ADT
Lists, Stacks, and Queues

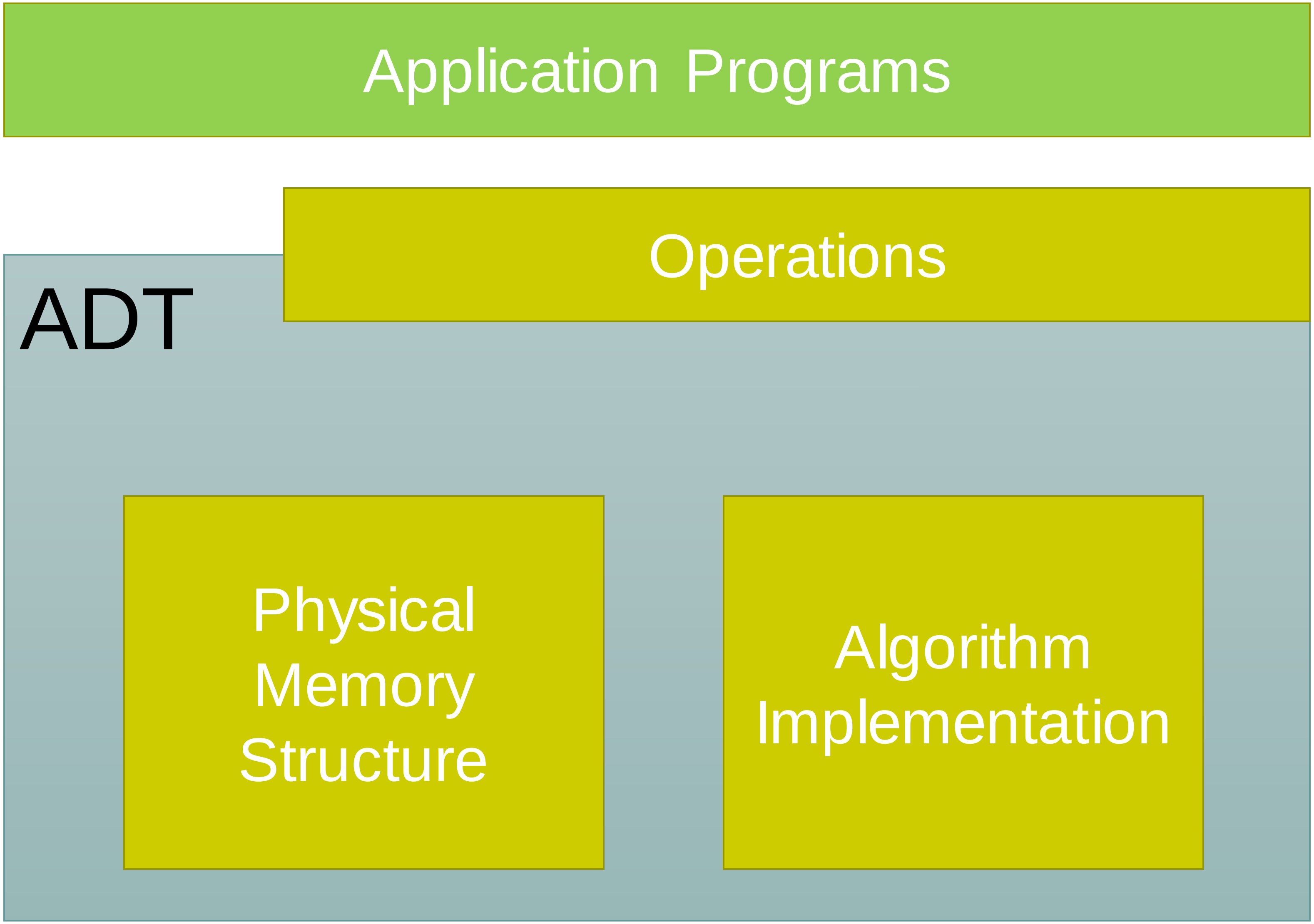
Instructor: Ahmed Eldawy

Objectives

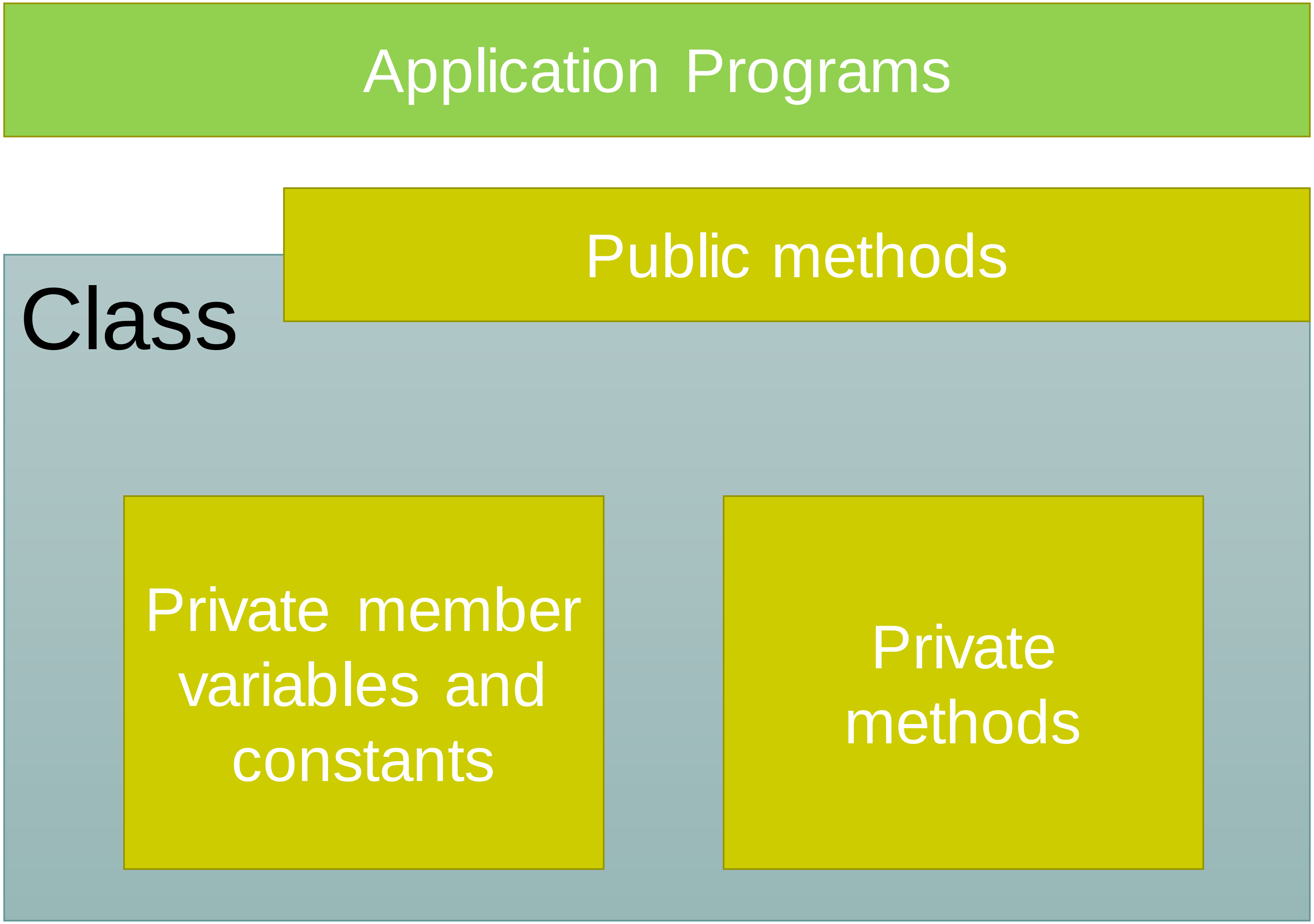
- Understand the importance of ADT
- Learn how to implement ADT in C++
- Recognize the difference between ADT definition and implementation
- Build an ADT for lists

