Lecture 3.
Basic data structures.

Algorithms and Data Structures
Ivan Solomatin
Harbour.Space@UTCC Bangkok

Outline

- Basic data structures
  - Static array
  - Vector (Dynamic array)
  - Linked List
  - Doubly Linked List
- Basic data structure interfaces
  - Stack (LIFO)
  - Queue (FIFO)
  - Deque

Basic data structures

- **Static array**
- Dynamic array (vector)
- Linked List
- Doubly Linked List

Static array
Definition

Pre-allocated fixed part of memory a. All values are stored contiguously and have same size: s bytes. Total amount of memory: $s * N$ bytes.

- set(i, value) – assign value to i-th item (numerating from 0)
- get(i) – return value of i-th item (numerating from 0)

As soon as values are stored contiguously, we can calculate address (number of first byte) of i-th element:

$$adress(a[i]) = adress(a) + i\ * s$$

```
> set(1, 2)
> set(3, 4)
> get(1) → 2
> get(0) → 0
```

	0	1	2	3	4	5
a	0	2	0	4	0	0

Basic data structures

- Static array
- **Dynamic array (vector)**
- Linked List
- Doubly Linked List

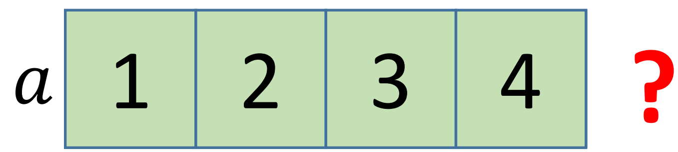
Vector

Definition

- set(i, value) – assign value to i-th item
- get(i) – return value of i-th item
- push_back(x) – add value x as new last element
- len() – return current 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

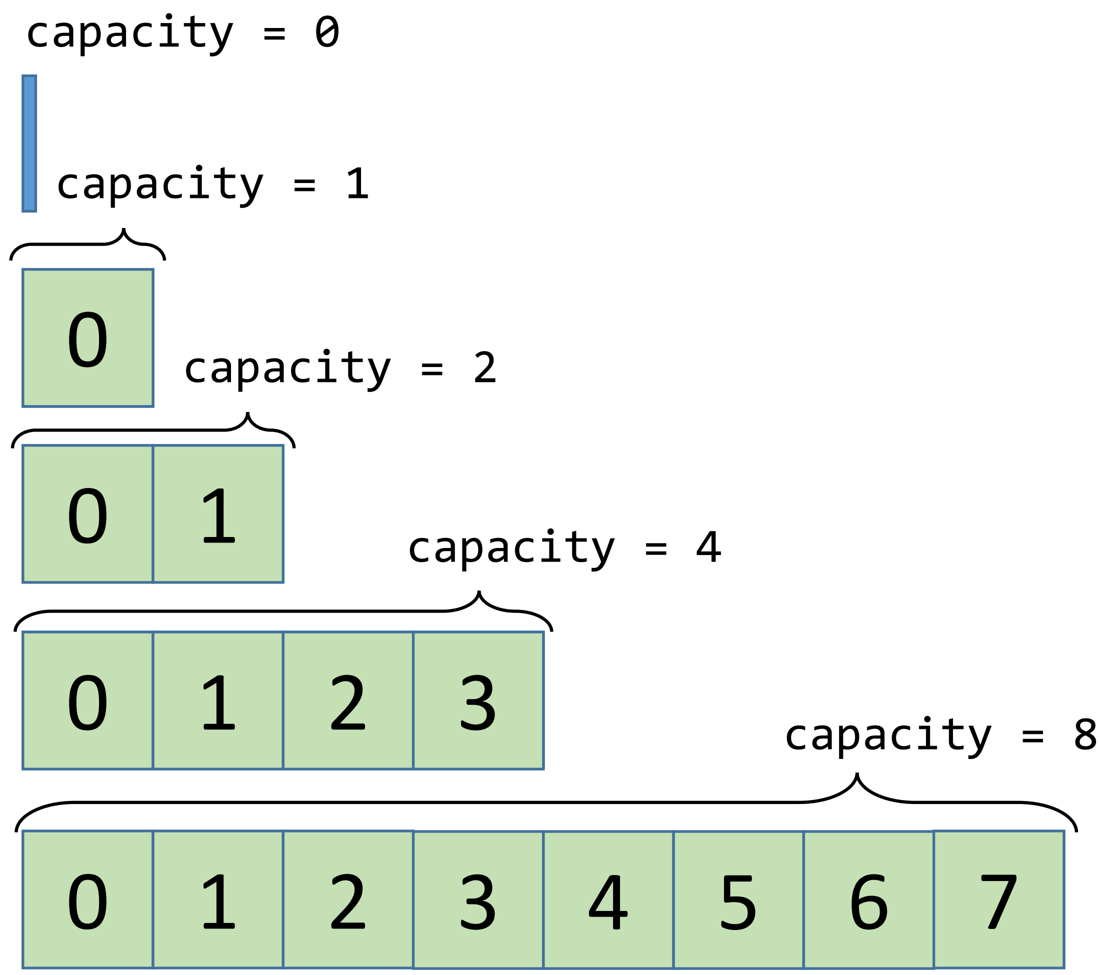
- Let's store elements contiguously to be able to easily get address of i-th element.
- Let's store **`capacity`** variable, which denotes amount of memory allocated in **`a`** (Initially: **`capacity=0`**)
- Let's store **`len`** variable, which denotes current number of elements in vector.
- push_back checks if **`(len < capacity)`**, and if yes, it just assigns value to **`a[len]`**, then increases **`len`**. Otherwise it increases **`capacity`**, reallocates memory and copies all current data to ne w memory. Then deallocates old memory.

```
new_capacity = (1 if capacity == 0 else capacity * 2)
```

