# Lecture 1.

## Algorithm complexity estimation.
## Basic data structures.

**Algorithms and Data Structures**
**Ivan Solomatin**
**MIPT 2020**

# Outline

- Course description.
- Algorithm complexity basics
- Basic data structures
  - Stack (static memory implementation)
  - Queue (static memory implementation)
  - Vector
  - Linked List
  - Doubly Linked List

# Course description

## Course goal:

- Get familiar with basic algorithms and data structures.
- Learn to implement basic algorithms and data structures on python.
- Learn how to apply obtained knowledge in practice.
- Get experience and intuition in programming problems solving.

## Sources:

- Thomas H. Cormen, et al. *Introduction to algorithms*. MIT press, 2009.
- www.geeksforgeeks.org/fundamentals-of-algorithms
- www.e-maxx.ru/algo/

# Course description

Course content:

- Lectures (online)
- Homework (problem solving in contest.yandex.ru)
- Seminars (online, real time):
  - homework analysis
  - Q&A

Final grade:

- Homework (~60%)
- Practical exam (problem solving) (~20%)
- Theoretical exam (~20%)

# Algorithm complexity basics
## Definition

Algorithm:

$$A: X \to Y$$

Number of elementary operations (processor instructions) for executing A on input $x \in X$:

$$t(A, x)$$

Size of input x:

$$s(x)$$

Worst case complexity:

$$\boldsymbol{N} \to \infty$$

$$T(A, N) = \max_{\substack{x \in X \\ s(x) = N}} t(A, x)$$

$$50N + 100500 = \boldsymbol{O(N)}$$

$$10N^2 + 5N + 1 = \boldsymbol{O(N^2)}$$

$$N \log_2 N = N \frac{\log_c N}{\log_c 2} = \boldsymbol{O(N \log N)}$$

$$42 = \boldsymbol{O(1)}$$

# Algorithm complexity basics
## Examples

Problem:

Calculate sum of two given numbers $x, y$:

$$x, y \in [-2^{63}; 2^{63})$$

Input:

- $x$ – 64 bit integer = 8 bytes
- $y$ – 64 bit integer = 8 bytes

input size: 16 bytes

Algorithm:

calculate sum, return result

Complexity:

$$O(1)$$

# Algorithm complexity basics

## Examples

Problem:

$$\text{Calculate sum of } N \text{ numbers } x_1, \dots, x_N.$$
$$x_i \in [-2^{31}; 2^{31})$$

Input:

- $x_1$ – 32bit integer = 4 bytes
- …
- $x_N$ – 32bit integer = 4 bytes

size: $8N$ bytes

Algorithm:

Iterate over $x_i$ and accumulate sum:

```
res = 0
for i in range(N):
    res += x[i]
```

Complexity:

$$O(N)$$

# Algorithm complexity basics

## Examples

**Problem:**

Check if there is a pair of equal numbers between given

$N$ numbers $x_1, \dots, x_N$.

$x_i \in [-2^{31}; 2^{31})$

**Input:**

- $x_1$ – 32bit integer = 4 bytes
- …
- $x_N$ – 32bit integer = 4 bytes

size: $8N$ bytes

**Algorithm:**

Iterate over each pair and check equality:

```
def f(x):
  N = len(x)
  for i in range(N):
    for j in range(i + 1, N):
      if x[i] == x[j]:
        return True
  return False
```

**Complexity:**

In worst case: $(\mathrm{N}-1)+(\mathrm{N}-2)+\cdots+1=\frac{N(\mathrm{N}-1)}{2}=O(N^2)$

# Basic data structures

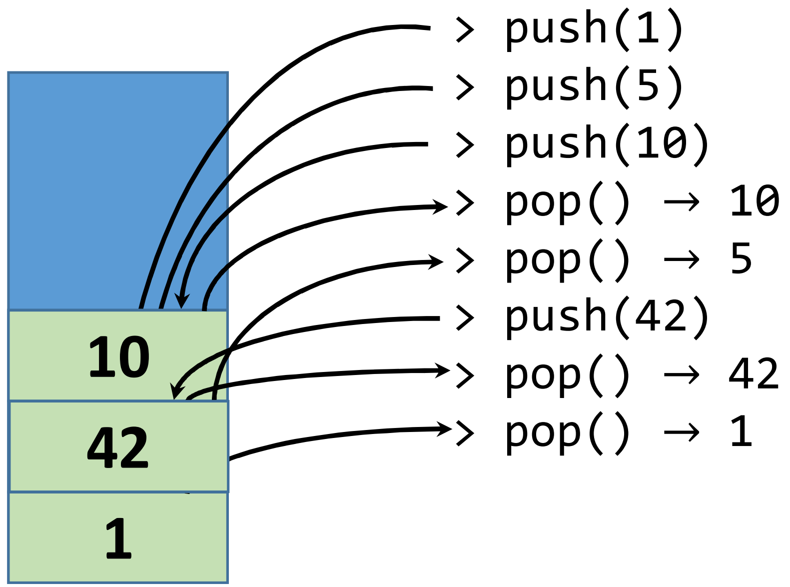
- Stack (LIFO)
- Queue (FIFO)

