vector.add(6)
vector.add(7)
```

} N штук

Оцените общее время работы последовательности запросов?



Потратили O(N^2) операций

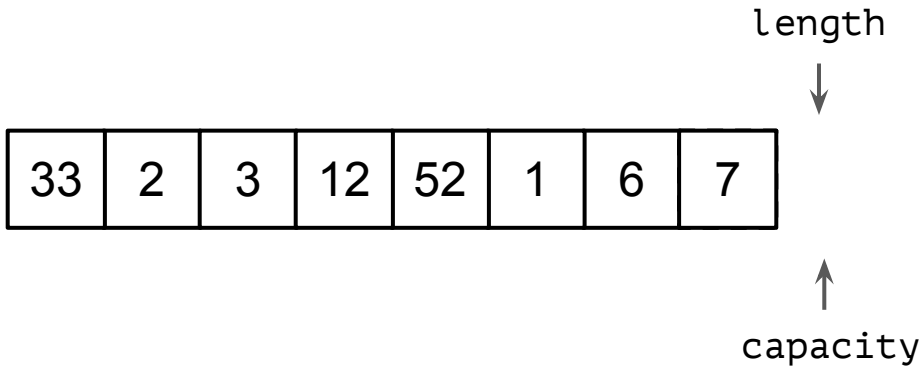
Пусть есть последовательность запросов к динамическому массиву:

```
vector.add(33)
vector.add(2)
...
vector.add(6)
vector.add(7)
```

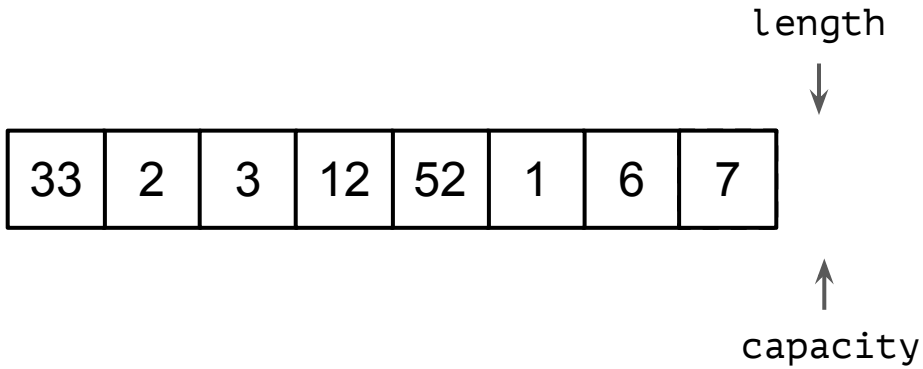
} N штук

Оцените общее время работы последовательности запросов?

O(N^2)

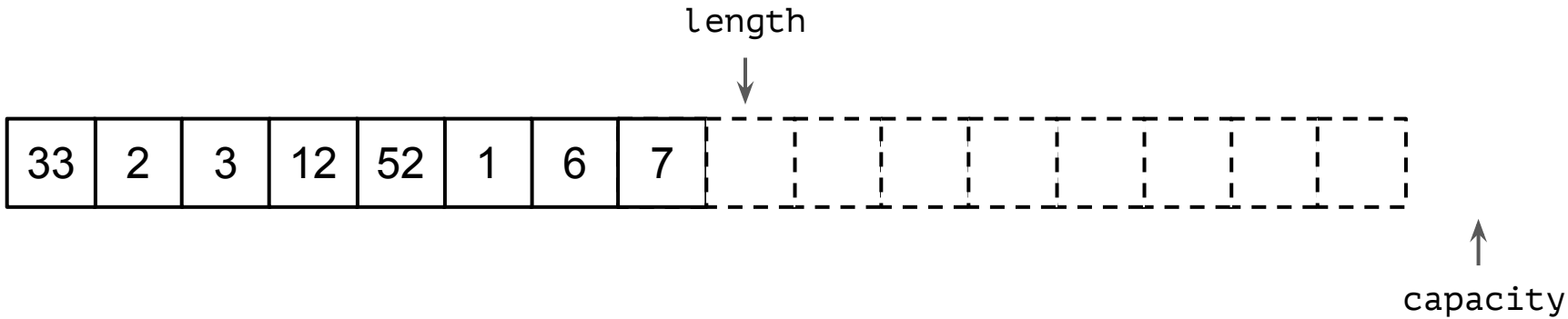


Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.



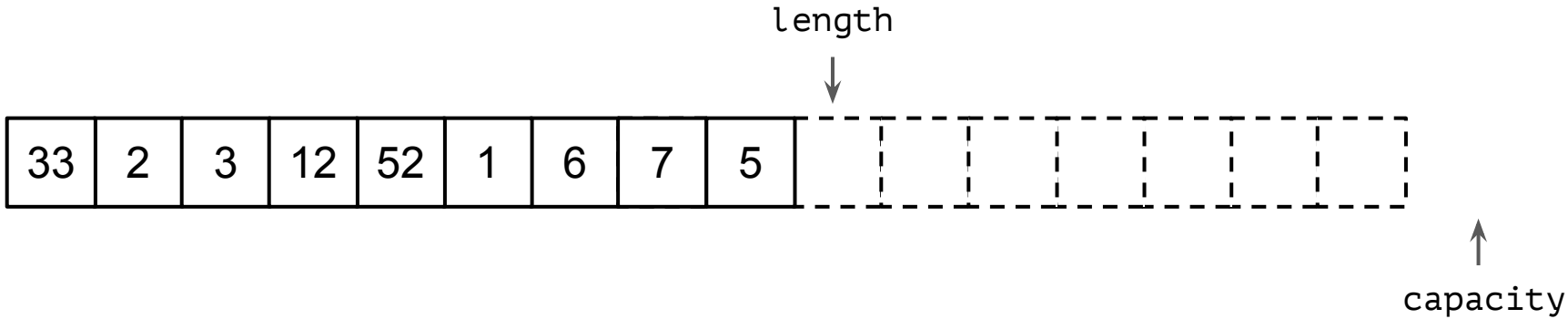
Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.

```
vector.add(5)
```



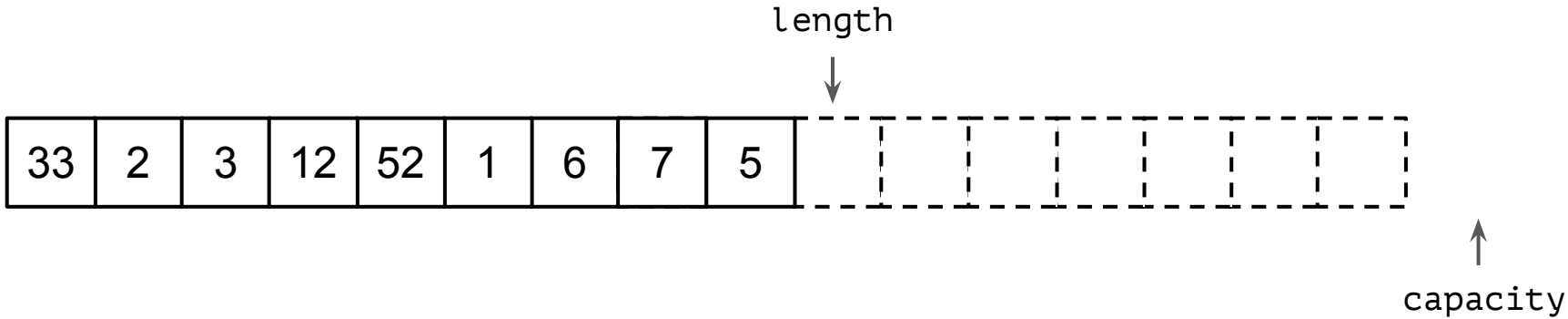
Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.

```
vector.add(5)
```



Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.

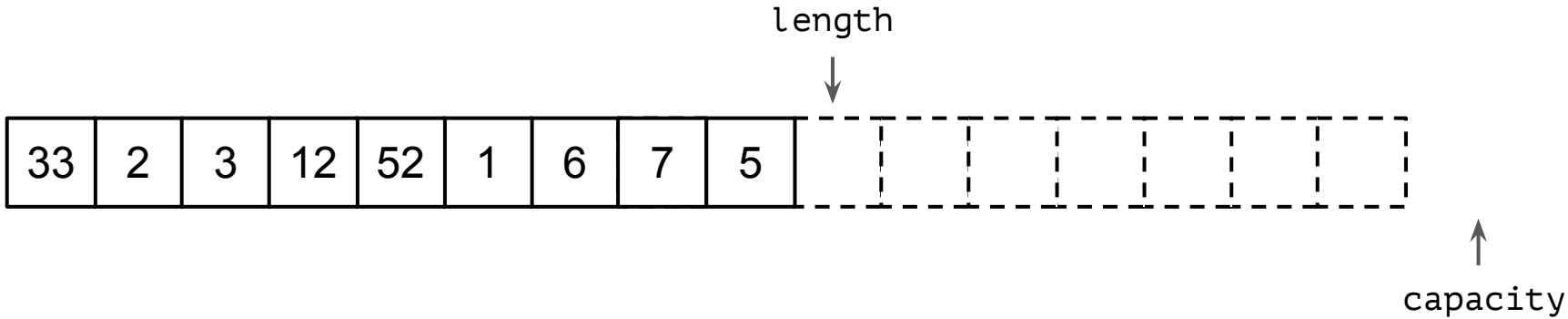
```
vector.add(5)
```



Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.

```
vector.add(5)
```

Очевидно, что вот такой запрос дорогой - мы тратим k*N операций, где N - количество уже элементов в массиве.

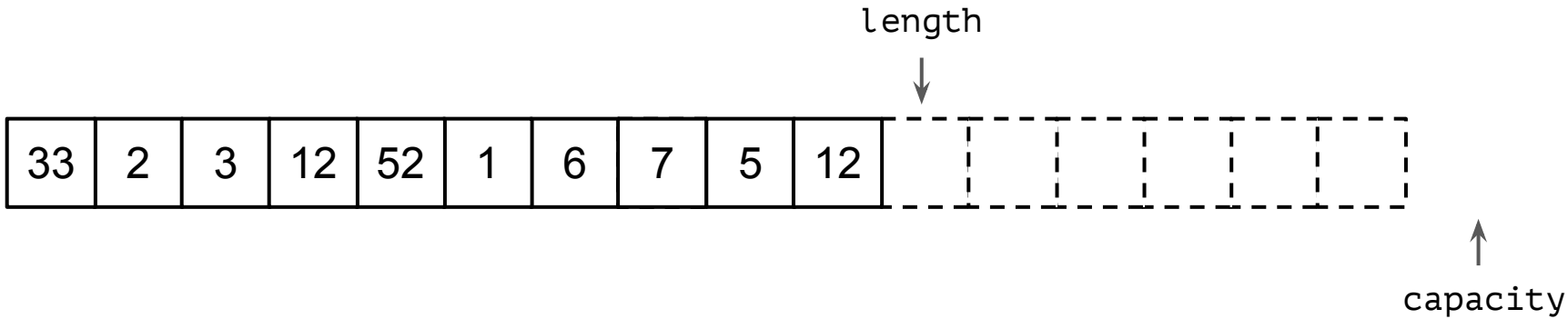


Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.

