Abstract Data Type

Example

Leetcode 217 (easy, Blind75): Contains duplicate

Given an array of n numbers, determine if any value appears at least twice.

How can we solve this problem?

Values + Operations

Say the values in the array are $a_0, a_1, \ldots, a_{n-1}$.

Natural idea: iterate over the array and for each value check if we have seen it before.

Basic task: Given a_i, is it equal to any of $a_0, \ldots, a_{i-1}$, the values we have already seen?

Double For Loop

We could accomplish this with a double for loop:

```
bool containsDuplicate(const std::vector<int>& arr) {
    for(int i = 0; i < arr.size(); ++i) {
        for(int j = 0; j < i; ++j) {
            if(arr[i] == arr[j]) {
                return true;
            }
        }
    }
    return false;
}
```

https://godbolt.org/z/fvo5GjE9e

Is there a better way?

Data Structure

Basic task: Given a_i, is it equal to any of $a_0, \ldots, a_{i-1}$, the values we have already seen?

In the double for loop solution, we leave the data in its original form.

To check if we have seen a value we then iterate through all the seen values.

Is there some other we can organize the values we have already seen so far to speed up this check?

In other words, can we put the values seen in a data structure to make lookup faster?

Contains and Insert

Let's be more formal about the operations we want to perform on our data structure.

- We want to check if it contains a given element.
- We want to insert elements we have seen into the data structure.

```
for(auto val : arr) {
    if(valuesSeen.contains(val)) {
        return true;
    }
    valuesSeen.insert(val);
}
return false;
```

Contains and Insert

```
for(auto val : arr) {
    if(valuesSeen.contains(val)) {
        return true;
    }
    valuesSeen.insert(val);
}
return false;
```

This is a generic template to solve Contains Duplicate that we can instantiate in different ways.

The double for loop is also an instantiation of this template.

Abstract Data Type

We have now given an Abstract Data Type (ADT) that will solve the problem.

An ADT is a collection of values and a specification of operations that can be performed on them.

For the Contains Duplicate problem we want an abstract data type with the operations `contains` and `insert`.

ADT vs. Data Structure

An ADT is like a user's manual. It specifies what a user can do, but says nothing about how the operations are implemented.

A data structure is a concrete implementation of an ADT.

To instantiate our algorithm for Contains Duplicate, problem we have to choose a data structure to implement the ADT we need.

The efficiency of our solution depends on the data structure we choose.

Problem Solving

I find it is often useful to design algorithms using abstract data types.

As you are thinking about a problem imagine what "special powers" would allow you to solve it.

These special powers form the operations of an abstract data type.

Then you can see if there is a data structure that can efficiently implement this abstract data type.