# Stack (LIFO: last it, first out)

## Definition

Data structure with following operations defined

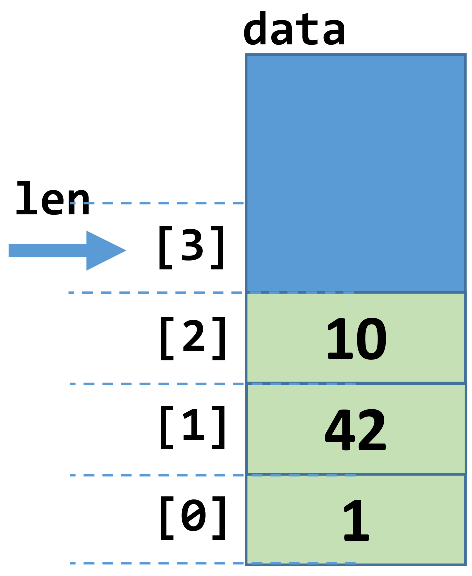
- push(x) – place x in top of the stack
- front() – return value from top of the stack
- pop() – return value from top of the stack and removes it
- len() – return number of elements in stack



# Stack (LIFO: last it, first out)

## Implementation idea

- Let's suppose that we know maximum number of elements which can be present in stack simultaneously: **`max_len`**
- Initialize an array (list) of **`max_len`** elements: **`data.`**
- Store **`len`** variable which denotes current number of elements in stack.
- push will write element to **`data[len]`** and increase **`len`**.
- front (and pop) will return **`data[len-1].`** pop will also decrease `len`.
- len will return **`data[len-1].`**




```
> push(1)
> push(5)
> push(10)
> pop() → 10
> pop() → 5
> push(42)
> pop() → 42
> pop() → 1
```

# Stack (LIFO: last it, first out)

## Complexity:

- push: $O(1)$ (assigning **`data[len]`** value and increasing **`len`**)
- front: $O(1)$ (getting and returning **`data[len]`**)
- pop: $O(1)$ (front(), decreasing **`len`**)
- size: $O(1)$ (just returning **`len`**)

# Stack (LIFO: last it, first out)

## Code example

```
class Stack:
    def __init__(self, max_len):
        self.data = [None] * max_len
        self.len = 0

    def push(self, val):
        self.data[self.len] = val
        self.len += 1

    def front(self):
        return self.data[self.len - 1]

    def pop(self):
        res = self.front()
        self.len -= 1
        return res

    def clear(self):
        self.len = 0

    def __len__(self):
        return self.len
```

# Queue (FIFO: first in, first out)
## Definition

Data structure with following operations defined:

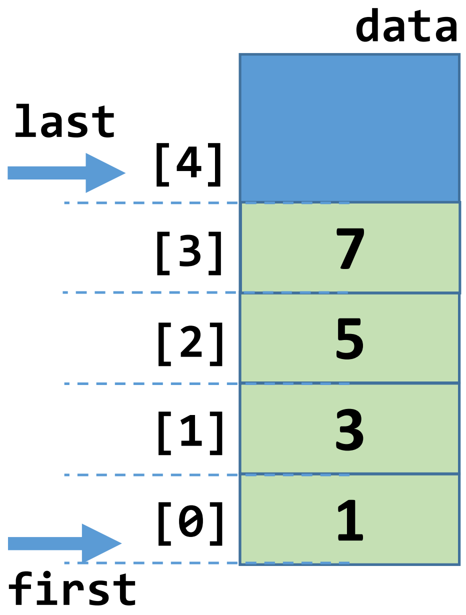
- push(x) – place x in the queue as last element
- front() – return first element of the queue
- pop() – return first element of the queue and removes it
- len() – return number of elements in queue

```
> push(1)
> push(3)
> push(5)
> pop() → 1
> pop() → 3
> push(7)
> pop() → 5
> pop() → 7
```