```
vector.add(5)
```

Очевидно, что вот такой запрос дорогой - мы тратим k*N операций, где N - количество уже элементов в массиве.

Но что можно сказать про следующие операции?

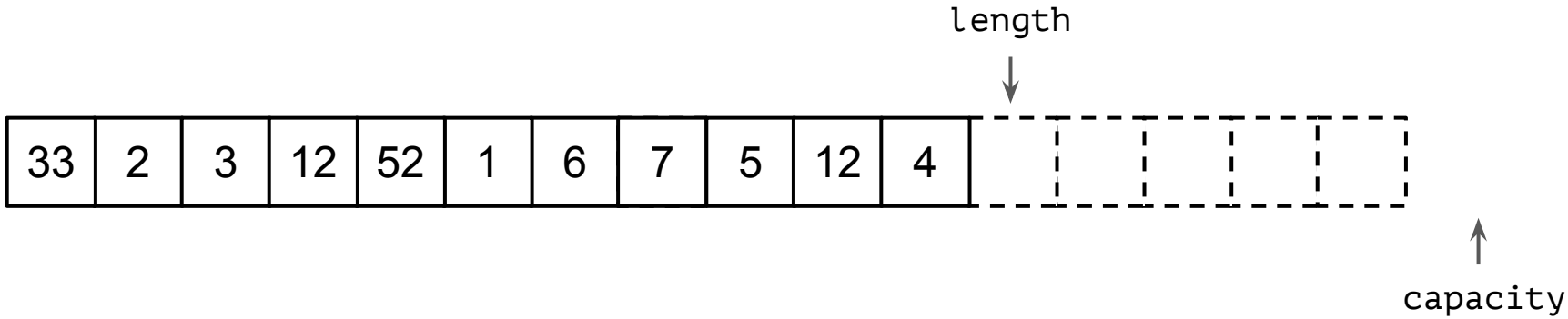


Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.

```
vector.add(5) => vector.add(12)
```

Очевидно, что вот такой запрос дорогой - мы тратим k*N операций, где N - количество уже элементов в массиве.

Но что можно сказать про следующие операции?

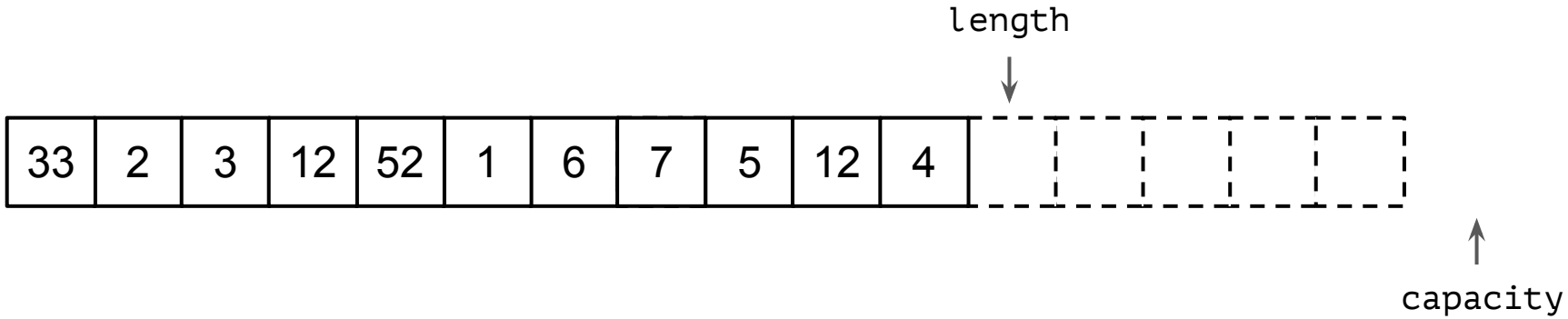


Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.

```
vector.add(5) => vector.add(12) => vector.add(4)
```

Очевидно, что вот такой запрос дорогой - мы тратим k*N операций, где N - количество уже элементов в массиве.

Но что можно сказать про следующие операции?

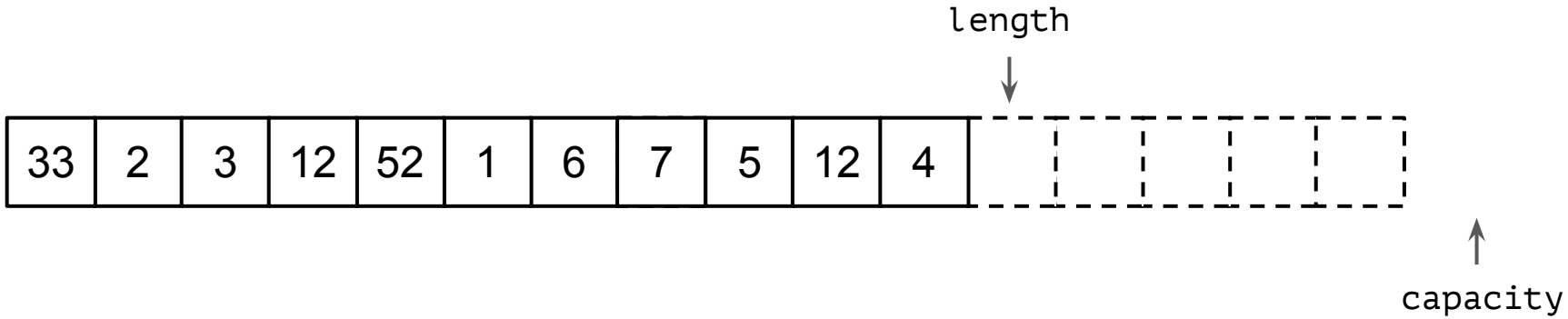


Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.

```
vector.add(5) => vector.add(12) => vector.add(4)
```

Очевидно, что вот такой запрос дорогой - мы тратим k*N операций, где N - количество уже элементов в массиве.

Но что можно сказать про следующие операции? Они работают за k



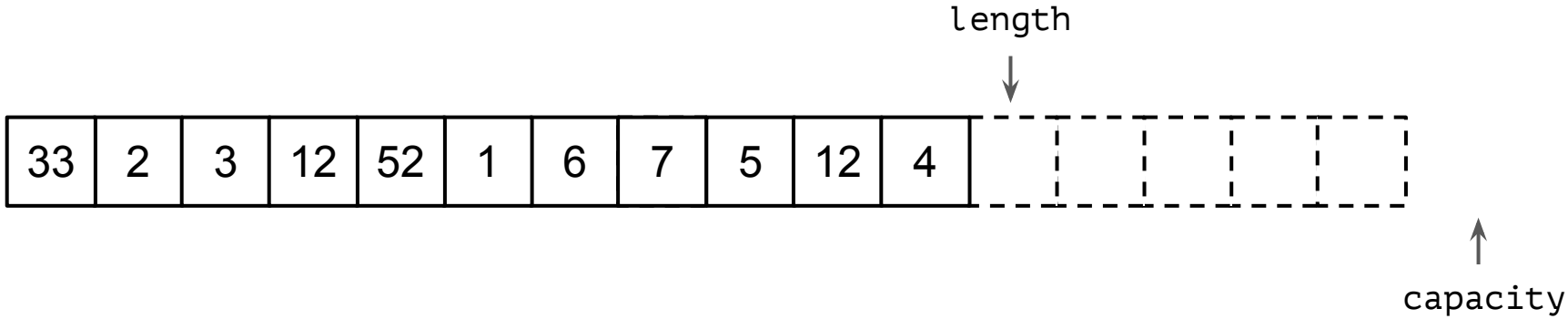
Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.

Пока снова не придет расплата

vector.add(5) => vector.add(12) => vector.add(4)

Очевидно, что вот такой запрос дорогой - мы тратим k*N операций, где N - количество уже элементов в массиве.

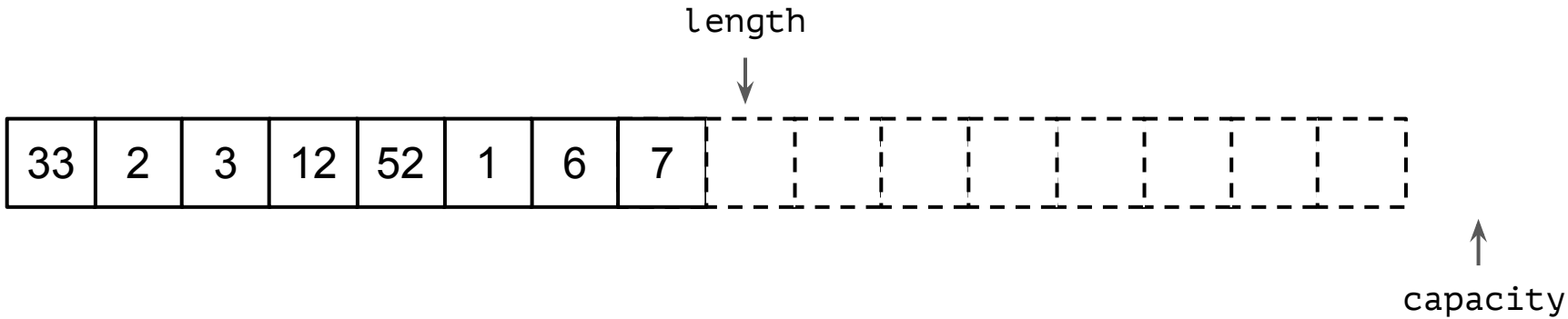
Но что можно сказать про следующие операции? Они работают за k




```
vector.add(33)
vector.add(2)
...
vector.add(6)
vector.add(7)
```

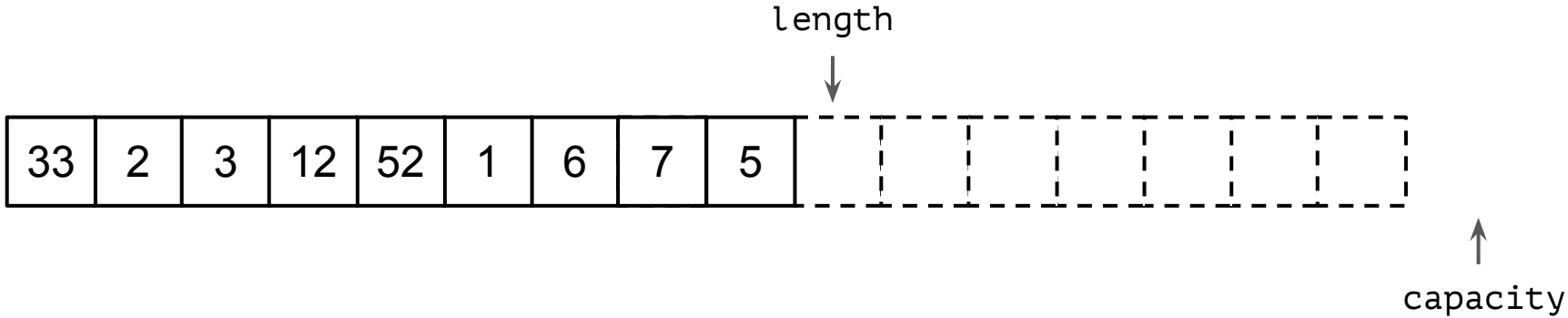
} N штук

Оцените общее время работы последовательности запросов?



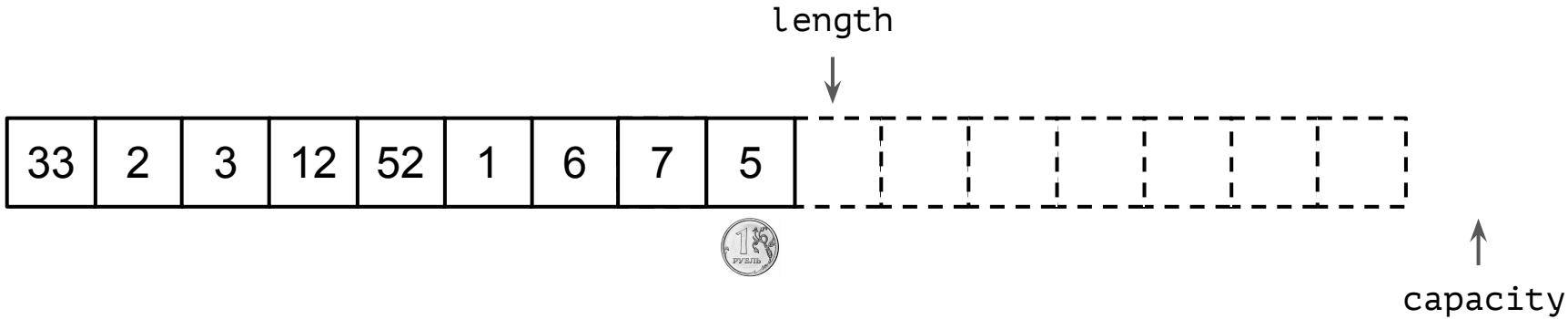
Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.