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

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

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):

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

c_1: adding to allocated memory

c_2: copying 1 element

So, complexity of pushing N elements to vector is:

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

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

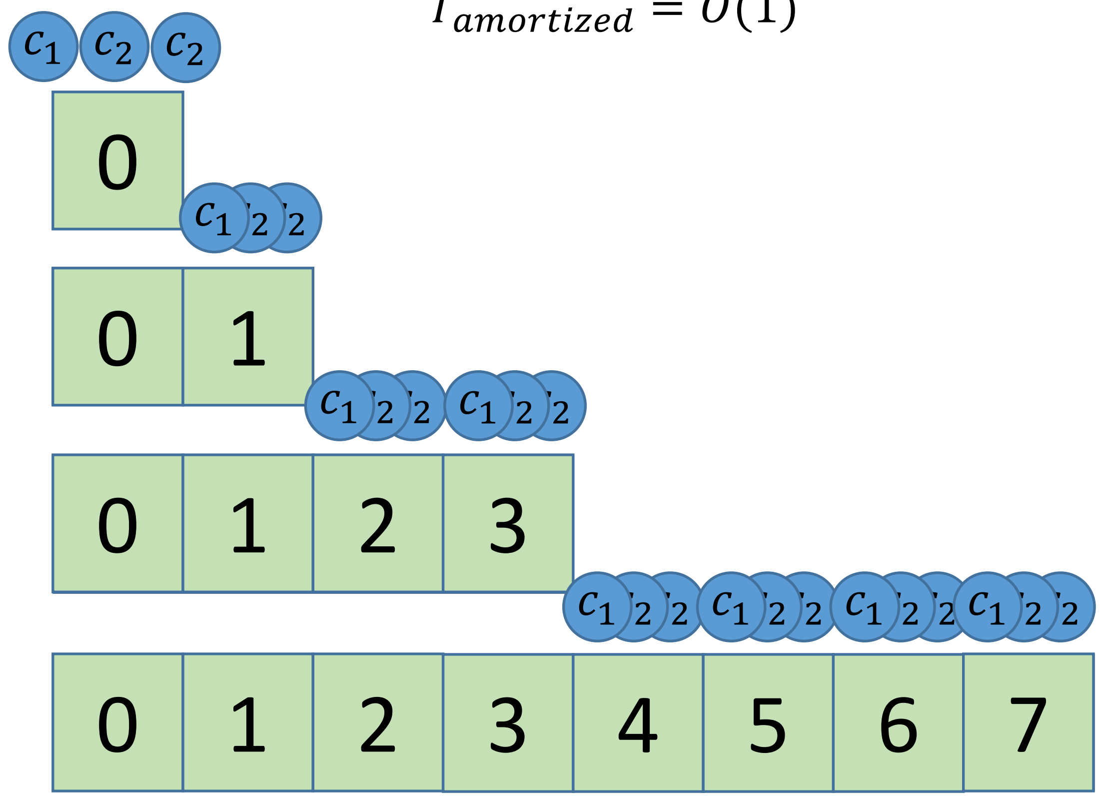
Complexity

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)
```

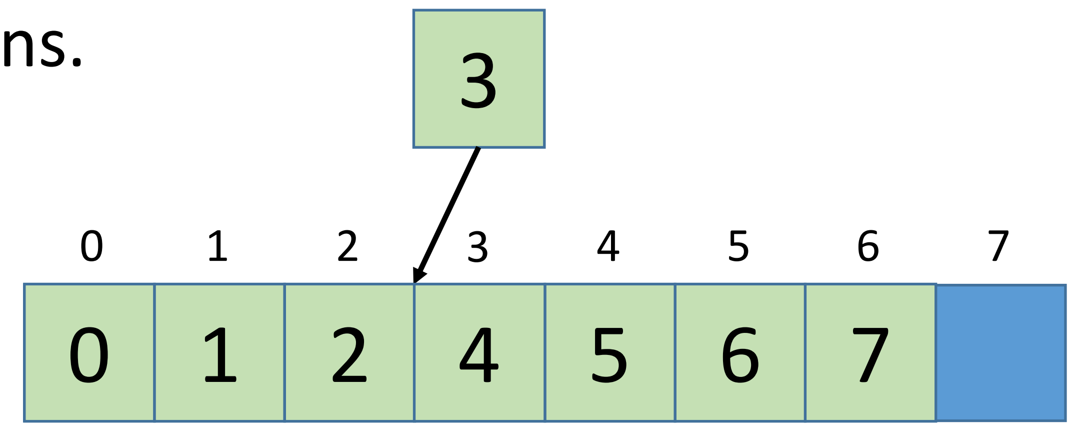


Vector