# Queue (FIFO: first in, first out)

## Implementation idea

- Let’s suppose that we know maximum number of elements which can be present in queue simultaneously: **`max_len`.**
- Initialize an array (list) of **`max_len`** elements: **`data`.**
- Store:
    - **`first`:** current position of first element
    - **`last`:** current position of last element
    - **`len`:** number of elements in queue.
- push will write element to **`data[last]`** and increase **`last`** and **`len`**.
- front (and pop) will return **`data[first]`.** pop will also increase **`first`** and decrease **`len`**.
- len will return **`len`.**
- If **`last`** (or **`first`**) equals **`max_len`**, set **`last`** (or **`first`**) to 0.




```
> push(1)
> push(3)
> push(5)
> pop() → 1
> pop() → 3
> push(7)
> pop() → 5
> pop() → 7
```

# Queue (FIFO: first in, first out)

## Complexity

- push: $O(1)$ (assigning value, increasing **`len`** and **`last`**)
- front: $O(1)$ (getting and returning **`data[first]`**)
- pop: $O(1)$ (front(), increasing **`first`**, decreasing **`len`**)
- len: $O(1)$ (returning **`len`**)

# Queue (FIFO: first in, first out)

Code example

```
class Queue:
    def __init__(self, max_len):
        self.max_len = max_len
        self.data = [None] * self.max_len
        self.first = 0
        self.last = 0
        self.len = 0

    def push(self, val):
        self.data[self.last] = val
        self.len += 1
        self.last += 1
        if self.last == self.max_len:
            self.last = 0

    def front(self):
        if len(self) > 0:
            return self.data[self.first]
        else:
            raise IndexError

    def pop(self):
        res = self.front()
        self.first += 1
        self.len -= 1
        if self.first == self.max_len:
            self.first = 0
        return res

    def clear(self):
        self.first = 0
        self.last = 0
        self.len = 0

    def __len__(self):
        return self.len
```

# Basic data structures

- Vector
- Linked List
- Doubly Linked List

# Vector

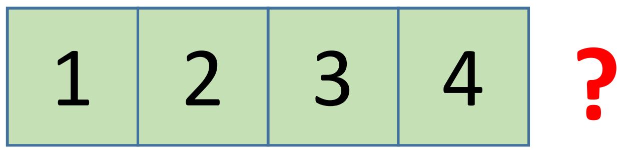
## Definition

Data structure with following operations defined:

- push_back(x) – place x in the vector as last element
- get(i) – return i-th element (numerating from 0)
- len() – return number of elements

+ we **don’t know** any limitations on number of elements

```
> push_back(1)
> push_back(2)
> push_back(3)
> get(1) → 2
> push_back(4)
> push_back(5)
```