Abstraction

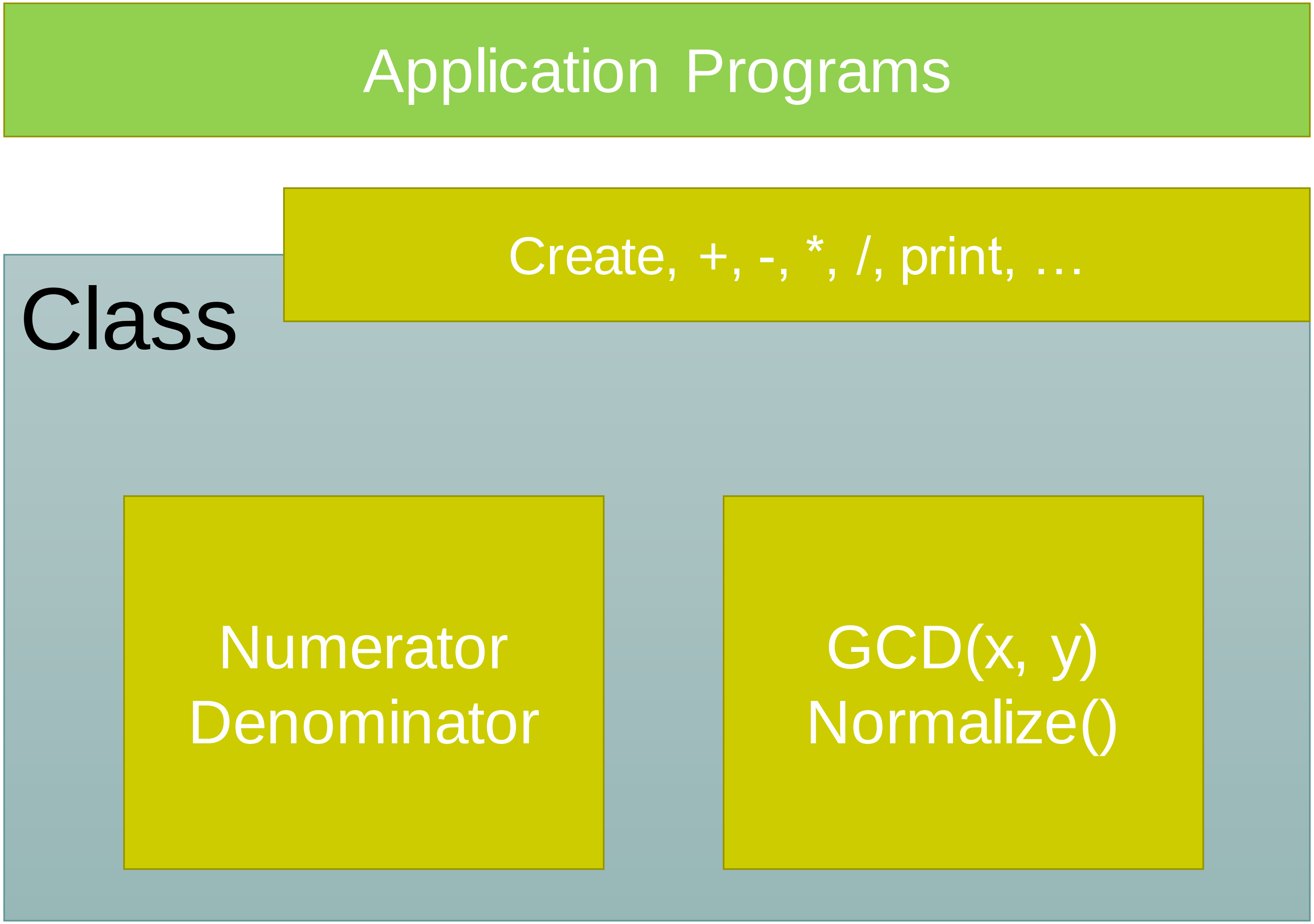
Abstract Data Types



Abstraction in C++



Example: Rational Numbers



ADT Design

- What is ADT design?
  - Defining the public interface
- Who designs an ADT?
  - You!
  - With your users
  - Sometimes, YOU are your own user

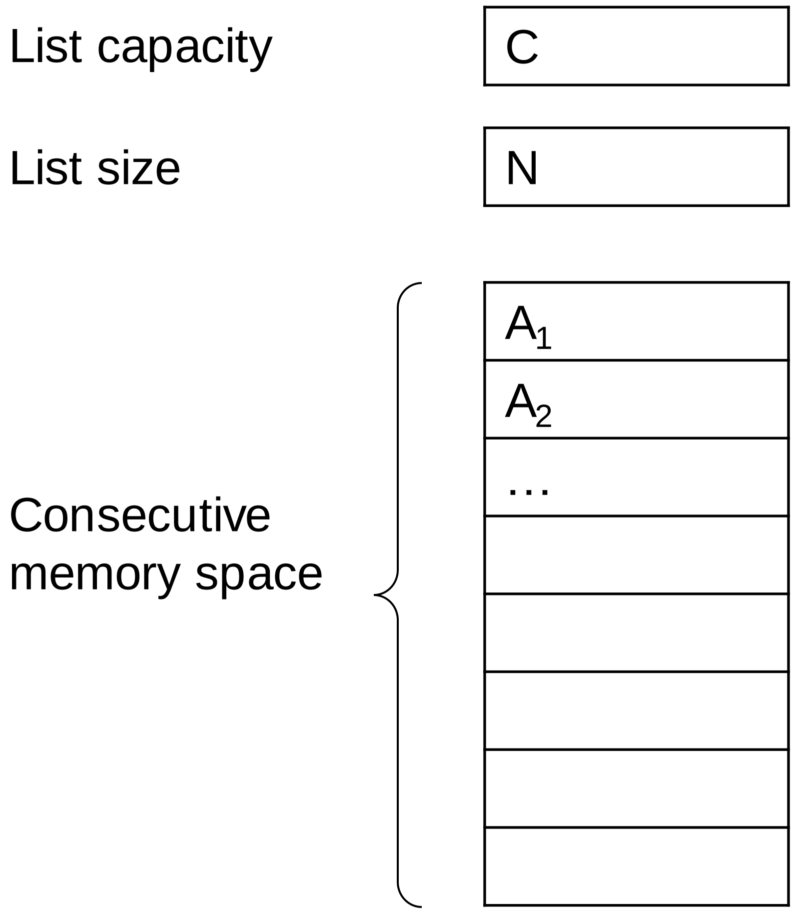
Lists

- List: A sequence of zero or more elements $A_1, A_2, \ldots, A_N$
- N: Size or length of the list
- A_1: First element
- A_N: Last element
- The order of items should be preserved