```
vector.add(5)
```



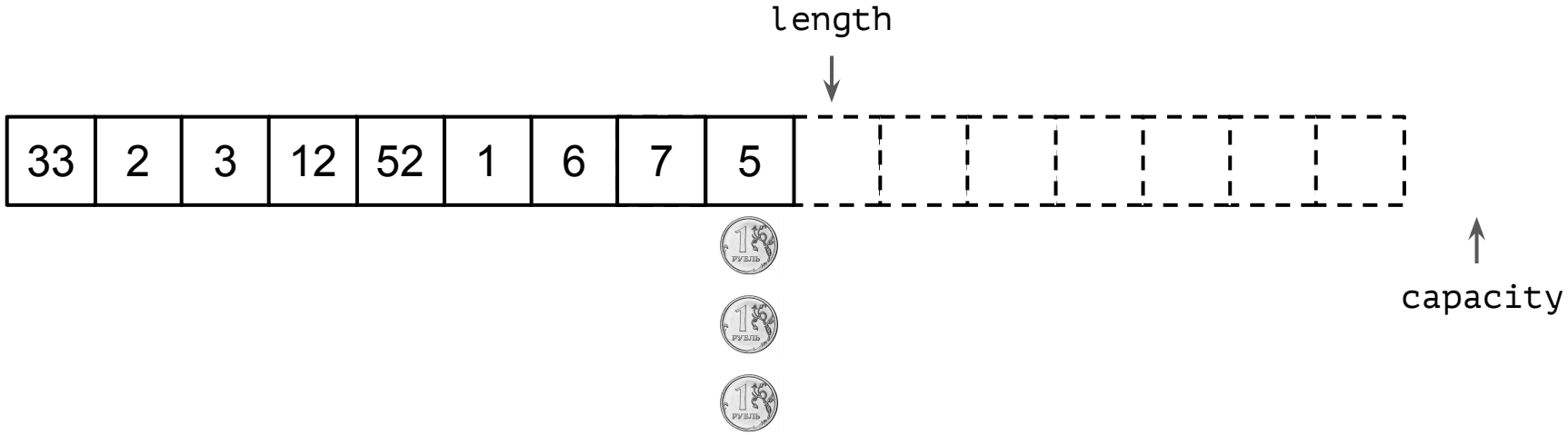
Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.

vector.add(5) - потратили k операций



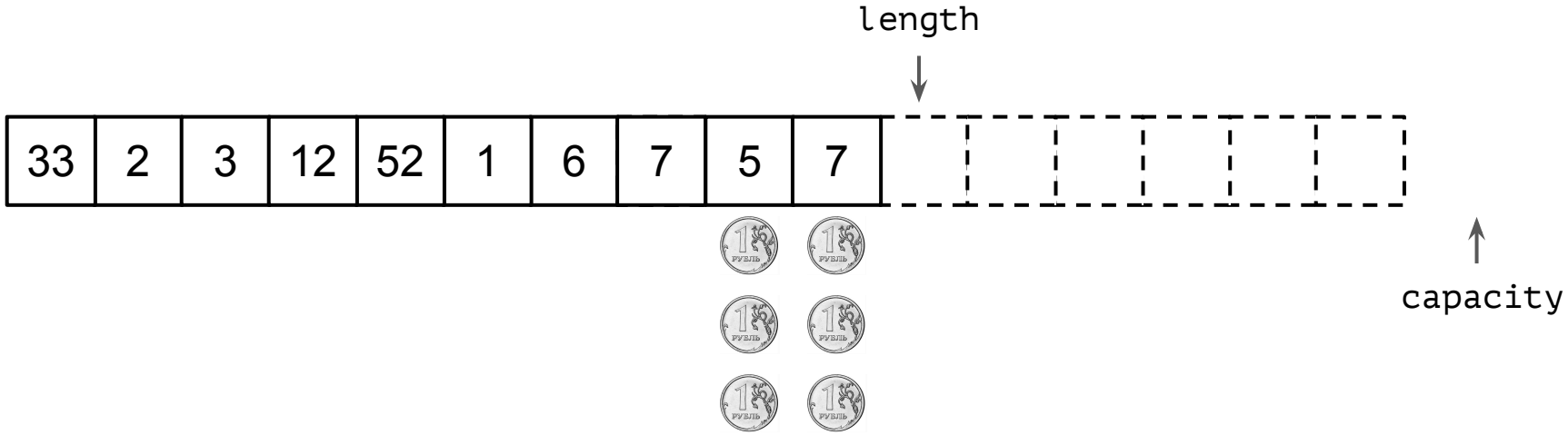
Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.

vector.add(5) - потратили k операций (пусть это 1 рубль)



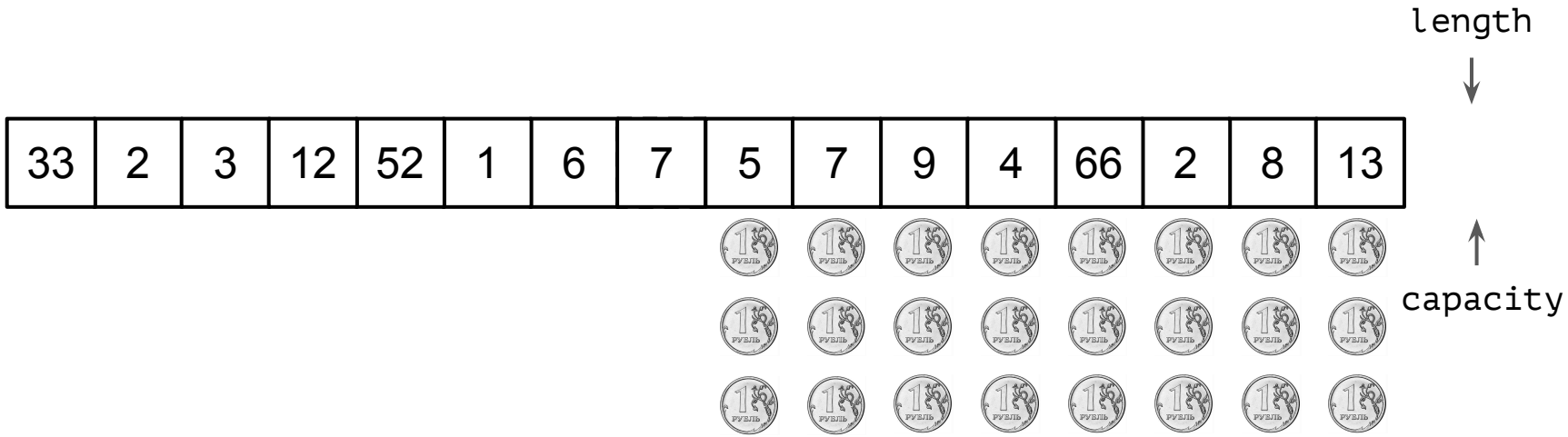
Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.

vector.add(5) - потратили k операций (пусть это 1 рубль)
а пусть потратили 3*k (нам жалко что ли)



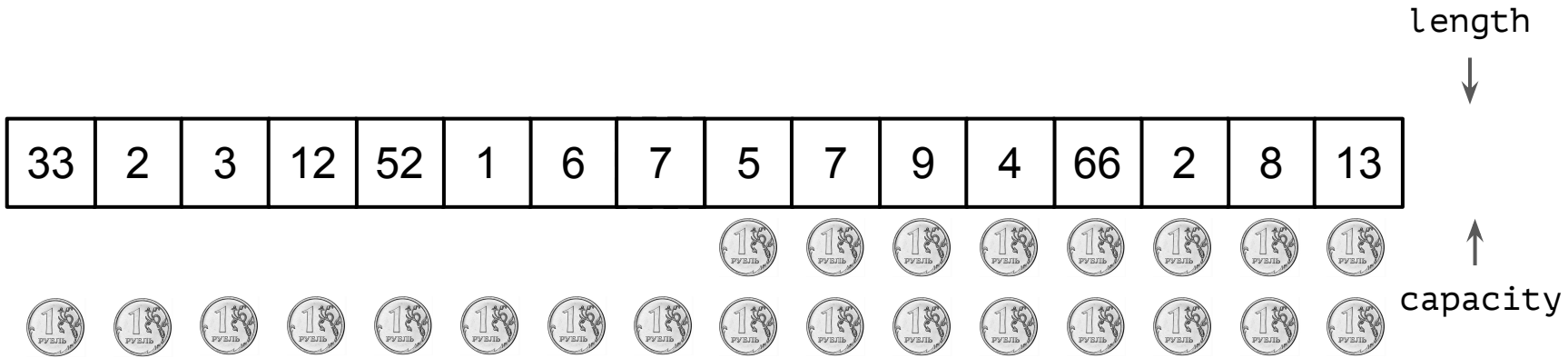
Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.

vector.add(5) - потратили k операций (пусть это 1 рубль)
а пусть потратили 3*k (нам жалко что ли)
vector.add(7) - опять



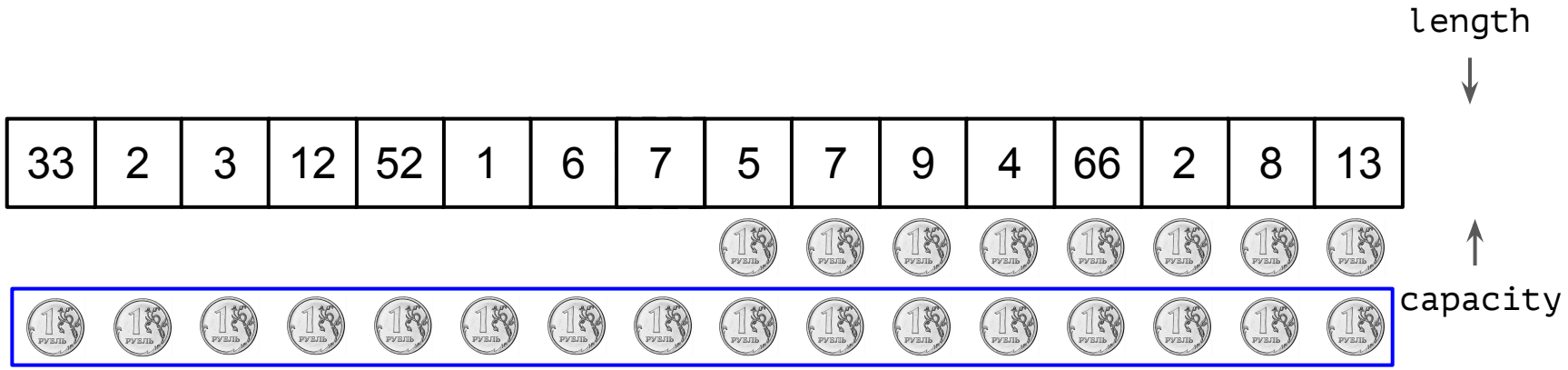
Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.