# Vector

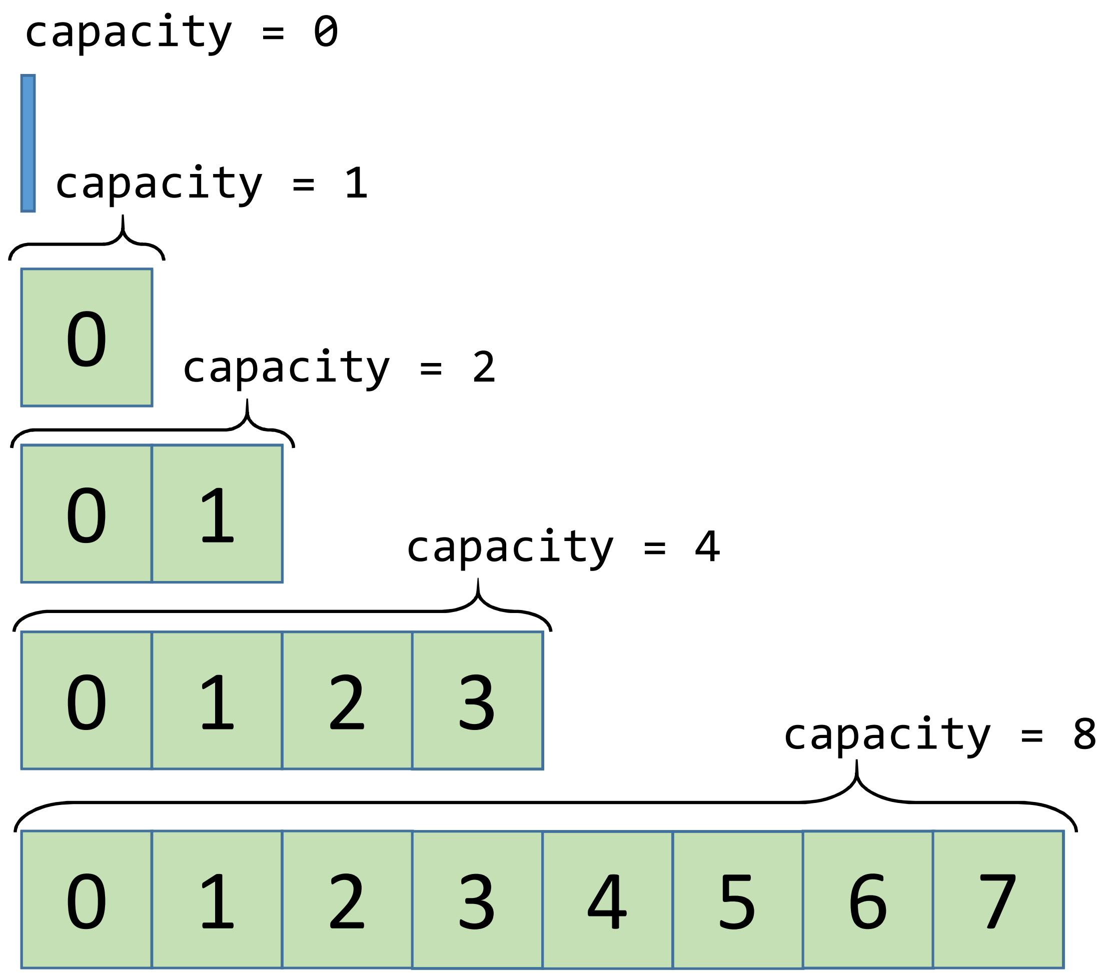
## Implementation idea

- Store **`capacity`** variable, which denotes amount of data allocated (Initially: **`capacity=0`**)
- Store size variable, which denotes number of elements in vector.
- push_back checks if **`(size == capacity)`**, and if yes, increases capacity, reallocates memory and copies all current data to new memory. Then deallocates old memory.
  ```
  new_capacity = (1 if capacity == 0 else capacity * 2)
  ```
- get(i) returns data[i]

# Vector

## Implementation idea

```
> push_back(0)
> push_back(1)
> push_back(2)
> push_back(3)
> push_back(4)
> push_back(5)
> push_back(6)
> push_back(7)
```



# Vector

## Complexity

- push_back: strictly saying, $O(N)$, because in worst case we need to reallocate memory and copy all data.
- get: $O(1)$ (getting and returning **`data[i]`**)

push_back takes $O(N)$ operations in worst case due to reallocation, but this case happens rarely. In most cases it takes $O(1)$ operations.

Let's calculate *amortized complexity* of push_back operation, which is average complexity for series of N operations:

$$T_{amortized} = \frac{1}{N}\sum_{i=0}^{N} T(i)\,,$$

where $T(i)$ is complexity of i-th operation.

# Vector

## Complexity

So, we should estimate average complexity of push_back operation when adding N elements one-by-one starting from zero:

Complexity of adding i-th element to vector (numerating from zero):

$$T(i) = \begin{cases} c_1, & if\ i \neq 2^k \\ c_1 + ic_2, & if\ i = 2^k \end{cases}$$

So, complexity of pushing N elements to vector is:

$$\sum_{i=0}^{N} T(i) = Nc_1 + \sum_{i=0}^{N} \begin{cases} 0, if\ i \neq 2^k \\ i, if\ i = 2^k \end{cases} = Nc_1 + c_2 \underbrace{\boxed{\sum_{i=0}^{\lfloor \log_2 N \rfloor} 2^i}}_{\leq 2N} \leq (c_1 + 2c_2)N = O(N)$$

That means that for push_back operation:

$$T_{amortized} \leq \frac{(c_1 + 2c_2)N}{N} = (c_1 + 2c_2) = O(1)$$

# Vector
## Complexity

Accounting method for estimation $T_{amortized}$.

Each operation call has it's cost (number of instructions). Let's suppose that for each instruction executing, you need $\$1$.

The idea of accounting method is to assign fixed cost to operation and prove that in each moment, amount of money obtained by data structure is enough for all the internal operations (like reallocation and copying).

Let's include cost of reallocation and copying elements to push_back cost, and use saved money when we do reallocation.

So, let's define cost of push_back as $(c_1 + 2c_2)\$$. (Adding to allocated memory costs $c_1\$$ and copying 1 element costs $c_2\$$).