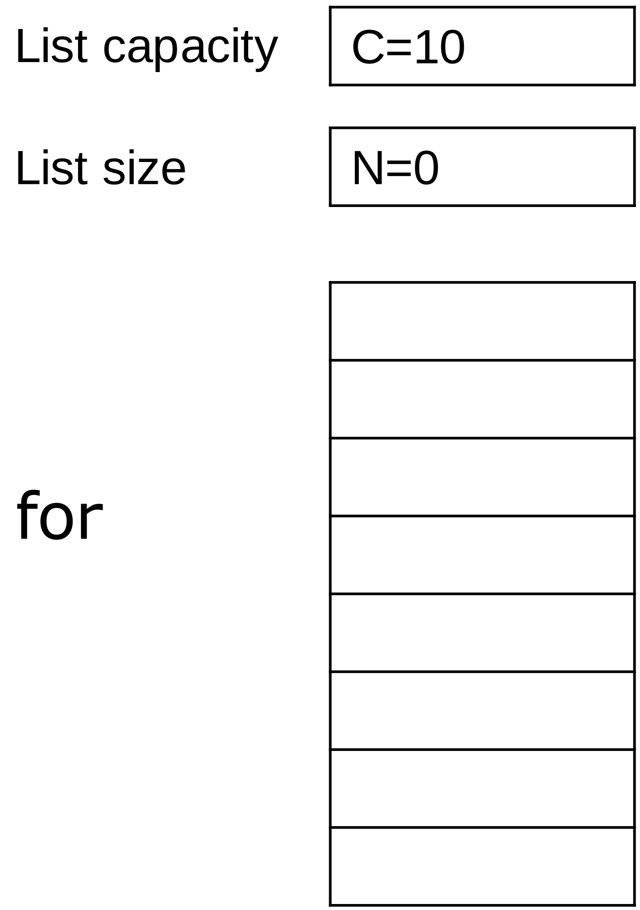
List ADT

- initialize(): Creates an empty list
- push_back(x): Appends the item x to the end of the list
- pop_back(): Removes the last element
- push_front(x): Prepends the item x at the beginning of the list
- pop_front(): Removes the first element
- insert(x, i): Inserts item x at position i
- erase(i): Deletes item at position i
- find(x): Finds the position of the element with value x
- size(): Returns the number of elements

Array Implementation of List

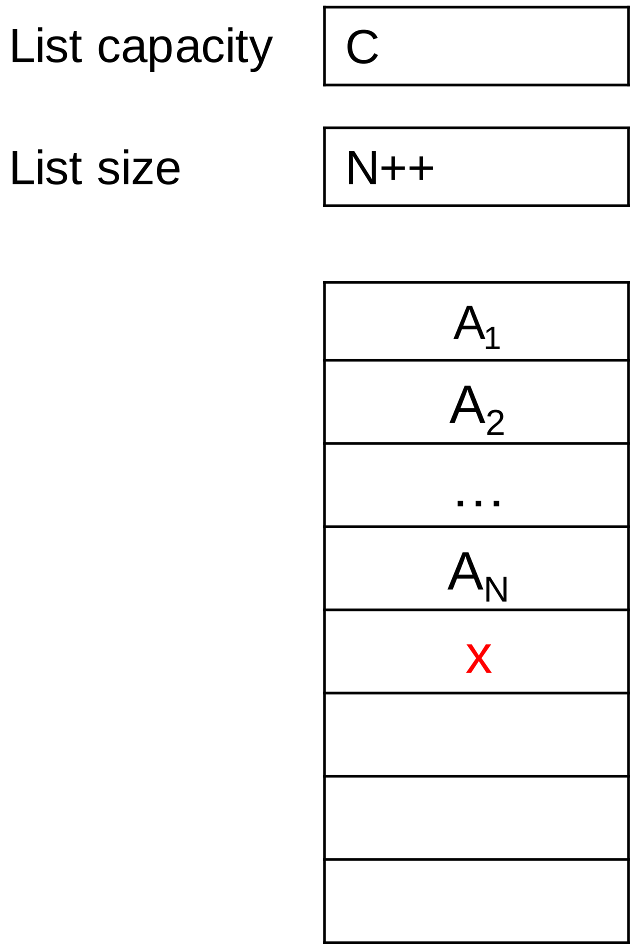


Initialize



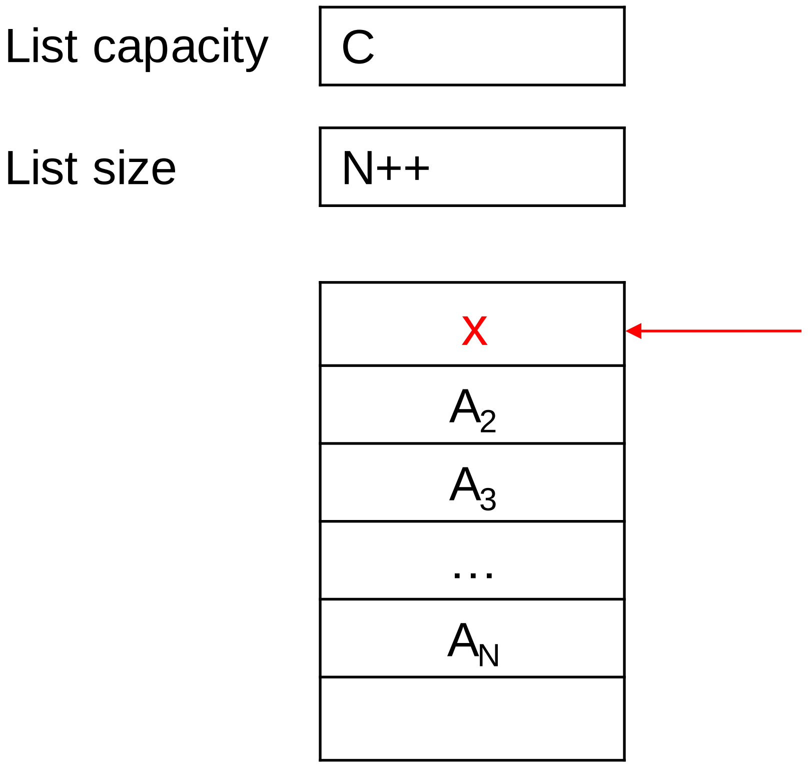

```
initialize() {
    C=10 // Initial capacity
    N=0 // Initial size
    Allocate a memory space for
    C elements
}
```

push_back(x)