Insert, remove

What if we want to insert a value in the middle of the array, or remove value from the middle of array.

We need to shift elements. That will require $O(N)$ operations.

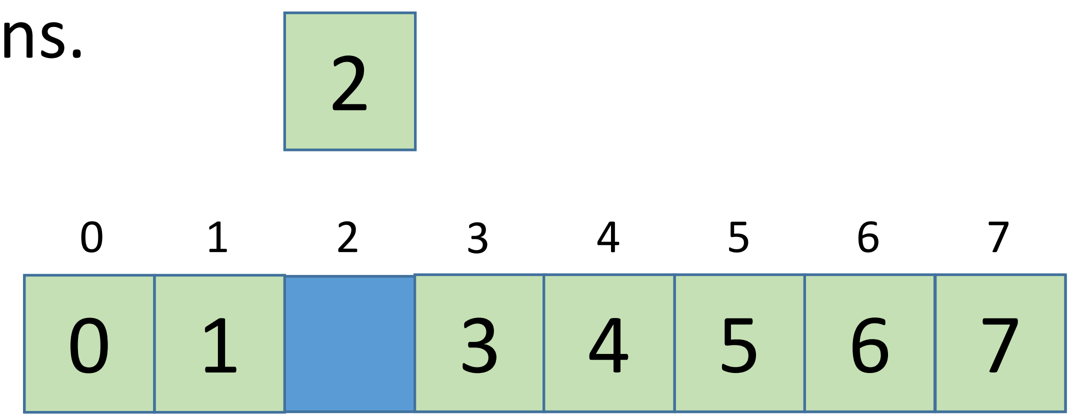


Vector

Insert, remove

What if we want to insert a value in the middle of the array, or remove value from the middle of array.

We need to shift elements. That will require $O(N)$ operations.



Vector

Set of operations

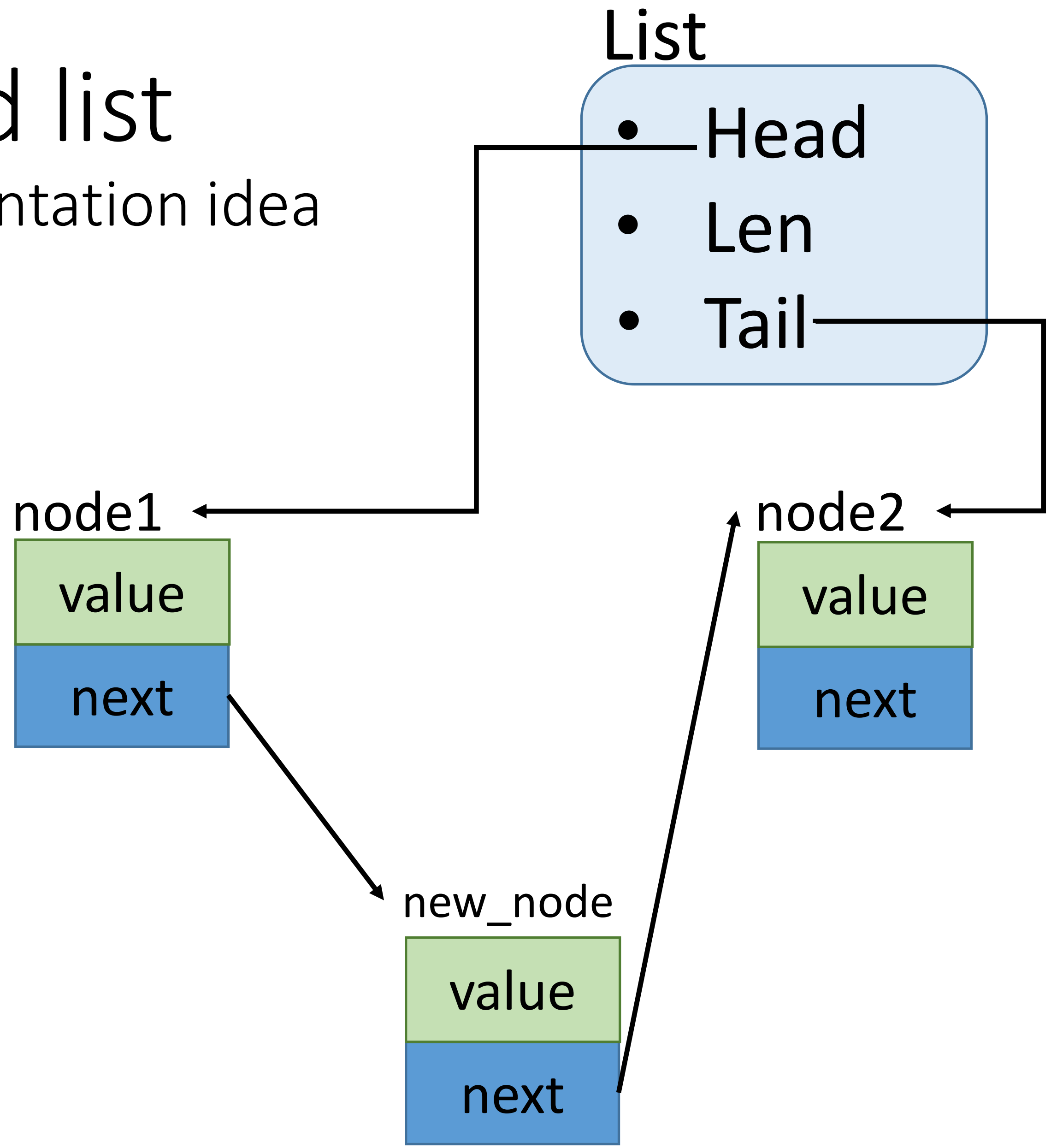
Operation	Complexity (for Vector)	Comments
push_back	$O(1)$	$O(1)$ is amortized complexity. Worst case for one operation is $\mathrm{O}(N)$.