# Vector

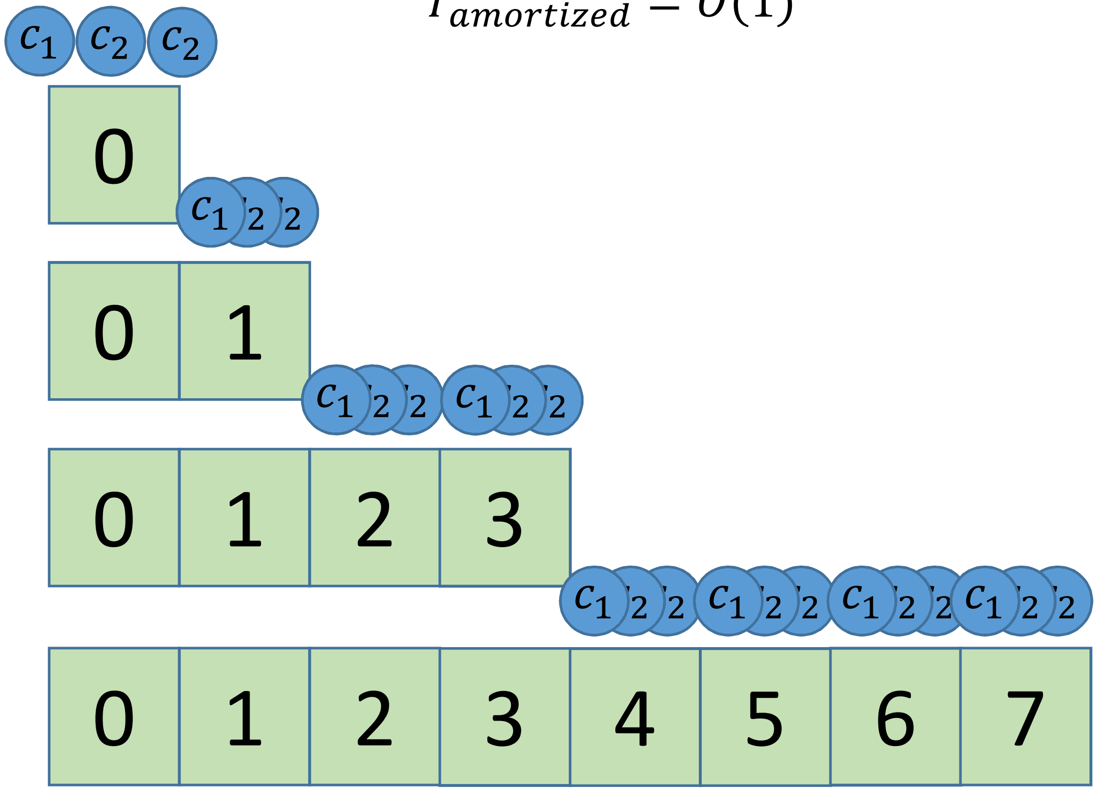
## Complexity

push_back cost: $(c_1 + 2c_2)\$$

We can prove using induction that money obtained for push_back is always enough to pay for reallocation.
So, for push_back operation:

$$T_{amortized} = O(1)$$

```
> push_back(0)
> push_back(1)
> push_back(2)
> push_back(3)
> push_back(4)
> push_back(5)
> push_back(6)
> push_back(7)
```