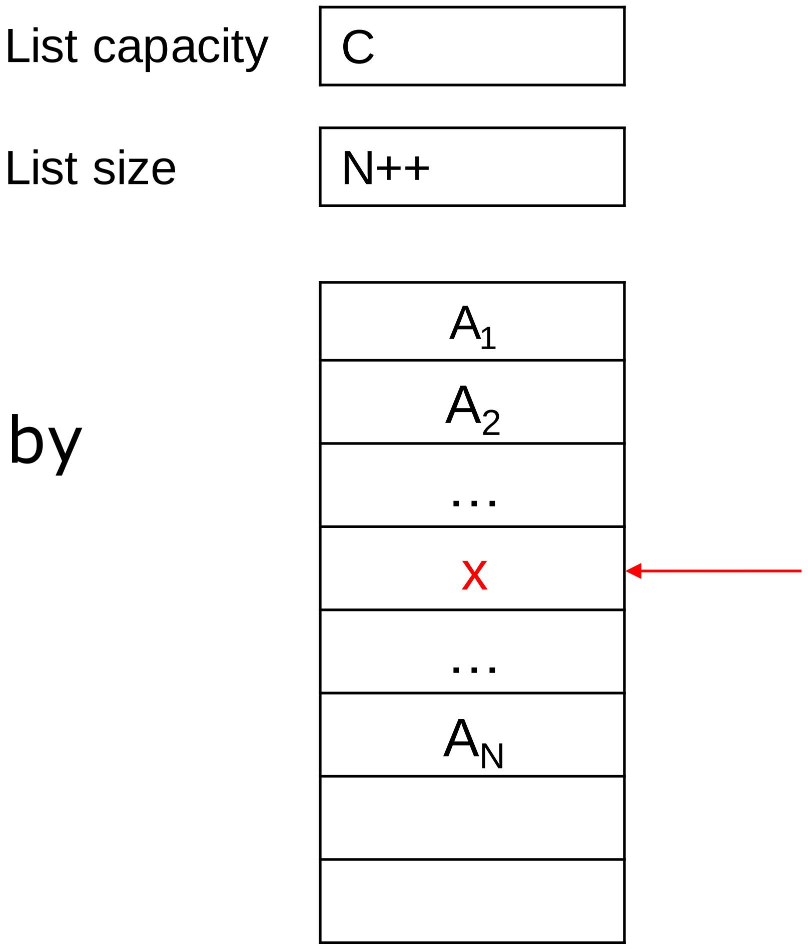

```
push_back(x) {
    if (N==C) the Expand A
    N = N + 1
    A_N = x
}
```

push_front(x)



push_front(x) {
    if (N==C) the Expand A
    Shift all elements A_1 to A_N by one position
    A_1 = x
    N = N + 1
}

insert(i, x)



insert(i, x) {

if (N==C) the Expand A

Shift all elements A_i to A_N by one position

$A_i = x$

$N = N + 1$

}

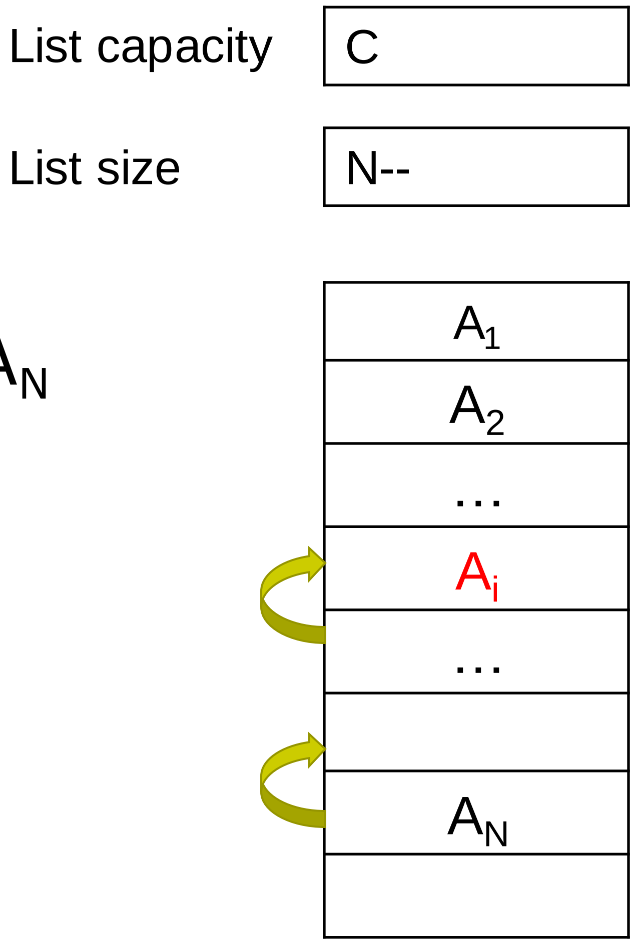
erase(i)