pop_back	$O(1)$	We can just decrease len. That will remove right most value.
get/set (by index)	$O(1)$	As elements are stored contiguously in a whole piece of memory, 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 if we want to remove element from the middle.
push_front	$O(N)$	We can use insert operation for that.
pop_front	$O(N)$	The same as for push_front.

Basic data structures

- Static array
- Dynamic array (vector)
- **Linked List**
- Doubly Linked List

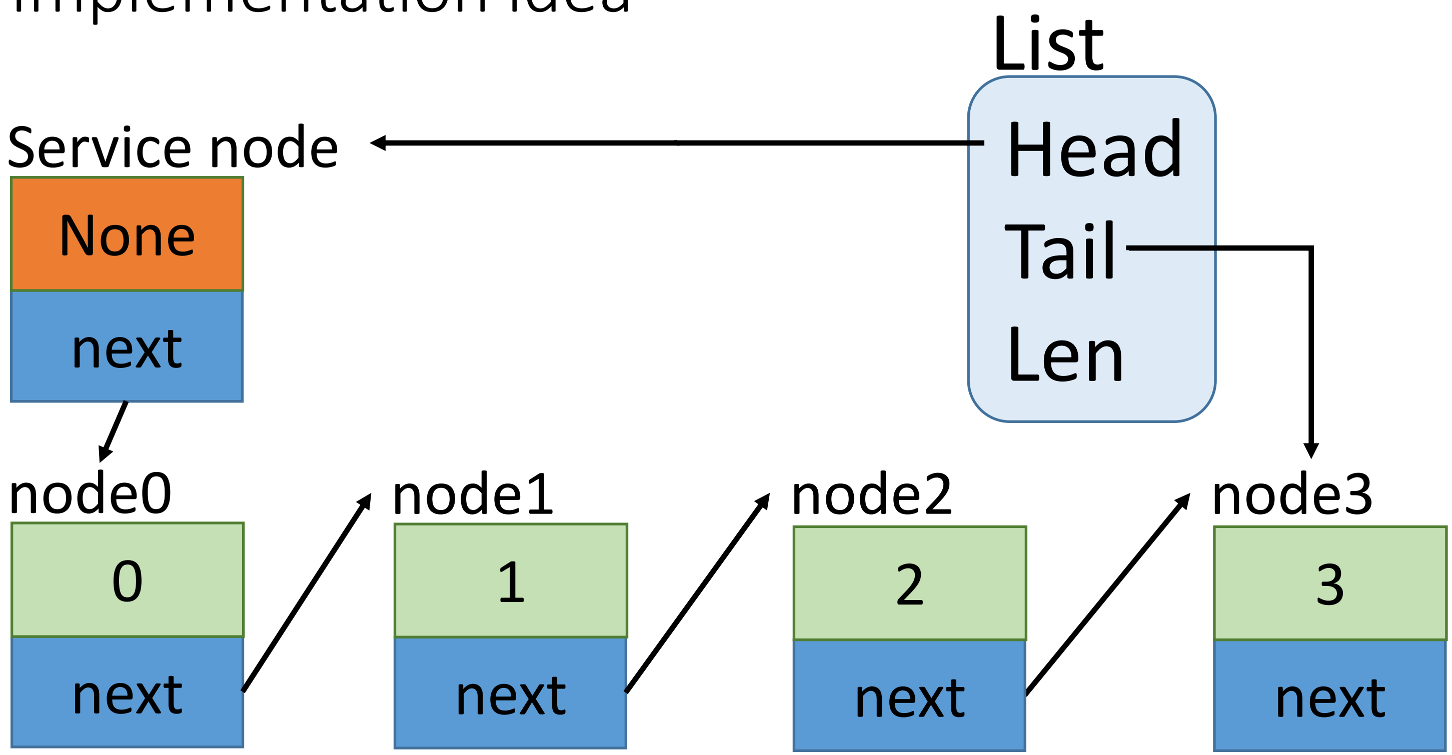
Linked list

Implementation idea



Linked list

Implementation idea