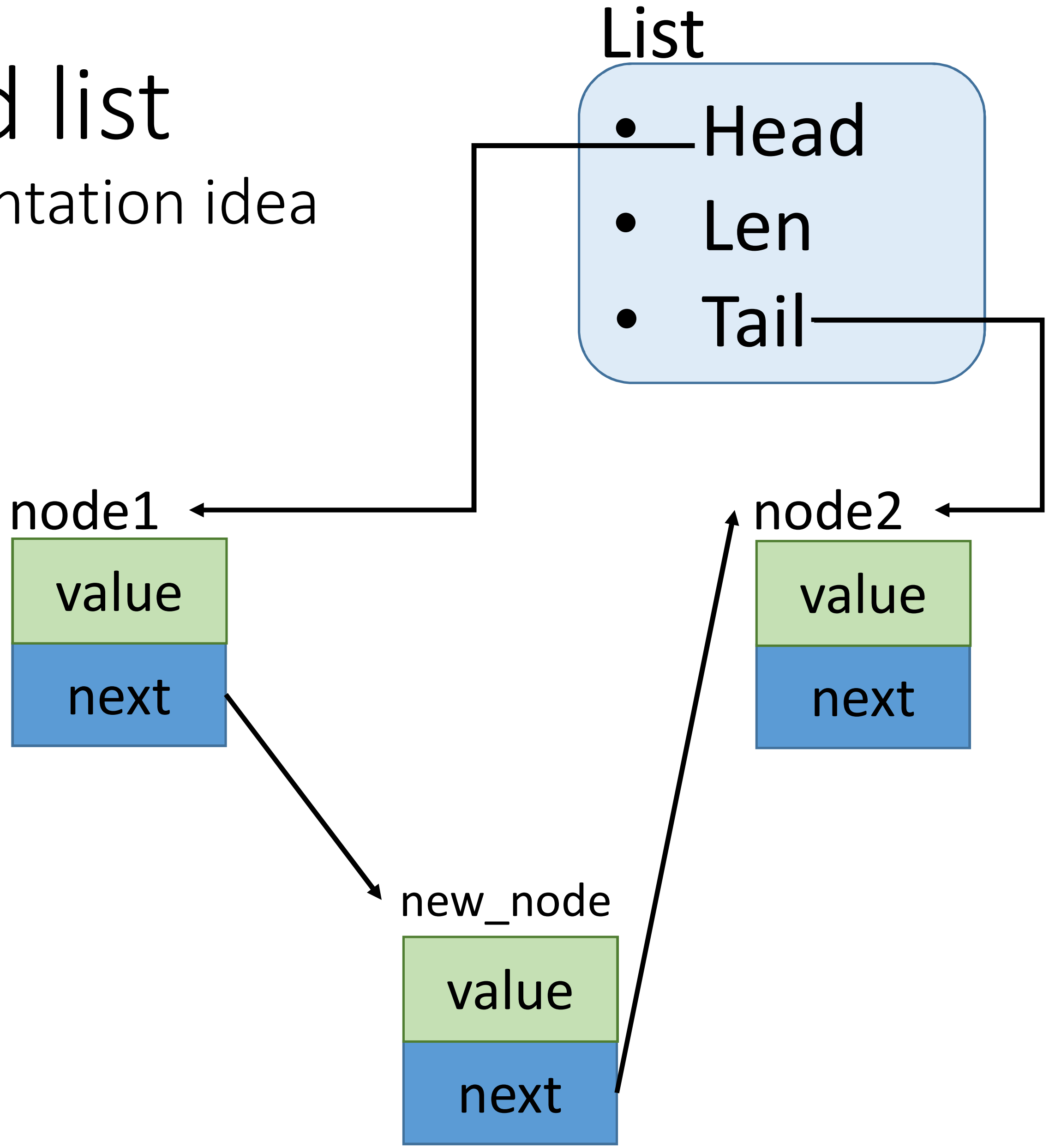
# Vector

## Set of operations

| Operation | Complexity (for Vector) | Comments |
|---|---|---|
| push_back | $O(1)$ | $O(1)$ is amortized complexity. Worst case for one operation is $O(N)$. |
| pop_back | $O(1)$ | We can use the same approach for pop_back as we used for stack on static memory, because we store size. |
| get (by index) | $O(1)$ | As elements are stored consistently in a whole piece of memory, so we can easily calculate address of each element. |
| insert | $O(N)$ | If we want to insert element into the middle of vector, we need to shift all elements to the right to save consistency. This means copying all elements to the right of inserted one. In worst case: N. |
| remove | $O(N)$ | As in the previous case, we need to shift elements to save consistency is we want to remove element from the middle. |
| push_front | $O(N)$ | We can use insert operation for that.<br>Also we can use an approach we used in static queue and reach amortized $O(1)$, but it will cause changes in vector structure. |
| pop_front | $O(N)$ | The same as for push_front. |