erase(i) {

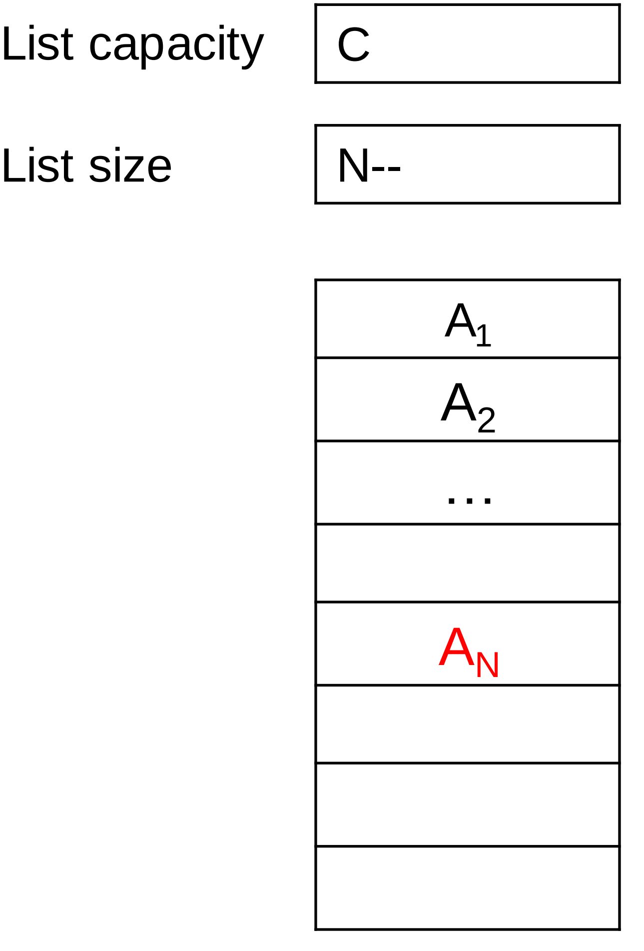
Shift all elements A_{i+1} to A_N by one position

N = N - 1

}

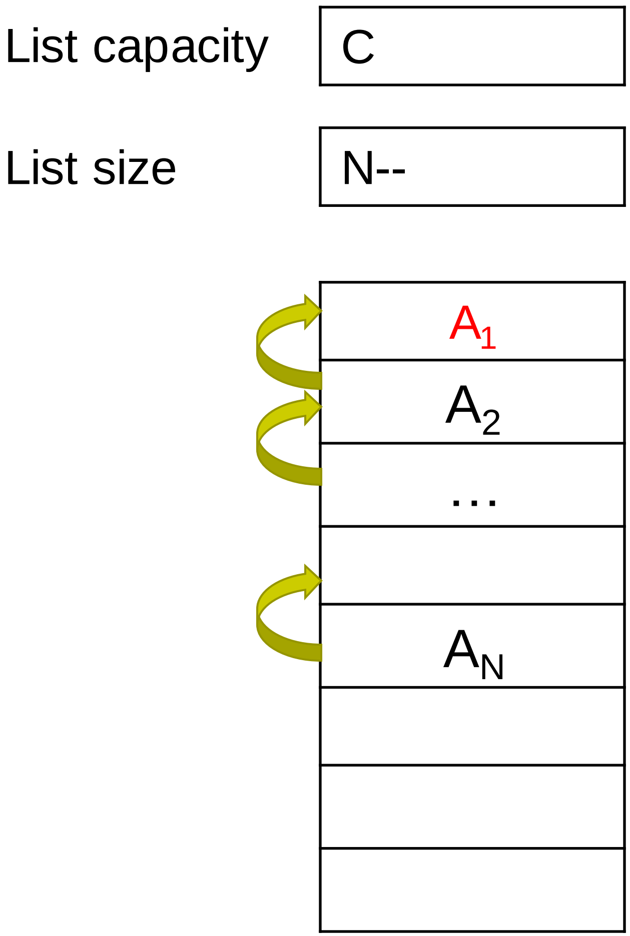


pop_back()