```
> push_front(1)
> push_front(0)
> push_back(3)
> insert(node1, 2)
> get(2)
> remove(node2)
```

Linked list

Implementation

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

class List:
  def __init__(self):
    # creating service node:
    self.head = ListNode(None, None)
    self.tail = self.head
    self.len = 0

  def insert(self, previous_node, val):
    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_front(self, val):
    return self.insert(self.head, val)
```

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

  def remove(self, node):
    prev_node = self.head
    while (prev_node is not None and
           prev_node.next != node):
      prev_node = prev_node.next
    if prev_node is not None:
      prev_node.next = node.next
      self.len -= 1
      if prev_node.next is None:
        self.tail = prev_node

  def pop_back(self):
    self.remove(self.tail)

  def pop_front(self):
    self.remove(self.head.next)

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

Linked list

Set of operations

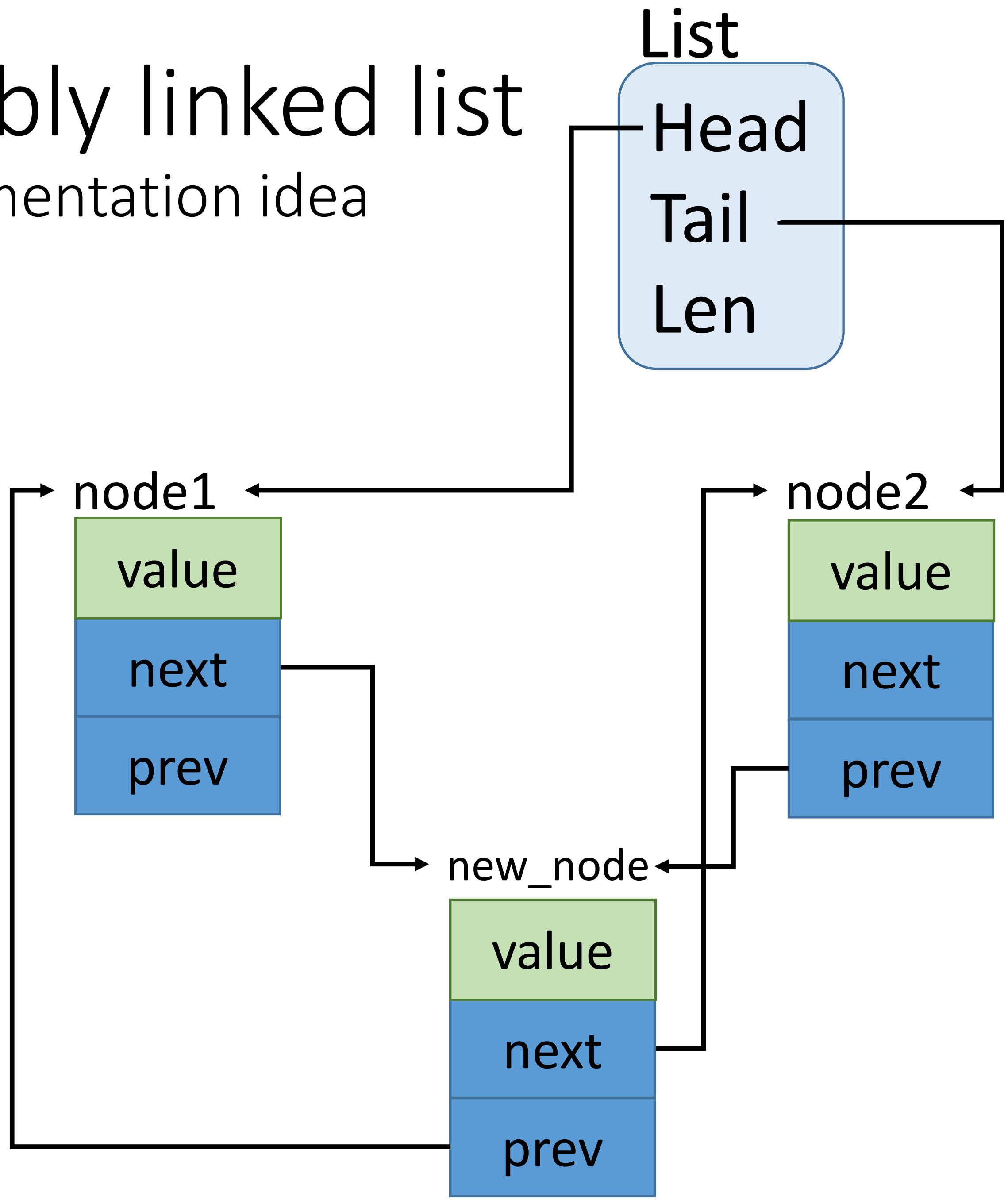
Operation	Complexity	Comments
push_back	$O(1)$	
pop_back	$O(\mathrm{N})$	To remove last element of list we need to get previous one. We can do that only iterating over all list starting from head.
get/set (by index)	$O(\mathrm{N})$	Getting/setting element by index requires iterating over all list. In worst case: $O(N)$ operations.
insert	$O(1)$	If we has a link to a list note to insert element next, it is $O(1)$. 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.