```
vector.add(5) - потратили k операций (пусть это 1 рубль)
                а пусть потратили 3*k (нам жалко что ли)
vector.add(7) - опять
…
vector.add(13)- опять
```



Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.

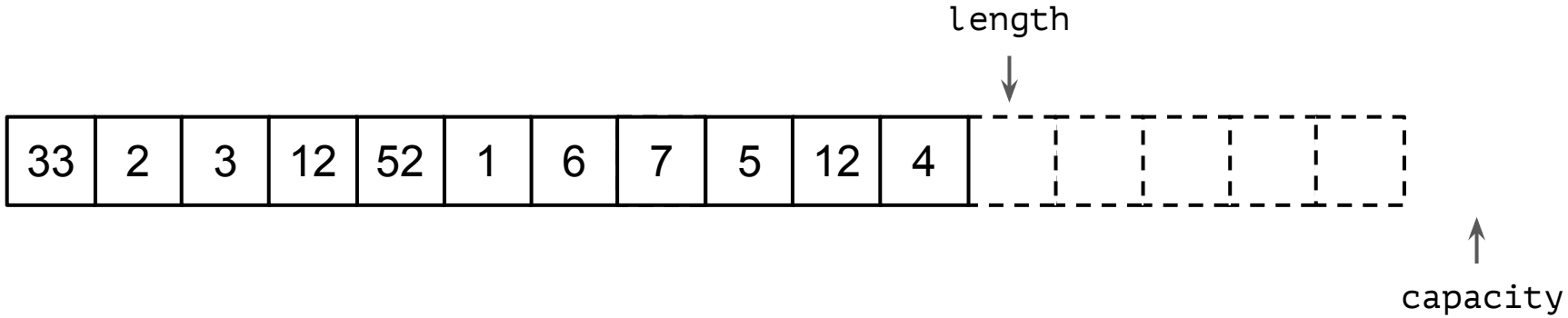
```
vector.add(5) - потратили k операций (пусть это 1 рубль)
                а пусть потратили 3*k (нам жалко что ли)
vector.add(7) - опять
…
vector.add(13)- опять
```



Эти деньги пойдут на следующий reallocate (тут как раз стоимость N операций)

Альтернативный вариант: давайте каждый раз (когда память кончается) увеличивать размер **в два** раза.

```
vector.add(5) - потратили k операций (пусть это 1 рубль)
                а пусть потратили 3*k (нам жалко что ли)
vector.add(7) - опять
…
vector.add(13)- опять
```

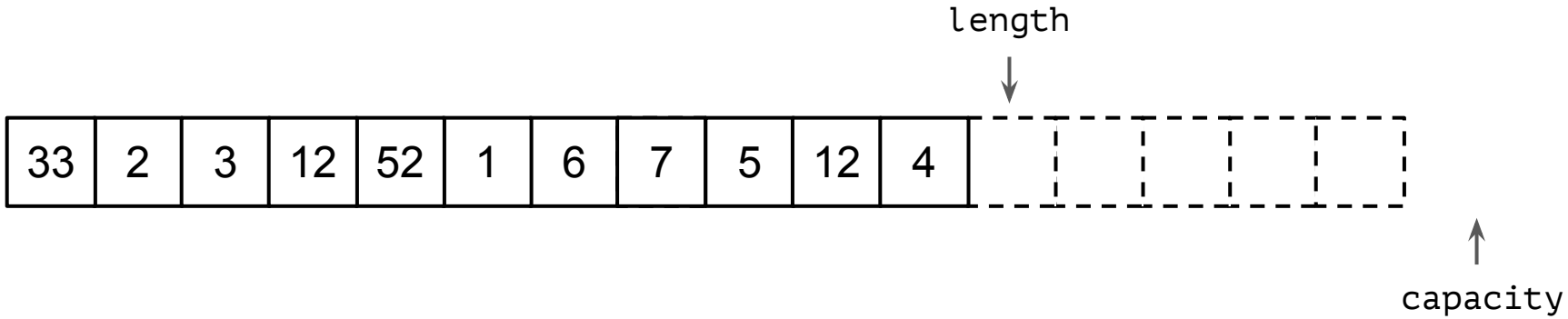



```
vector.add(33)
vector.add(2)
...
vector.add(6)
vector.add(7)
```

} N штук

Оцените общее время работы последовательности запросов?

Все время работы - O(N), т.к. [можно считать, что] каждый запрос был за O(1)



Тогда говорят, что амортизационное (учётное) время работы запроса add — O*(1)

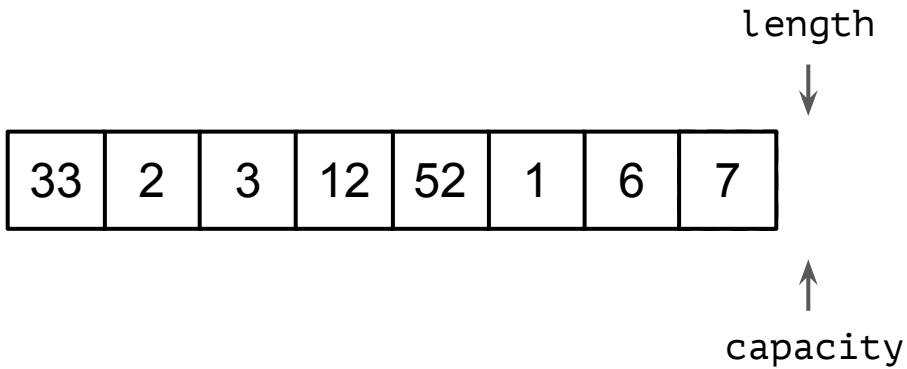
```
vector.add(33)
vector.add(2)
...
vector.add(6)
vector.add(7)
```

} N штук

Оцените общее время работы последовательности запросов?

Все время работы - O(N), т.к. [можно считать, что] каждый запрос был за O(1)

Динамический массив



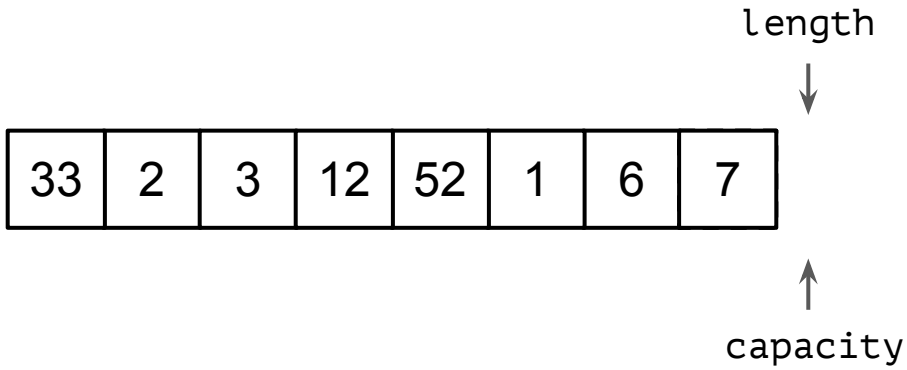
А насколько увеличивать размер?

Допустим, на 1 элемент

```
def add(self, value: int):
    if self.length == self.capacity:
        self.capacity += 1
        self.data = reallocate(self.data, self.capacity)

    self.data[self.length] = value
    self.length += 1
```

Динамический массив



А насколько увеличивать размер?

в 2 раза, чтобы получить O*(1)

```
def add(self, value: int):
    if self.length == self.capacity:
        self.capacity *= 2
        self.data = reallocate(self.data, self.capacity)

    self.data[self.length] = value
    self.length += 1
```

Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

1. Доступ к i-ому элементу

---

2. Добавление элемента в "конец"
3. Удаление последнего элемента

Как реализовывать будем? Динамический массив!

Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

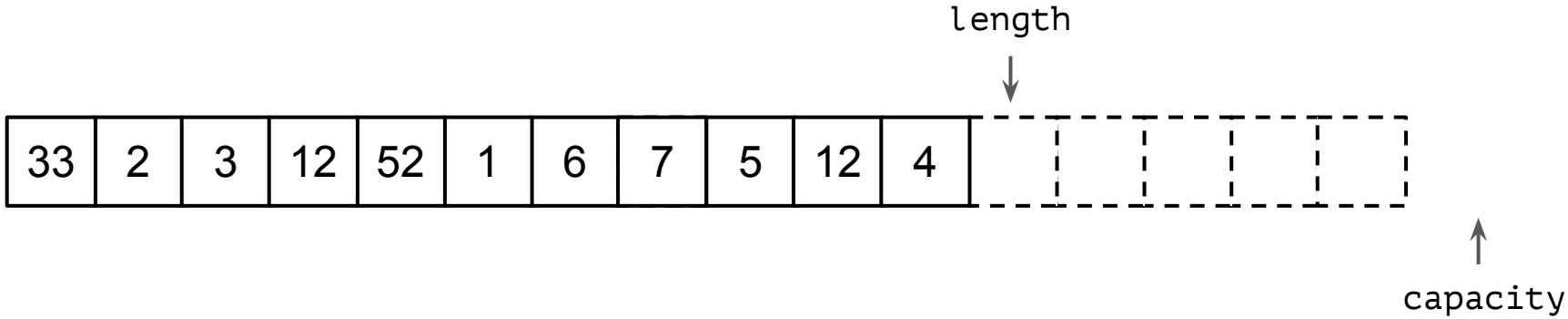
Которое бы поддерживало следующие запросы:

✓ 1. Доступ к i-ому элементу $O(1)$

✓ 2. Добавление элемента в "конец" $O^*(1)$

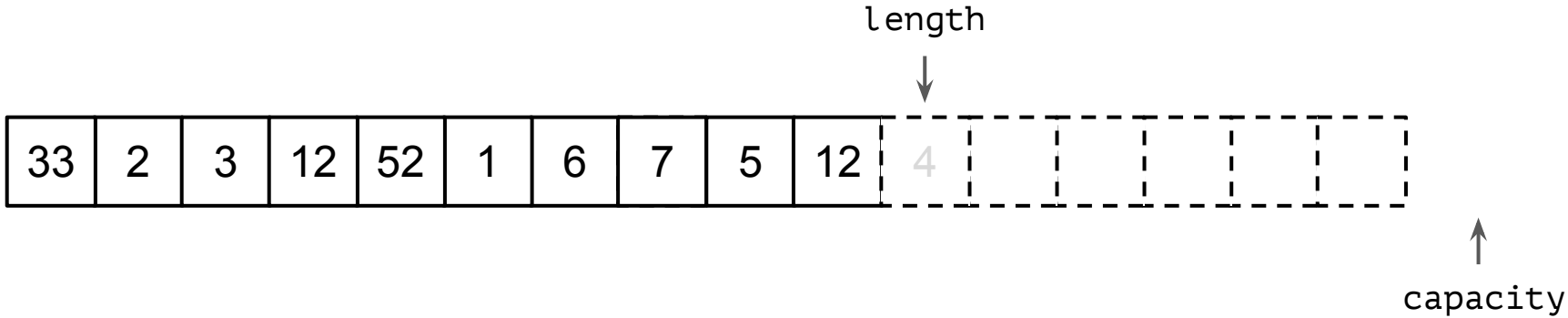
3. Удаление последнего элемента

Как реализовывать будем? Динамический массив!



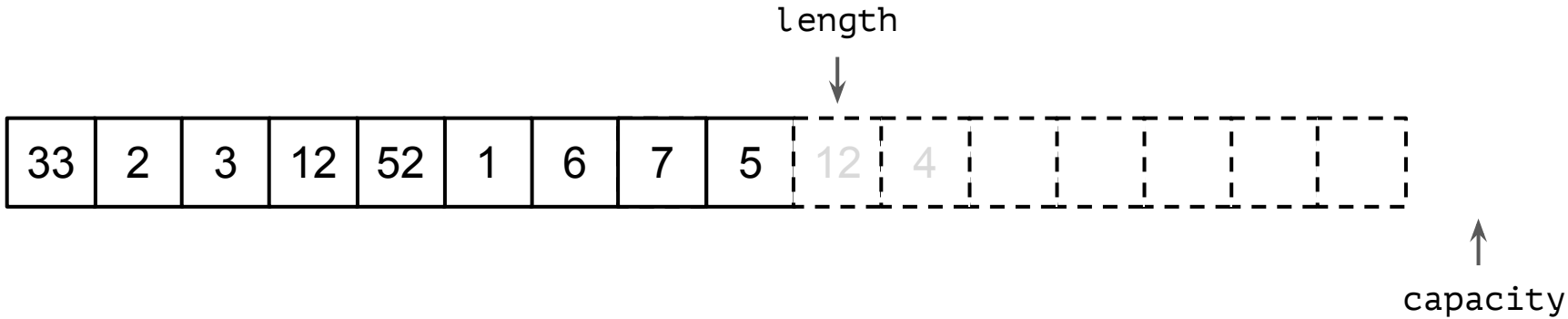
Как удалять последний элемент?