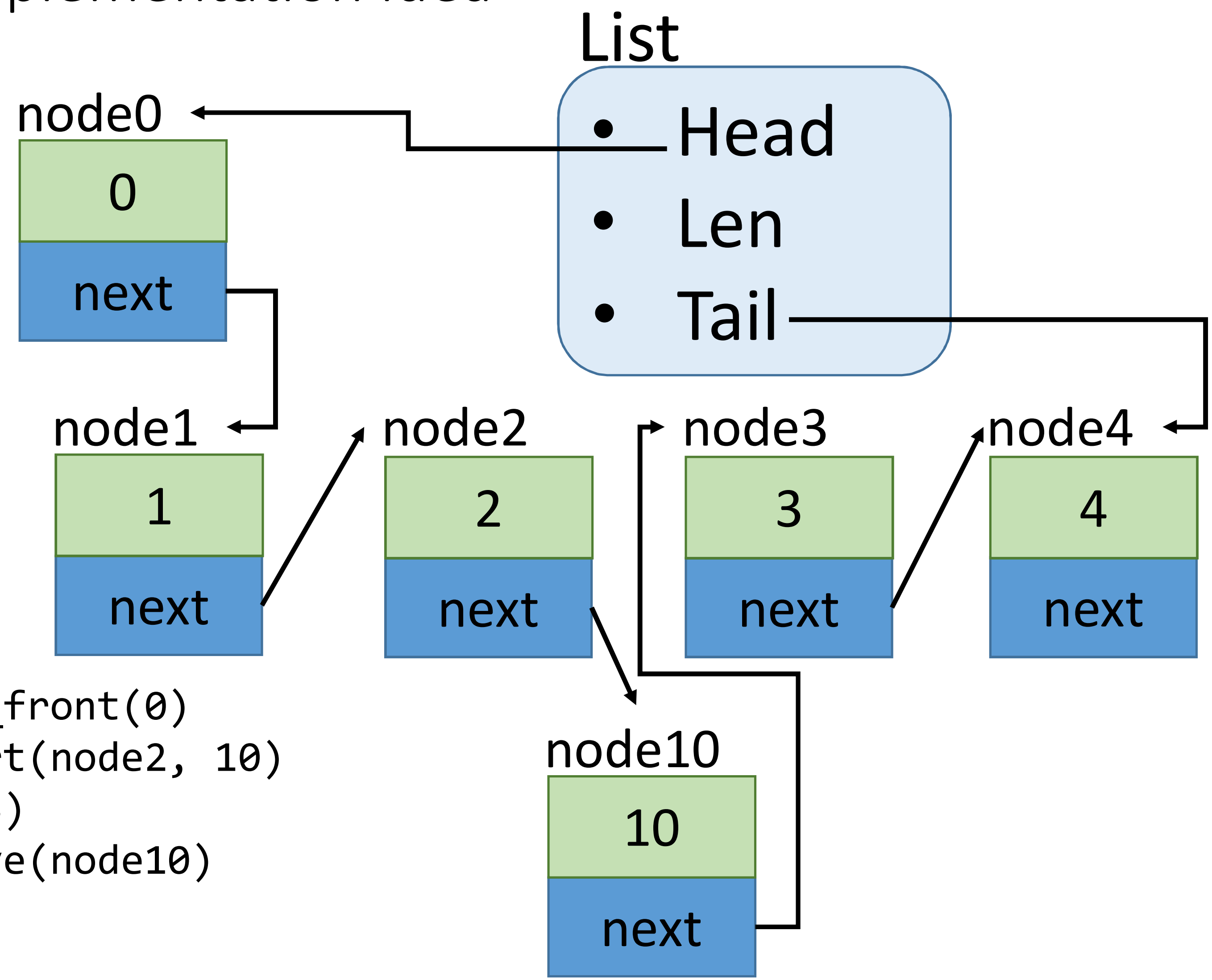
# Linked list

Implementation idea



# Linked list

## Implementation idea




```
> push_front(0)
> insert(node2, 10)
> get(3)
> remove(node10)
```

# Linked list

## Implementation

```python
class ListNode:
    def __init__(self, val, next):
        self.val = val
        self.next = next

    def __next__(self):
        return self.next

    def get_value(self):
        return self.val

    def copy(self):
        return ListNode(self.val, self.next)
```

# Linked list

## Implementation

```
class List:
    def __init__(self):
        self.head = None
        self.tail = None
        self.len = 0

    def push_front(self, val):
        new_node = ListNode(val, self.head)
        self.head = new_node
        if self.tail is None:
            self.tail = new_node
        self.len += 1

    def insert(self, previous_node, val):
        if not isinstance(previous_node, ListNode):
            raise TypeError("Expected previous_node to be ListNode instance")
        new_node = ListNode(val, previous_node.next)
        previous_node.next = new_node
        self.len += 1
        if new_node.next is None:
            self.tail = new_node
        return new_node

    def push_back(self, val):
        return self.insert(self.tail, val)
```

# Linked list

## Implementation

```
    def get_node_by_index(self, i):
        if not (0 <= i < self.len):
            raise IndexError('List index out of range')
        res = self.head
        for i in range(i):
            res = next(res)
        return res

    def insert_by_index(self, i, val):
        if isinstance(i, int):
            if i < 0:
                i += self.len + 1
            if i == 0:
                return self.push_front(val)
            else:
                prev_node = self.get_node_by_index(i - 1)
                return self.insert(prev_node, val)
        else:
            TypeError("Expected i to be integer")

    def __len__(self):
        return self.len

    def __getitem__(self, i):
        return self.get_node_by_index(i).val
```