Basic data structures

- Static array
- Dynamic array (vector)
- Linked List
- **Doubly Linked List**

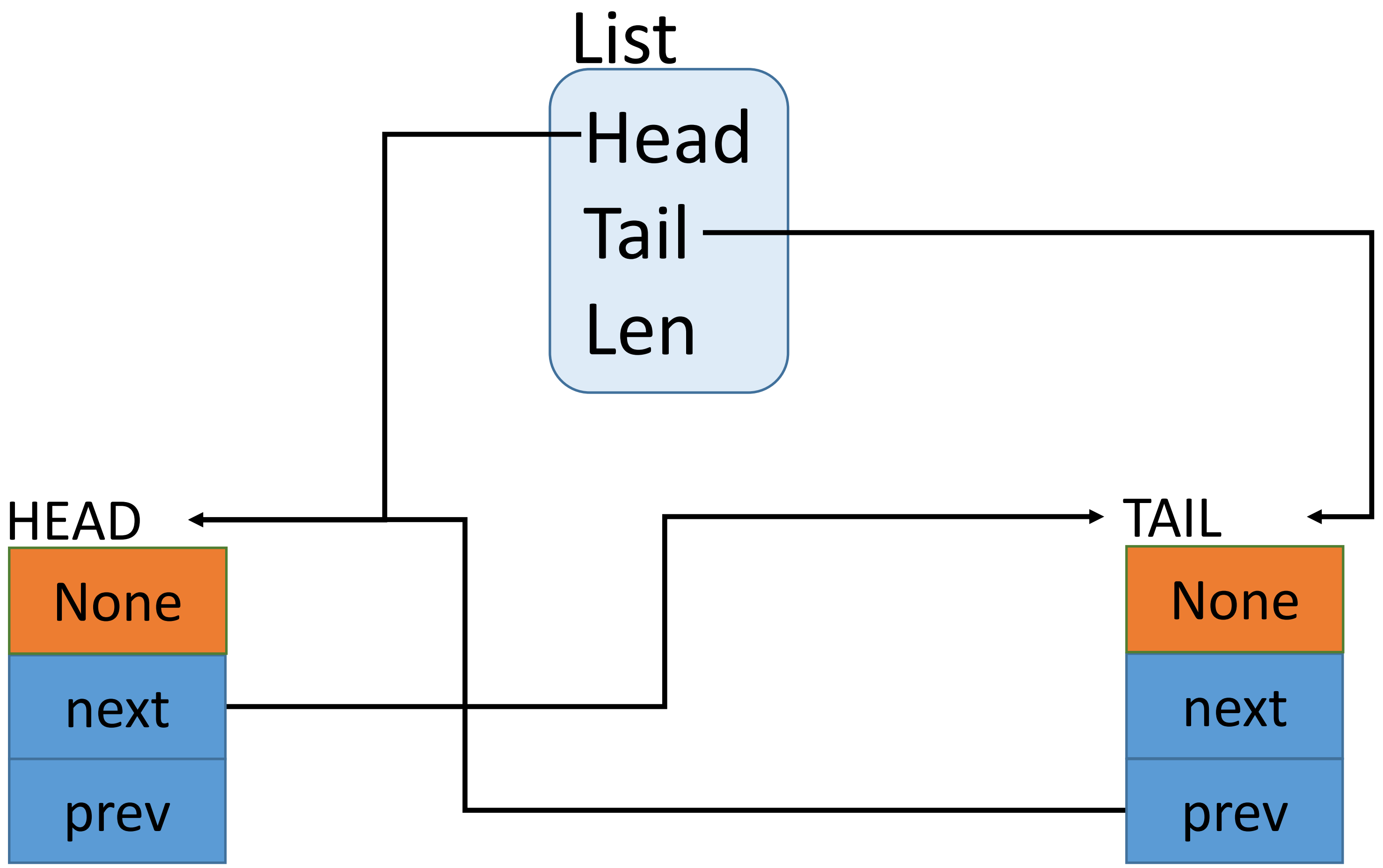
Doubly linked list

Implementation idea



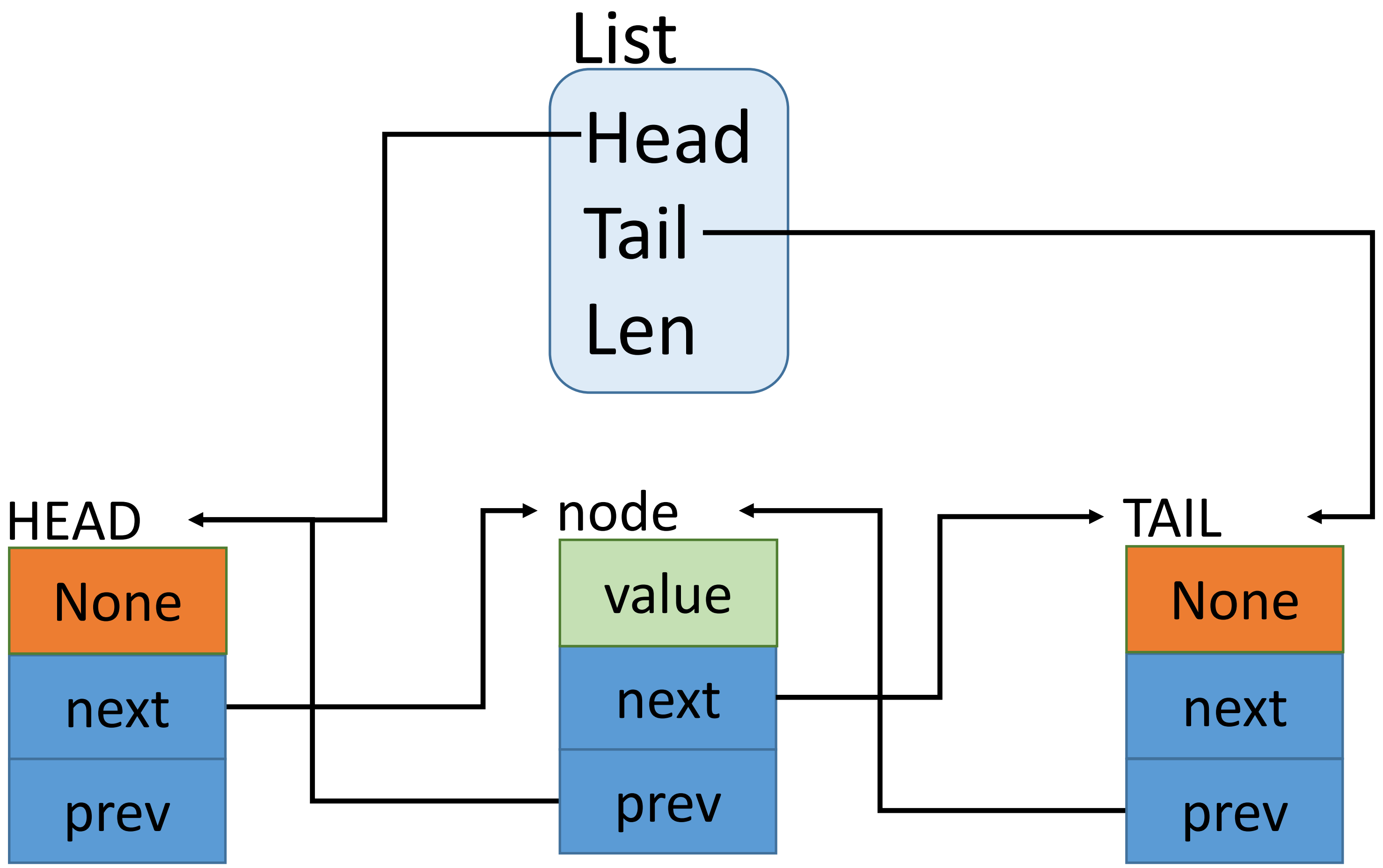
Doubly linked list

Hint



Doubly linked list

Hint



Doubly linked list

Complexity

Operation	**Complexity**	**Comments**
push_back	$O(1)$	
pop_back	$O(1)$	
get/set (by index)	$O(\mathrm{N})$	Getting/setting 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 $O(1)$. To insert by index, we need to perform get() first.
remove	$O(1)$	If we have a node to be deleted, it is $O(1)$. To remove by index, we need to perform get() first.
push_front	$O(1)$	
pop_front	$O(1)$	

Basic data structure interfaces

- **Stack (LIFO)**
- Queue (FIFO)
- Deque (Double Ended queue)