```
vector.remove()
```



Как удалять последний элемент?

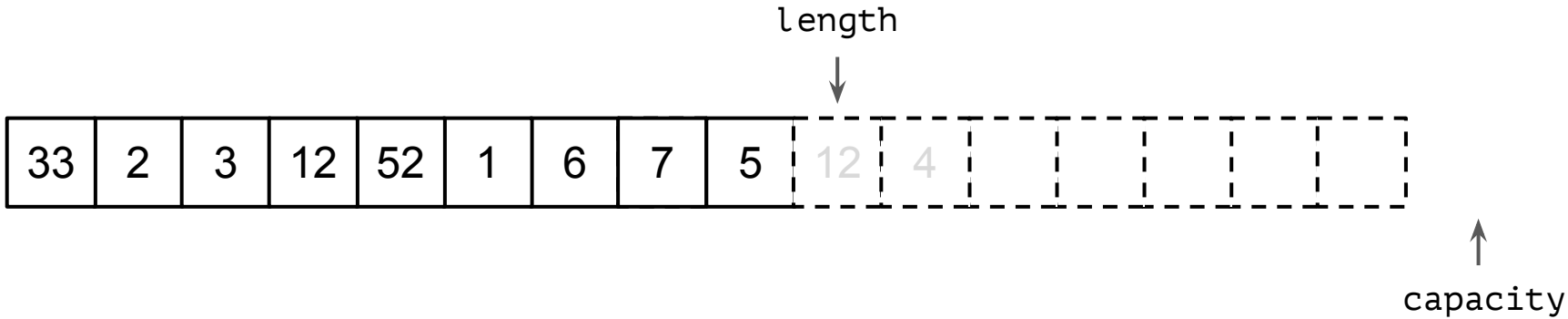
vector.remove()



Как удалять последний элемент?

Но если вы хоть немного экономите память, то вы захотите "сдуть" (shrink) массив.

```
vector.remove()
vector.remove()
```

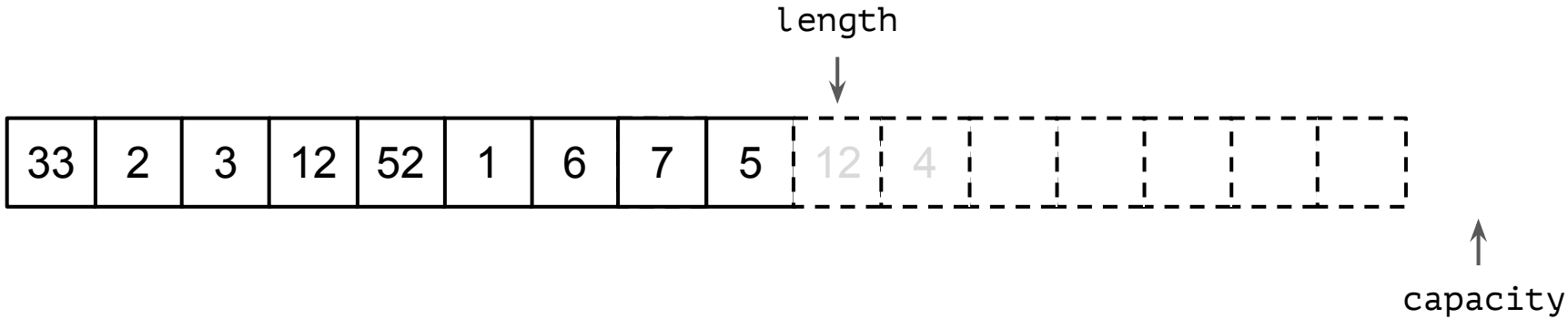


Как удалять последний элемент?

Но если вы хоть немного экономите память, то вы захотите "сдуть" (shrink) массив.

Когда?

vector.remove()
vector.remove()

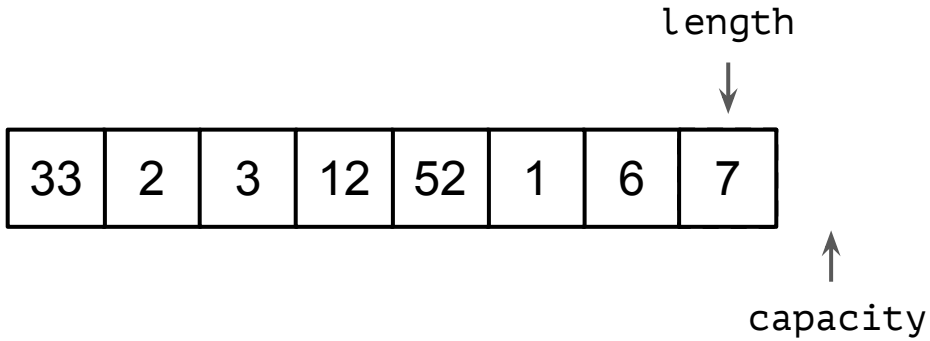


Как удалять последний элемент?

Но если вы хоть немного экономите память, то вы захотите "сдуть" (shrink) массив.

Когда? Наивный подход - когда дошли до середины.

```
vector.remove()
vector.remove()
```

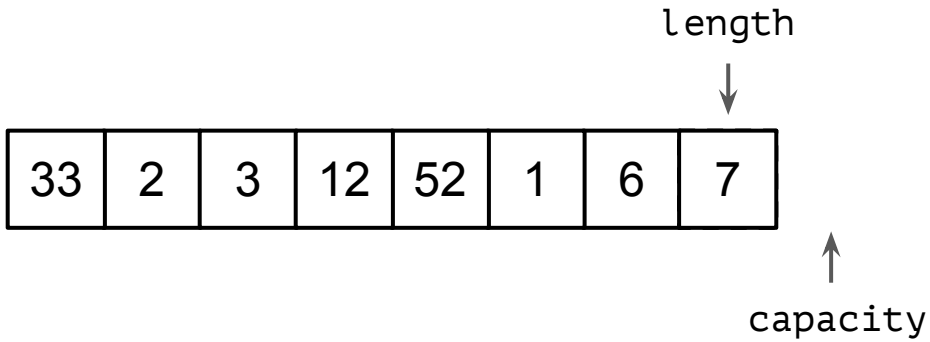


Как удалять последний элемент?

Но если вы хоть немного экономите память, то вы захотите "сдуть" (shrink) массив.

Когда? Наивный подход - когда дошли до середины.

```
vector.remove()
vector.remove()
...
vector.remove()
```

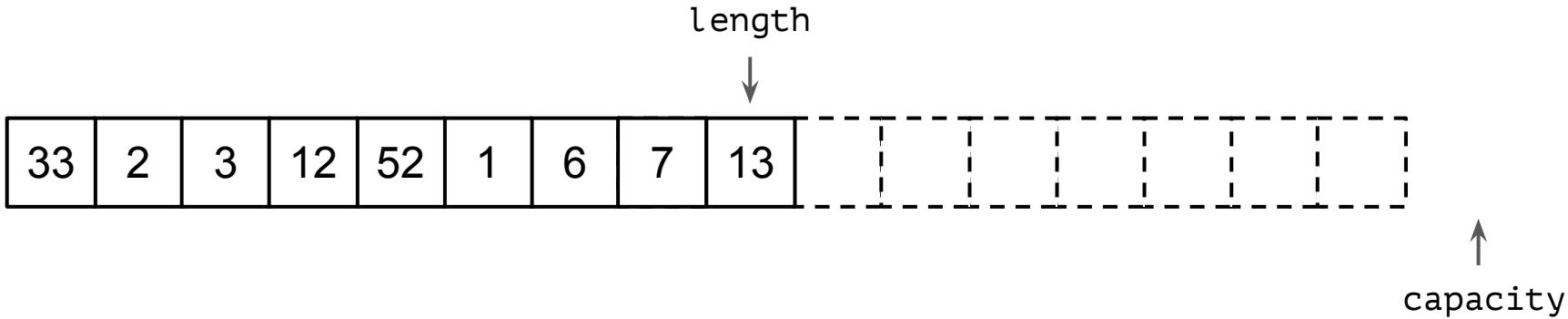


Как удалять последний элемент?

Но если вы хоть немного экономите память, то вы захотите "сдуть" (shrink) массив.

Когда? Наивный подход - когда дошли до середины.
Но теперь представьте, что дальнейшая последовательность такая:

```
vector.add(13)
vector.remove()
vector.add(13)
vector.remove()
```

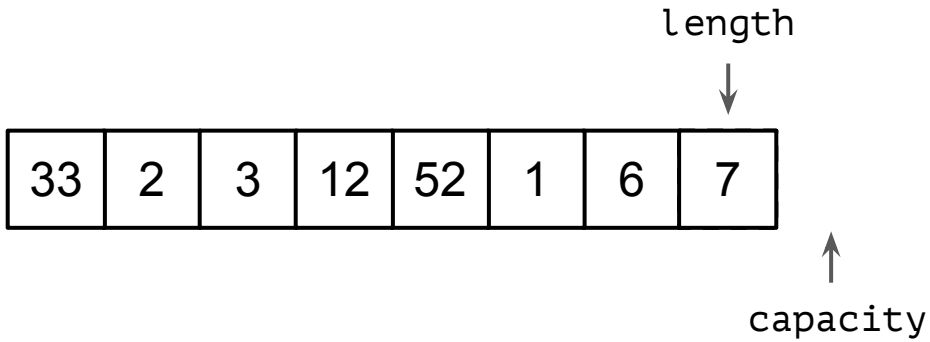


Как удалять последний элемент?

Но если вы хоть немного экономите память, то вы захотите "сдуть" (shrink) массив.

Когда? Наивный подход - когда дошли до середины.
Но теперь представьте, что дальнейшая последовательность такая:

```
vector.add(13)
vector.remove()
vector.add(13)
vector.remove()
```

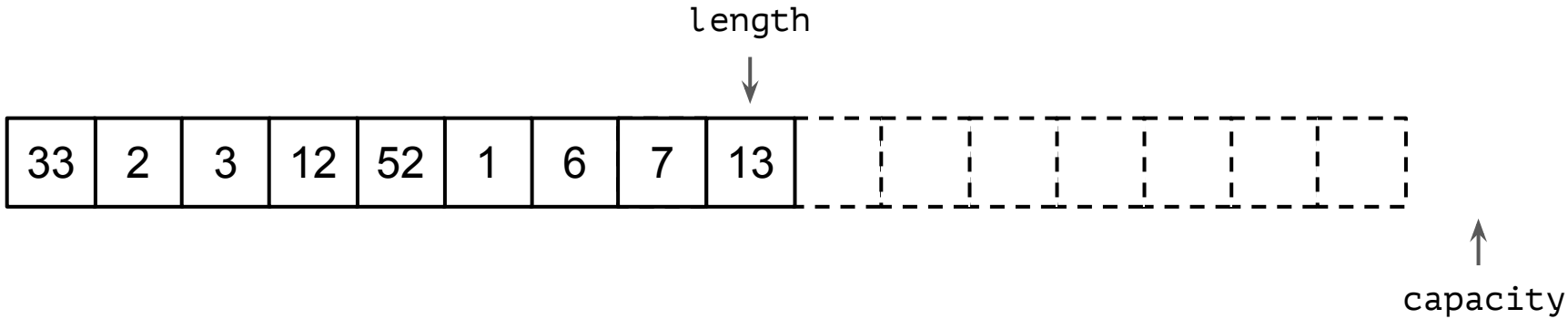


Как удалять последний элемент?

Но если вы хоть немного экономите память, то вы захотите "сдуть" (shrink) массив.

Когда? Наивный подход - когда дошли до середины.
Но теперь представьте, что дальнейшая последовательность такая:

```
vector.add(13)
vector.remove()
vector.add(13)
vector.remove()
```

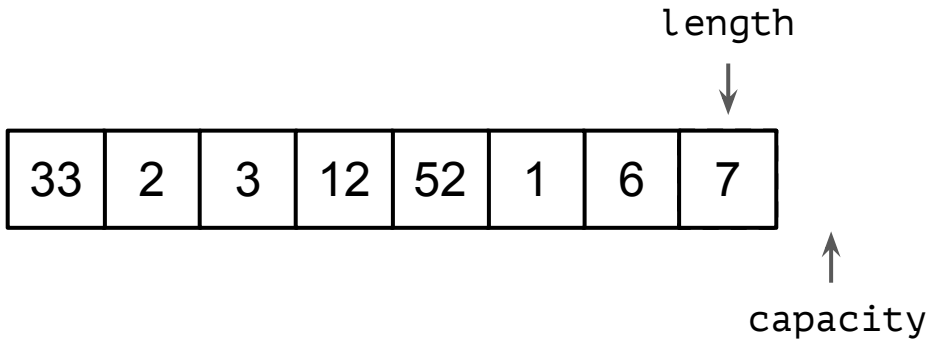


Как удалять последний элемент?

Но если вы хоть немного экономите память, то вы захотите "сдуть" (shrink) массив.

Когда? Наивный подход - когда дошли до середины.
Но теперь представьте, что дальнейшая последовательность такая:

```
vector.add(13)
vector.remove()
vector.add(13)
vector.remove()
```



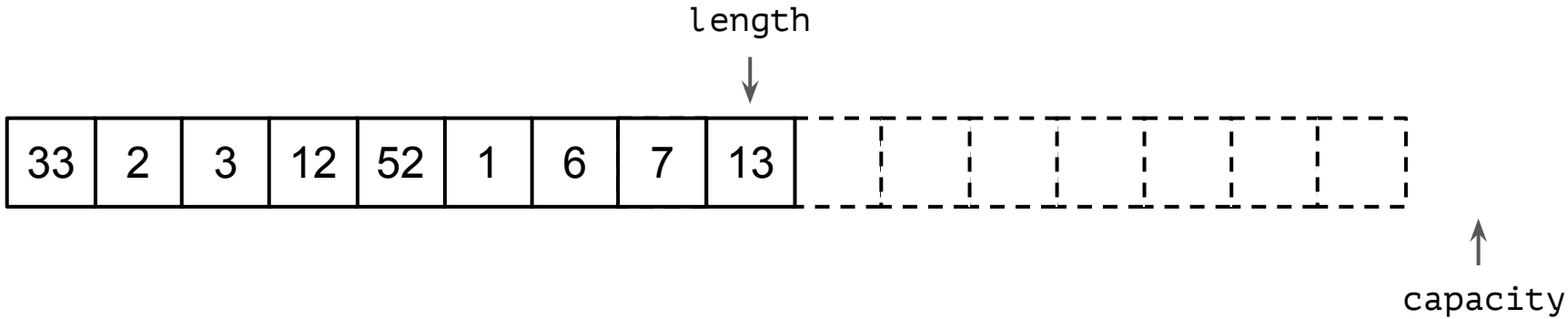
Как удалять последний элемент?

Но если вы хоть немного экономите память, то вы захотите "сдуть" (shrink) массив.

Когда? Наивный подход - когда дошли до середины.
Но теперь представьте, что дальнейшая последовательность такая:

```
vector.add(13)
vector.remove()
vector.add(13)
vector.remove()
```

Такая ситуация называется trashing и из-за нее амортизация работает хуже

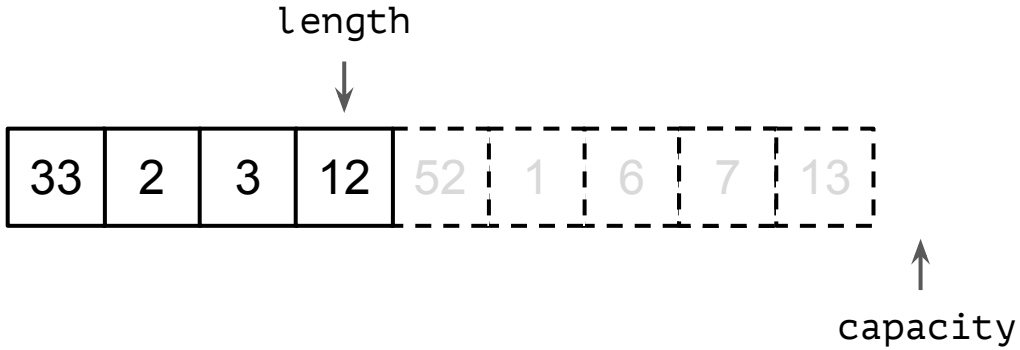


Как удалять последний элемент?

Но если вы хоть немного экономите память, то вы захотите "сдуть" (shrink) массив.

Когда? Наивный подход - когда дошли до середины.
Но теперь представьте, что дальнейшая последовательность такая:

Решение - сдувать массив, когда заполненной остается одна четверть.

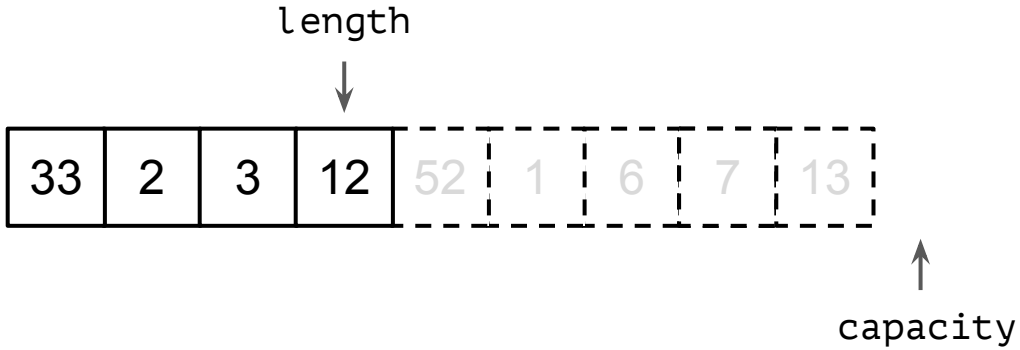


Как удалять последний элемент?

Но если вы хоть немного экономите память, то вы захотите "сдуть" (shrink) массив.

Когда? Наивный подход - когда дошли до середины.
Но теперь представьте, что дальнейшая последовательность такая:

Решение - сдувать массив, когда заполненной остается одна четверть.



Как удалять последний элемент?

Но если вы хоть немного экономите память, то вы захотите "сдуть" (shrink) массив.

Когда? Наивный подход - когда дошли до середины.
Но теперь представьте, что дальнейшая последовательность такая:

Решение - сдувать массив, когда заполненной остается одна четверть. Такое решение дает амортизационное время работы запросов O*(1)

Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

✓ 1. Доступ к i-ому элементу $O(1)$

✓ 2. Добавление элемента в "конец" $O^*(1)$

3. Удаление последнего элемента

Как реализовывать будем? Динамический массив!

Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

✓ 1. Доступ к i-ому элементу $O(1)$

✓ 2. Добавление элемента в "конец" $O^*(1)$

✓ 3. Удаление последнего элемента $O^*(1)$

Как реализовывать будем? Динамический массив!

Мини-задача #15 (1 балл)

Реализовать собственный динамический массив, в котором поддержаны операции:

1. Доступа к элементу по индексу
2. Добавление элемента в конец
3. Удаления последнего элемента

Операции работают за O(1) или O*(1).

Обрабатывается случаи выхода за границу массива и удаление элемента из пустого массива.

Пользоваться готовыми библиотечными решениями нельзя.

Динамические массивы

Минусы такого решения?

Динамические массивы

Минусы такого решения:

1. Учётное время работы хорошее,
   но сложность в худшем случае высокая.

Динамические массивы

Минусы такого решения:

1. Учётное время работы хорошее,
   но сложность в худшем случае высокая.

2. Избыточное потребление памяти (если значения большого размера)

Динамические массивы

Минусы такого решения:

1. Учётное время работы хорошее,
   но сложность в худшем случае высокая.

2. ~~Избыточное потребление памяти (если значения большого размера).~~ Редкая ситуация, тогда можно хранить массив ключей.

Динамические массивы

Минусы такого решения:

1. Учётное время работы хорошее, но сложность в худшем случае высокая.

2. ~~Избыточное потребление памяти (если значения большого размера).~~ Редкая ситуация, тогда можно хранить массив ключей.

3. Бывают операции и пострашнее добавления

Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

✓ 1. Доступ к i-ому элементу $O(1)$

✓ 2. Добавление элемента в "конец" $O^*(1)$

✓ 3. Удаление последнего элемента $O^*(1)$

4. Удалить элемент из "начала"?

Динамические массивы

Минусы такого решения:

1. Учётное время работы хорошее, но сложность в худшем случае высокая.
2. ~~Избыточное потребление памяти (если значения большого размера).~~ Редкая ситуация, тогда можно хранить массив ключей.
3. Бывают операции и пострашнее добавления

Динамические массивы

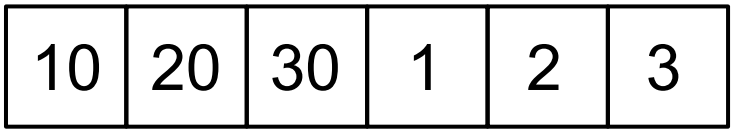
Альтернативы?

Связные списки!

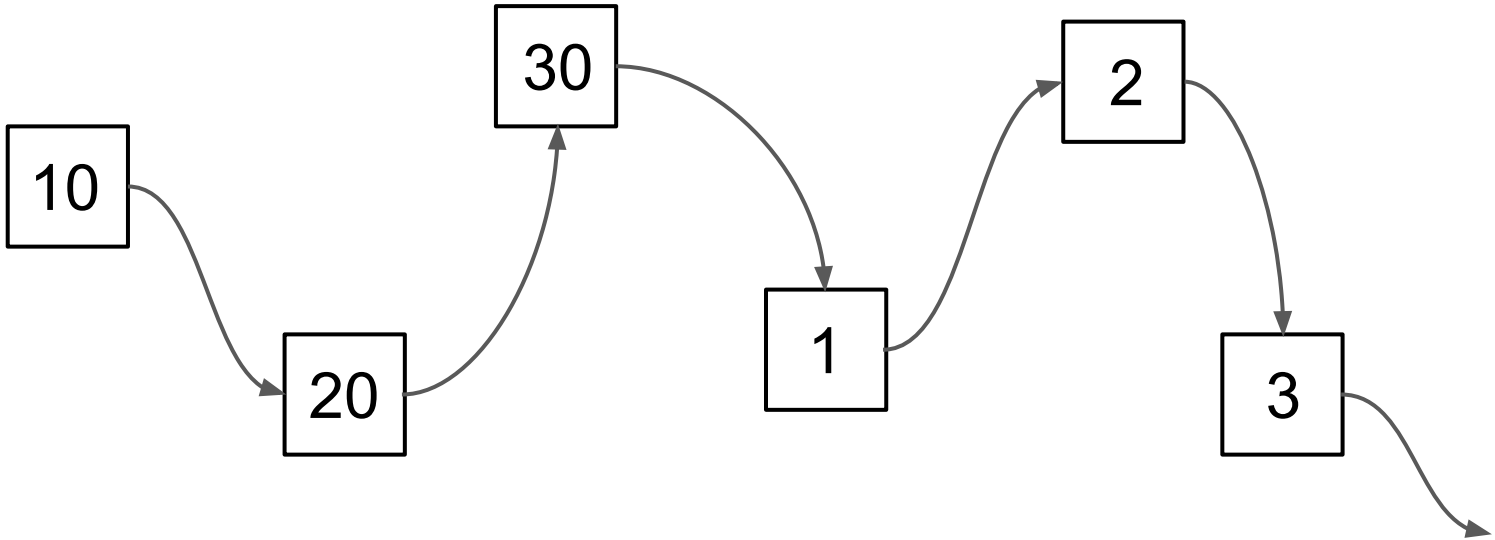
Минусы такого решения:

1. Учётное время работы хорошее, но сложность в худшем случае высокая.
2. ~~Избыточное потребление памяти (если значения большого размера).~~ Редкая ситуация, тогда можно хранить массив ключей.
3. Бывают операции и пострашнее добавления

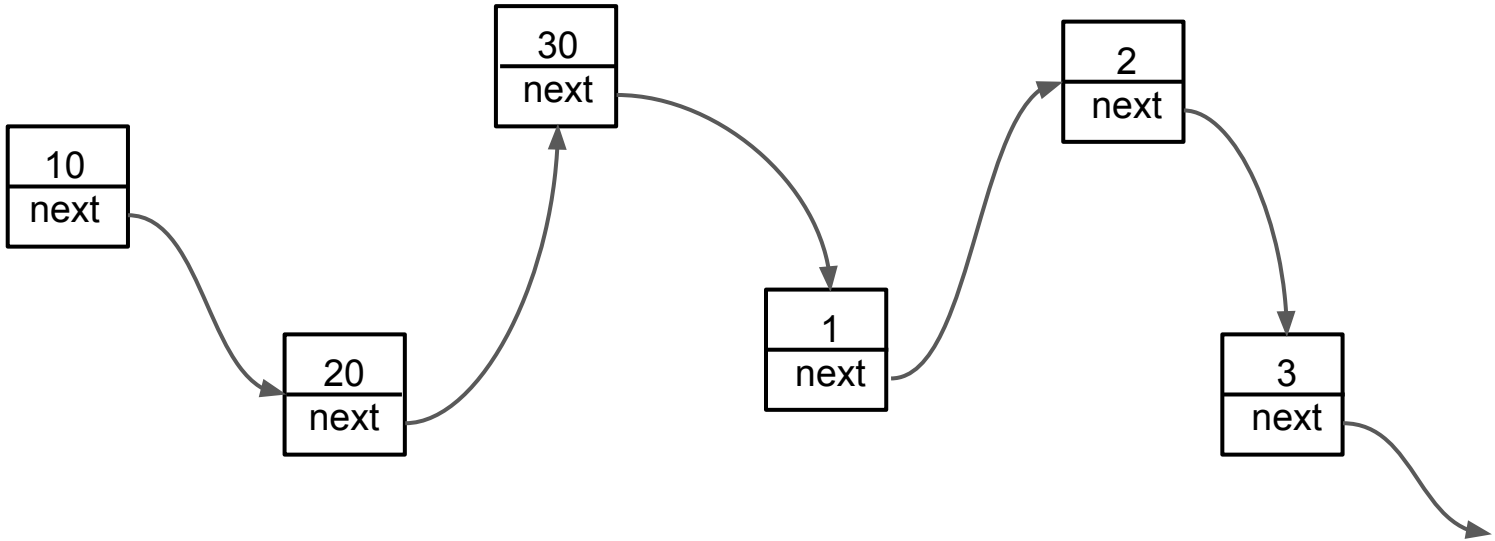
Связные списки



Связные списки



Связные списки