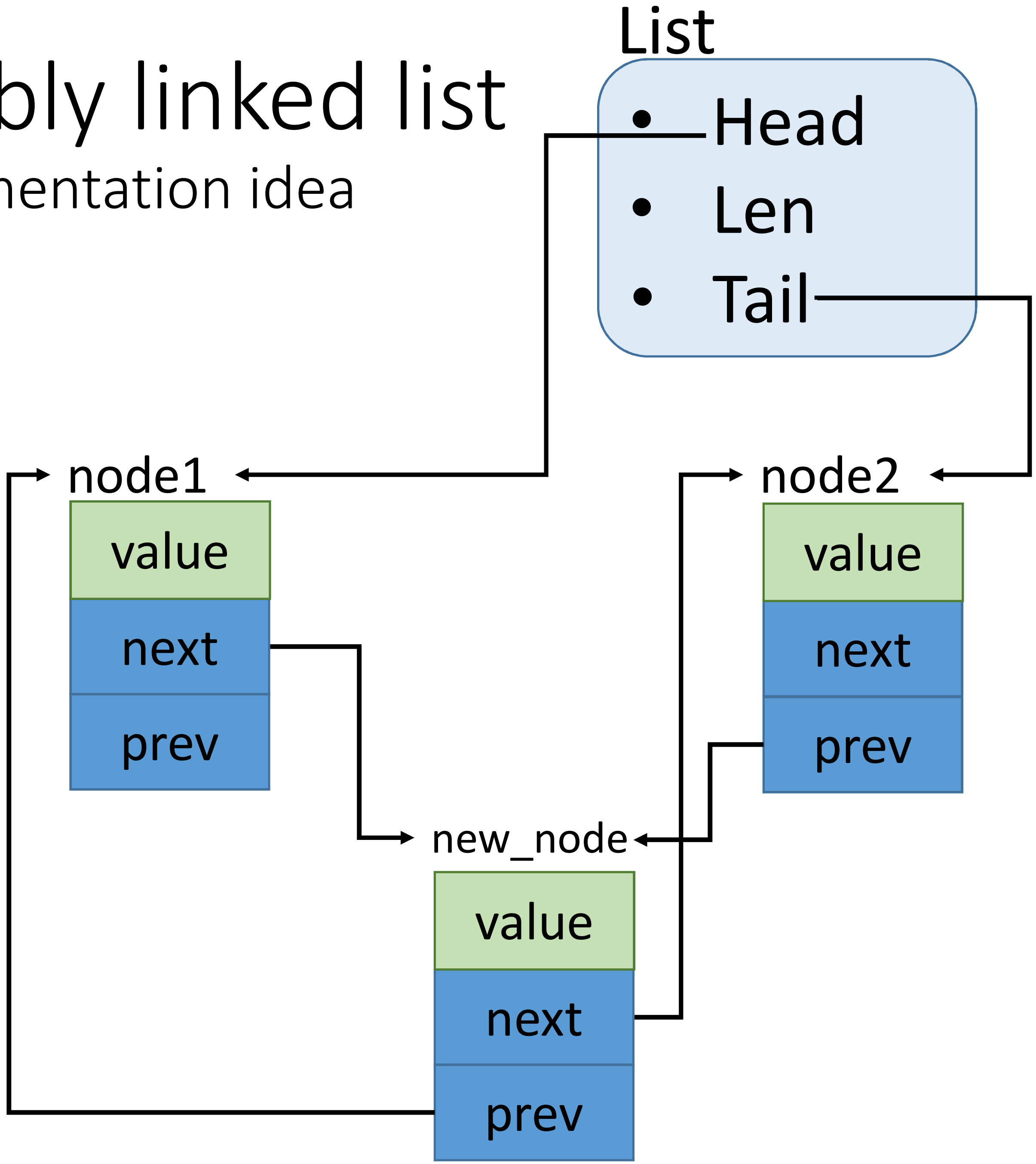
# Linked list

## Set of operations

| Operation | Complexity (for Vector) | Comments |
|---|---|---|
| push_back | $O(1)$ | |
| pop_back | $O(\mathrm{N})$ | To remove last element of list we need to get previous one. We can do that only iterating over all list starting from head. |
| get | $O(\mathrm{N})$ | Getting element by index requires iterating over all list. In worst case: $O(N)$ operations. |
| insert | $O(1)$ | If we has a link to a list note to insert element next, it is $O(1)$.<br>To insert by index, we need to perform get() first. |
| remove | $O(N)$ | As in pop_back, we need to iterate over list to obtain previous node. |
| push_front | $O(1)$ | |
| pop_front | $O(1)$ | Fortunately, we don't have previous element for first one, so, we can just overwrite head. |

# Doubly linked list

Implementation idea



# Linked list

## Extended set of operations

| Operation | Complexity (for Vector) | Comments |
| --- | --- | --- |
| push_back | $O(1)$ | |
| pop_back | $O(1)$ | |
| get (by index) | $O(\mathrm{N})$ | Getting element by index requires iterating over all list. In worst case: $O(N)$ operations. |
| insert | $O(1)$ | If we have a node to insert next element, it is $O(1)$.<br>To insert by index, we need to perform get() first. |
| remove | $O(1)$ | If we have a node to be deleted, it is $O(1)$.<br>To remove by index, we need to perform get() first. |
| push_front | $O(1)$ | |
| pop_front | $O(1)$ | |

We can implement stack and queue using

- Vector
- Linked List
- Doubly Linked List

- list (vector implementation)
- collections.deque (doubly linked list implementation)
- list can be used as stack:

```
x = list()
x.append(1)
x.append(10)
print(x.pop())
print(x.pop())
```

- collections.deque can be used as queue:

# Thank you for watching!