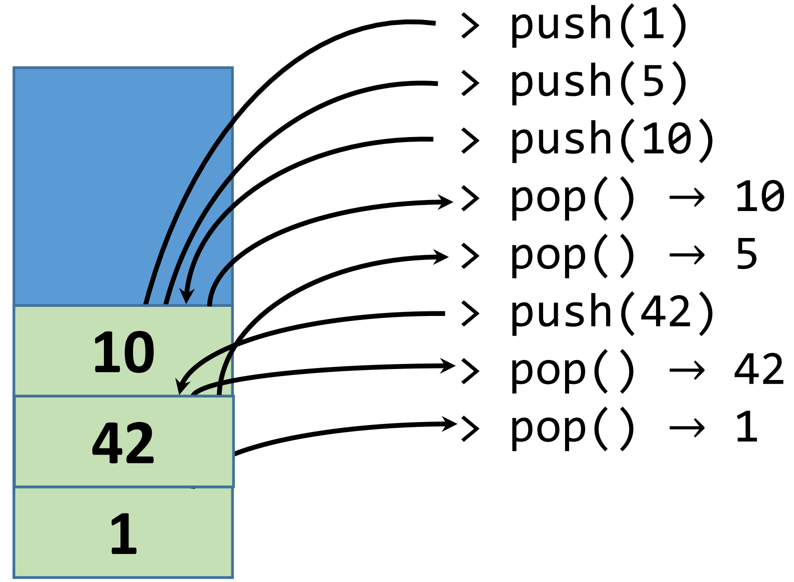
Stack (LIFO: last it, first out)

Definition

Data structure with following operations defined

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

Analogy:
Coins in the glass.



Basic data structure interfaces

- Stack (LIFO)
- **Queue (FIFO)**
- Deque (Double Ended queue)

Queue (FIFO: first in, first out)

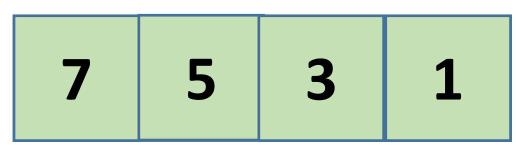
Definition

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
```

Analogy:
Queue in the shop.



Basic data structure interfaces

- Stack (LIFO)
- Queue (FIFO)
- **Deque (Double Ended queue)**

Deque (Double Ended queue)
Definition

Data structure with following operations defined:

- push_back(x) – place x in the queue as last element
- push_front(x) – place x in the queue as first element
- back() – return last element of the queue
- front() – return first element of the queue
- pop_back() – return last element of the queue and remove it
- pop_front() – return first element of the queue and remove it
- len() – return number of elements in queue

This data structure may be operated from both ends.

Conclusion

Conclusion

We can implement stack, queue and deque using:

- Static array
- Vector
- Linked List
- Doubly Linked List

Complexity for basic operations

Operation	Vector	Linked List	Doubly Linked List
push_back	$O(1)$	$O(1)$	$O(1)$
pop_back	$O(1)$	$O(\mathrm{N})$	$O(1)$
get/set (by index)	$O(1)$	$O(\mathrm{N})$	$O(\mathrm{N})$
insert	$O(N)$	$O(1)$	$O(1)$
remove	$O(N)$	$O(N)$	$O(1)$
push_front	$O(N)$	$O(1)$	$O(1)$
pop_front	$O(N)$	$O(1)$	$O(1)$

Python implementations

- 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!