```
class LinkedList:
    class Node:
        def __init__(self, val, nxt=None):
            self.val = val
            self.next = nxt

def __init__(self):
    self.head = self.tail = None
```

```
class LinkedList:
    class Node:
        def __init__(self, val, nxt=None):
            self.val = val
            self.next = nxt

def __init__(self):
    self.head = self.tail = None

def add_to_tail(self, val):
    node = LinkedList.Node(val)
    if self.tail:
        self.tail.next = node
        self.tail = node
    else:
        self.head = self.tail = node
```

```
class LinkedList:
    class Node:
        def __init__(self, val, nxt=None):
            self.val = val
            self.next = nxt

def __init__(self):
    self.head = self.tail = None

def add_to_tail(self, val):
    node = LinkedList.Node(val)
    if self.tail:
        self.tail.next = node
        self.tail = node
    else:
        self.head = self.tail = node

def add_to_head(self, val):
    node = LinkedList.Node(val, self.head)
    self.head = node
    if not self.tail:
        self.tail = self.head
```

```
class LinkedList:
    class Node:
        def __init__(self, val, nxt=None):
            self.val = val
            self.next = nxt

def __init__(self):
    self.head = self.tail = None

def add_to_tail(self, val):
    node = LinkedList.Node(val)
    if self.tail:
        self.tail.next = node
        self.tail = node
    else:
        self.head = self.tail = node

def add_to_head(self, val):
    node = LinkedList.Node(val, self.head)
    self.head = node
    if not self.tail:
        self.tail = self.head
```

```
class LinkedList:
    class Node:
        def __init__(self, val, nxt=None):
            self.val = val
            self.next = nxt

def __init__(self):
    self.head = self.tail = None

def add_to_tail(self, val):
    node = LinkedList.Node(val)
    if self.tail:
        self.tail.next = node
        self.tail = node
    else:
        self.head = self.tail = node

def add_to_head(self, val):
    node = LinkedList.Node(val, self.head)
    self.head = node
    if not self.tail:
        self.tail = self.head
```

```
def get_kth_node(self, k: int) -> Node:
```

```
class LinkedList:
    class Node:
        def __init__(self, val, nxt=None):
            self.val = val
            self.next = nxt

def __init__(self):
    self.head = self.tail = None

def add_to_tail(self, val):
    node = LinkedList.Node(val)
    if self.tail:
        self.tail.next = node
        self.tail = node
    else:
        self.head = self.tail = node

def add_to_head(self, val):
    node = LinkedList.Node(val, self.head)
    self.head = node
    if not self.tail:
        self.tail = self.head
```

```
def get_kth_node(self, k: int) -> Node:
    node = self.head
    for i in [0; k):
        if not node:
            return None
        node = node.next
    return node
```

```
class LinkedList:
    class Node:
        def __init__(self, val, nxt=None):
            self.val = val
            self.next = nxt

def __init__(self):
    self.head = self.tail = None

def add_to_tail(self, val):
    node = LinkedList.Node(val)
    if self.tail:
        self.tail.next = node
        self.tail = node
    else:
        self.head = self.tail = node

def add_to_head(self, val):
    node = LinkedList.Node(val, self.head)
    self.head = node
    if not self.tail:
        self.tail = self.head
```

```
def get_kth_node(self, k: int) -> Node:
    node = self.head
    for i in [0; k):
        if not node:
            return None
        node = node.next
    return node


def sum(self) -> int:
```

```
class LinkedList:
    class Node:
        def __init__(self, val, nxt=None):
            self.val = val
            self.next = nxt

def __init__(self):
    self.head = self.tail = None

def add_to_tail(self, val):
    node = LinkedList.Node(val)
    if self.tail:
        self.tail.next = node
        self.tail = node
    else:
        self.head = self.tail = node

def add_to_head(self, val):
    node = LinkedList.Node(val, self.head)
    self.head = node
    if not self.tail:
        self.tail = self.head
```

```
def get_kth_node(self, k: int) -> Node:
    node = self.head
    for i in [0; k):
        if not node:
            return None
        node = node.next
    return node


def sum(self) -> int:
    node = self.head
    res = 0
    while node:
        res += node.val
        node = node.next
    return res
```

```
class LinkedList:
    class Node:
        def __init__(self, val, nxt=None):
            self.val = val
            self.next = nxt

def __init__(self):
    self.head = self.tail = None

def add_to_tail(self, val):
    node = LinkedList.Node(val)
    if self.tail:
        self.tail.next = node
        self.tail = node
    else:
        self.head = self.tail = node

def add_to_head(self, val):
    node = LinkedList.Node(val, self.head)
    self.head = node
    if not self.tail:
        self.tail = self.head
```

```
def get_kth_node(self, k: int) -> Node:
    node = self.head
    for i in [0; k):
        if not node:
            return None
        node = node.next
    return node


def sum(self) -> int:
    node = self.head
    res = 0
    while node:
        res += node.val
        node = node.next
    return res
```

Оцените сложности операций?

```
class LinkedList:
    class Node:
        def __init__(self, val, nxt=None):
            self.val = val
            self.next = nxt

def __init__(self):
    self.head = self.tail = None

def add_to_tail(self, val):
    node = LinkedList.Node(val)
    if self.tail:
        self.tail.next = node
        self.tail = node
    else:
        self.head = self.tail = node

def add_to_head(self, val):
    node = LinkedList.Node(val, self.head)
    self.head = node
    if not self.tail:
        self.tail = self.head
```

$O(1)$

$O(1)$

```
def get_kth_node(self, k: int) -> Node:
    node = self.head
    for i in [0; k):
        if not node:
            return None
        node = node.next
    return node


def sum(self) -> int:
    node = self.head
    res = 0
    while node:
        res += node.val
        node = node.next
    return res
```

Оцените сложности операций?