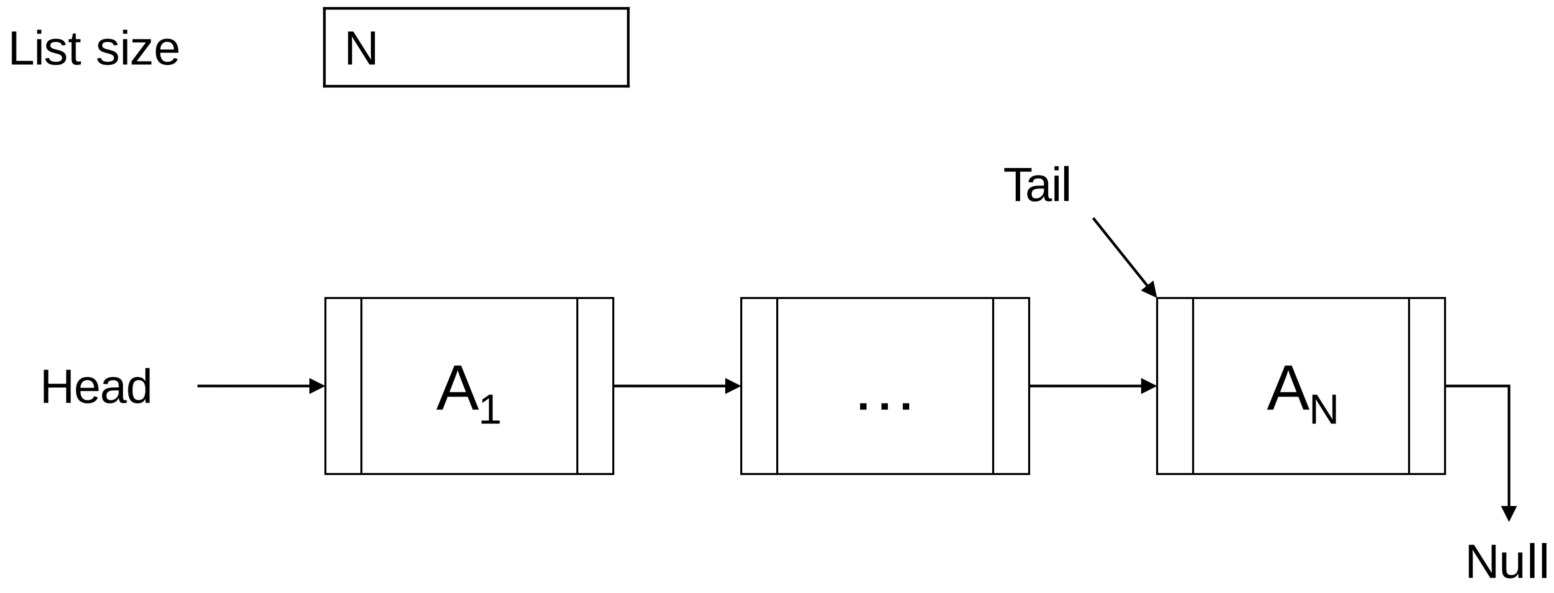

```
pop_back() {
    N = N - 1
}
```

pop_front()

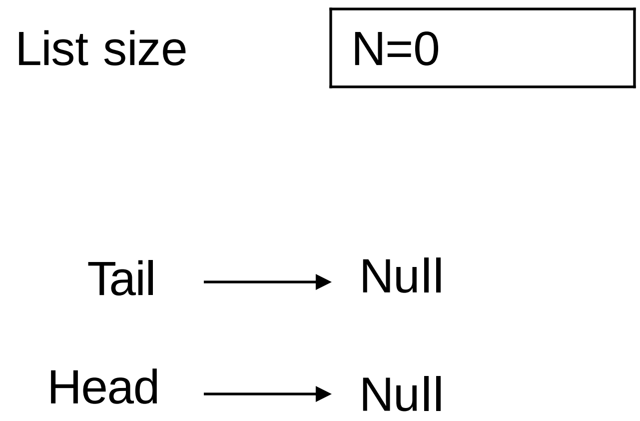


pop_front() {
    Shift all elements A_1 to A_N
    by one position
    N = N - 1
}

Linked-list Implementation



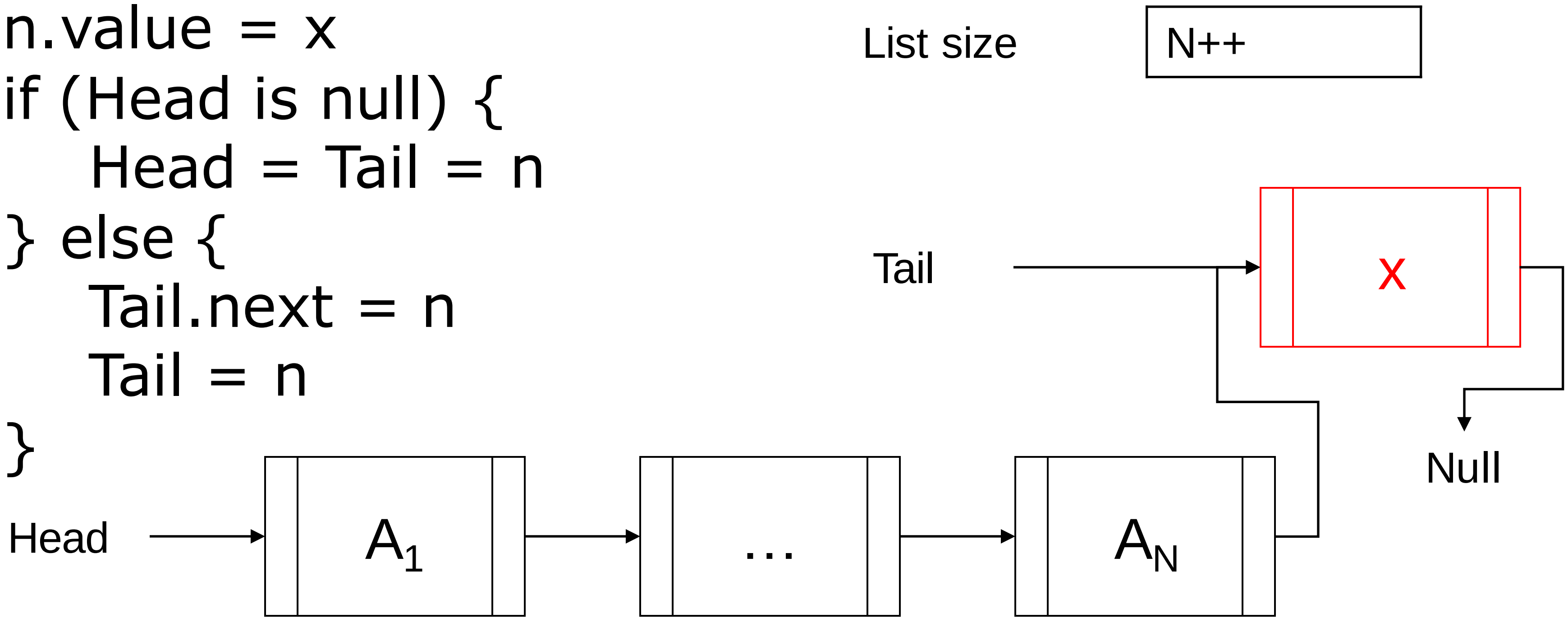
Initialize




```
initialize() {
    N=0
    Tail = Head = Null
}
```