```
class LinkedList:
    class Node:
        def __init__(self, val, nxt=None):
            self.val = val
            self.next = nxt

def __init__(self):
    self.head = self.tail = None

def add_to_tail(self, val):
    node = LinkedList.Node(val)
    if self.tail:
        self.tail.next = node
        self.tail = node
    else:
        self.head = self.tail = node

def add_to_head(self, val):
    node = LinkedList.Node(val, self.head)
    self.head = node
    if not self.tail:
        self.tail = self.head
```

$O(1)$

$O(1)$

```
def get_kth_node(self, k: int) -> Node:
    node = self.head
    for i in [0; k):
        if not node:
            return None
        node = node.next
    return node


def sum(self) -> int:
    node = self.head
    res = 0
    while node:
        res += node.val
        node = node.next
    return res
```

$O(n)$

Оцените сложности операций?

```
class LinkedList:
    class Node:
        def __init__(self, val, nxt=None):
            self.val = val
            self.next = nxt

def __init__(self):
    self.head = self.tail = None

def add_to_tail(self, val):
    node = LinkedList.Node(val)
    if self.tail:
        self.tail.next = node
        self.tail = node
    else:
        self.head = self.tail = node

def add_to_head(self, val):
    node = LinkedList.Node(val, self.head)
    self.head = node
    if not self.tail:
        self.tail = self.head
```

$O(1)$

$O(1)$

```
def get_kth_node(self, k: int) -> Node:
    node = self.head
    for i in [0; k):
        if not node:
            return None
        node = node.next
    return node
```

$O(n)$

```
def sum(self) -> int:
    node = self.head
    res = 0
    while node:
        res += node.val
        node = node.next
    return res
```

$O(n)$

Оцените сложности операций?

Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

✓ 1. Доступ к i-ому элементу $O(1)$

✓ 2. Добавление элемента в "конец" $O^*(1)$

✓ 3. Удаление последнего элемента $O^*(1)$

Как реализовывать будем? Динамический массив!

Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

✓ 1. Доступ к i-ому элементу $O(n)$

✓ 2. Добавление элемента в "конец" $O(1)$

✓ 3. Удаление последнего элемента $O(1)$

Как реализовывать будем? Связный список!

Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

✓ 1. Доступ к i-ому элементу $O(n)$

✓ 2. Добавление элемента в "конец" $O(1)$

✓ 3. Удаление последнего элемента $O(1)$

4. Удалить элемент из "начала"?

Задача

Реализовать хранилище однотипных упорядоченных данных (например, чисел)

Которое бы поддерживало следующие запросы:

✓ 1. Доступ к i-ому элементу $O(n)$

✓ 2. Добавление элемента в "конец" $O(1)$

✓ 3. Удаление последнего элемента $O(1)$

✓ 4. Удалить элемент из "начала"? $O(1)$

Связные списки

Минусы такого решения?

Связные списки

Минусы такого решения:

1. Взятие i-ого за линию.

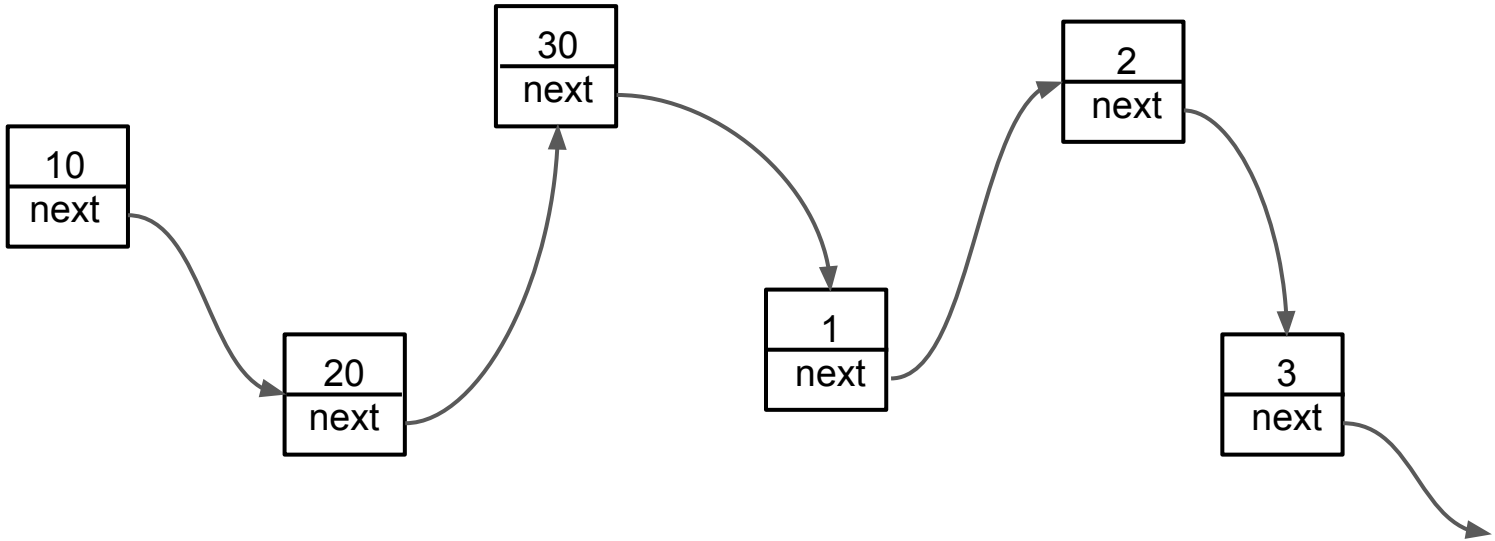
Связные списки

Минусы такого решения:

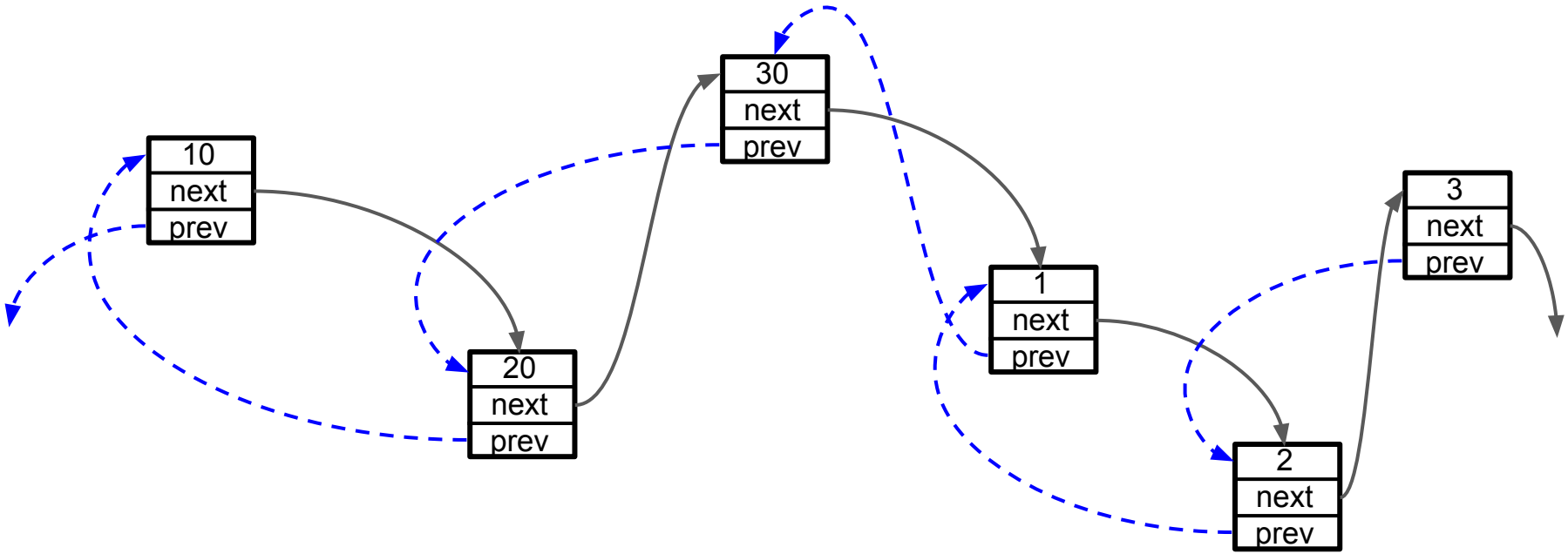
1. Взятие i-ого за линию.

2. Нужно хранить память на следующий элемент (а иногда и на предыдущий, такие списки называются двусвязными)

Односвязные списки



Двусвязные списки



Связные списки

Минусы такого решения:

1. Взятие i-ого за линию.
2. Нужно хранить память на следующий элемент (а иногда и на предыдущий, такие списки называются двусвязными)
3. Теряется локальность данных (последовательный доступ дольше)

Динамические массивы VS связные списки

	Массивы	Списки
Взятие элемента по индексу	O(1)	O(N)
Добавление/удаление в конец	O(N), но O*(1)	O(1)

Динамические массивы VS связные списки

	Массивы	Списки
Взятие элемента по индексу	O(1)	O(N)
Добавление/удаление в конец	O(N), но O*(1)	O(1)
Поиск по значению	O(N)	O(N)

Динамические массивы VS связные списки

	Массивы	Списки
Взятие элемента по индексу	O(1)	O(N)
Добавление/удаление в конец	O(N), но O*(1)	O(1)
Поиск по значению	O(N)	O(N)
Поиск по значению в отсортированном	O(logN)	O(N)

Мини-задача #16 (1 балл)

По заданному связному списку нужно найти цикл в нем и вернуть номер узла, с которого он начался.

Нельзя:

1. Модифицировать элементы списка
2. Пользоваться встроенными коллекциями

Нужно: использовать только O(1) дополнительной памяти.

Решение проверить на leetcode:
https://leetcode.com/problems/linked-list-cycle-ii/

Мини-задача #17 (1 балл)

По заданному связному списку и двум значениям элементов из него: построить новый связный список в котором все элементы между заданных в условии идут в обратном порядке.

Для решения используйте однопроходный алгоритм.

Решение проверить на leetcode:
https://leetcode.com/problems/reverse-linked-list-ii/

Takeaways

- Амортизационный анализ, как альтернатива сложности в худшем случае.

Takeaways

- Амортизационный анализ, как альтернатива сложности в худшем случае.
- Динамические массивы и правила их расширения и сдувания.

Takeaways

- Амортизационный анализ, как альтернатива сложности в худшем случае.
- Динамические массивы и правила их расширения и сдувания.
- Связные списки, как альтернатива динамическим массивам.
- Trade-offs

Мини-задача #15 (1 балл)

Реализовать собственный динамический массив, в котором поддержаны операции:

1. Доступа к элементу по индексу
2. Добавление элемента в конец
3. Удаления последнего элемента

Операции работают за O(1) или O*(1).

Обрабатывается случаи выхода за границу массива и удаление элемента из пустого массива.

Пользоваться готовыми библиотечными решениями нельзя.

Мини-задача #16 (1 балл)

По заданному связному списку нужно найти цикл в нем и вернуть номер узла, с которого он начался.

Нельзя:

1. Модифицировать элементы списка
2. Пользоваться встроенными коллекциями

Нужно: использовать только O(1) дополнительной памяти.

Решение проверить на leetcode:
https://leetcode.com/problems/linked-list-cycle-ii/

Мини-задача #17 (1 балл)

По заданному связному списку и двум значениям элементов из него: построить новый связный список в котором все элементы между заданных в условии идут в обратном порядке.

Для решения используйте однопроходный алгоритм.

Решение проверить на leetcode:
https://leetcode.com/problems/reverse-linked-list-ii/