push_back(x)

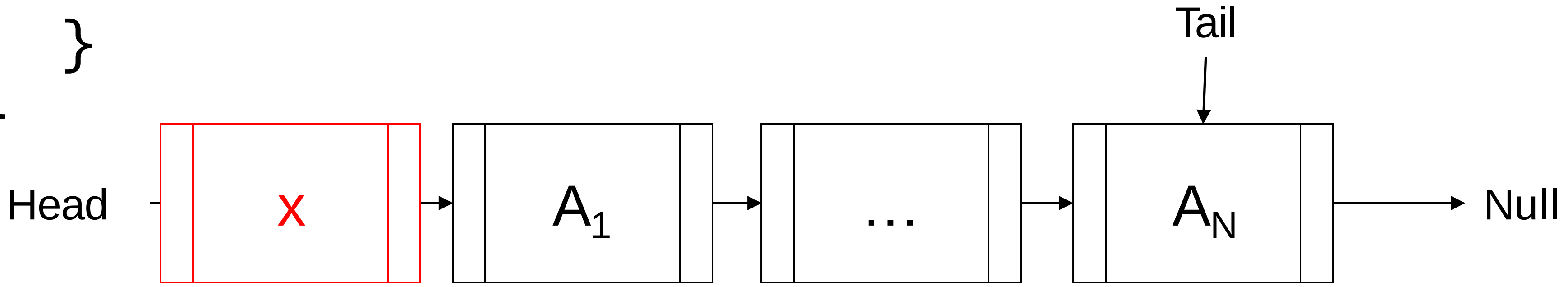
```
push_back(x) {
    N=N+1
    n = Allocate new node
    n.next = null
    n.value = x
    if (Head is null) {
        Head = Tail = n
    } else {
        Tail.next = n
        Tail = n
    }
}
```



push_front(x)

```
push_front(x) {
    N=N+1
    n = Allocate new node
    n.next = Head
    n.value = x
    if (Head is null) {
        Head = Tail = n
    } else {
        Head = n
    }
}
```

List size | N++

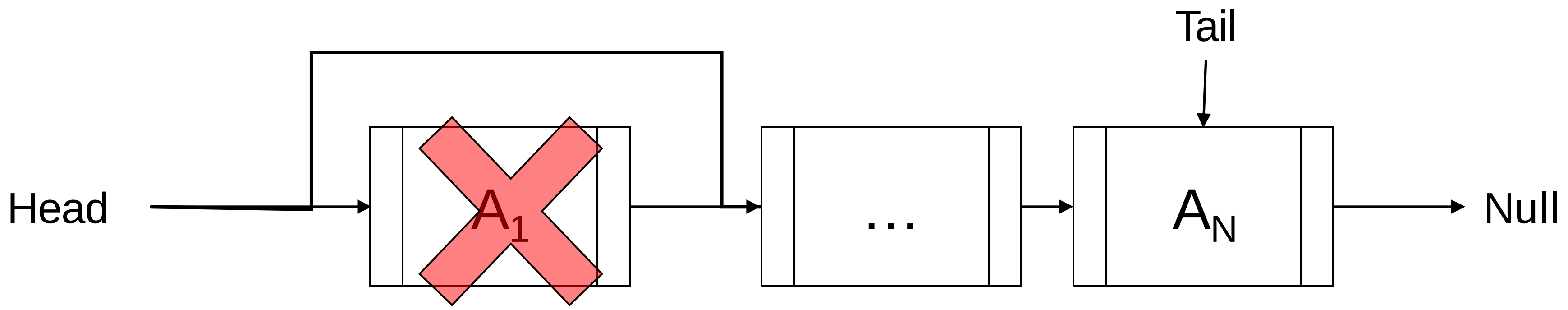


pop_front(x)

```
pop_front(x) {
    if N=0 then raise exception
    N=N-1
    old_node = Head
    Head = Head.next
    delete old_node
    // Are we done?
}
```

List size

N--



Array Vs Linked-list

Operation	Array	Linked-list
Initialize		
push_back		
push_front		
pop_back		
pop_front		
find		
erase		
clear		

Array Vs Linked-list

Operation	Array	Linked-list
Initialize	O(1)	O(1)
push_back	O(1)	O(1)
push_front	O(n)	O(1)
pop_back	O(1)	O(1)
pop_front	O(n)	O(1)
find	O(n)	O(n)
erase	O(n)	O(n)
clear	O(1)	O(n)