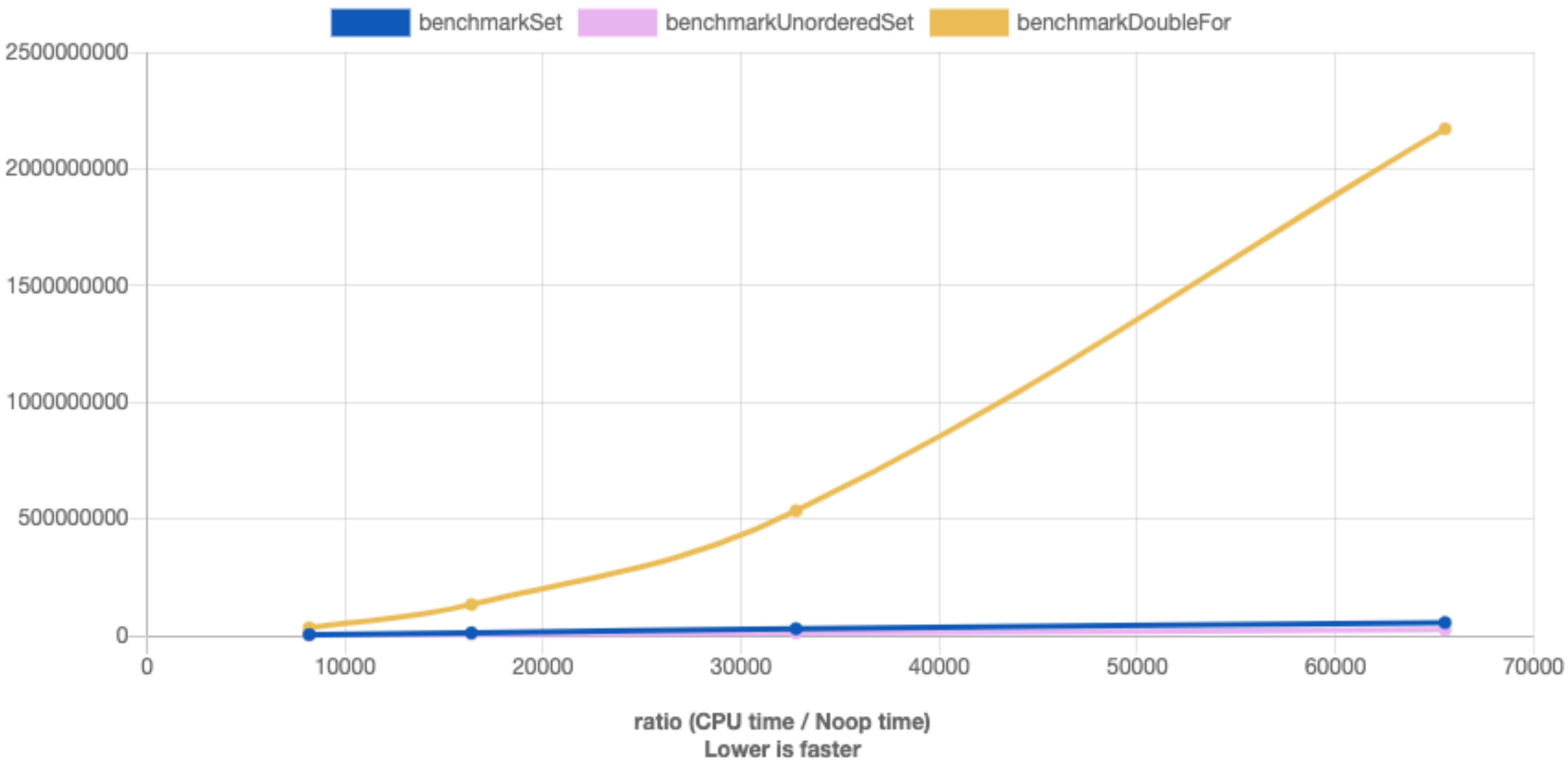
Library of ADTs

Throughout this course you will see the most important abstract data types and how to implement them to develop your own library of ADTs.

This will help you design algorithms by knowing what "special powers" are possible.

In particular, you will learn about data structures like balanced binary search trees and hash tables that are good for the Contains Duplicate problem.

Contains Duplicate



https://quick-bench.com/q/dN0VNDYP6xMIA577rkW8e-TppTA

Fixed Size Array

Fixed-Size Array ADT

In a fixed-size array, we must specify the maximum number of items it can hold on initialization.

$A \leftarrow \texttt{Array(n)}$ creates an empty array that can hold n elements.

We can perform two operations on a fixed size array:

$A.\texttt{get(i)}$ for $0 \leq i < n$ returns the i^{th} item in the array.

$A.\texttt{set(i, x)}$ for $0 \leq i < n$ set the value of the i^{th} item to be x .

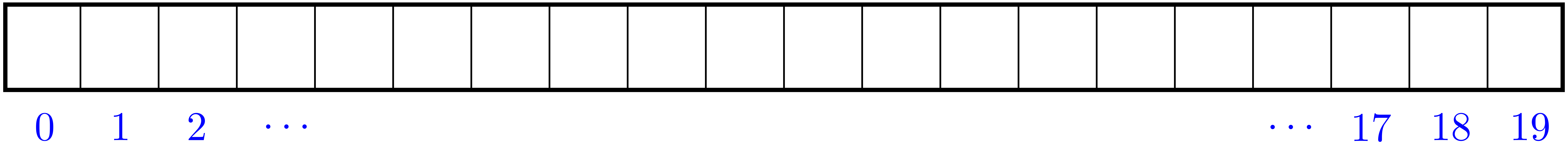
Model of Computation

In this course, we want to talk about the complexity of algorithms and data structures.

How much time does it take to get the i^{th} item of an array?

In order to do this, we have to say something about the model of computation we are using.

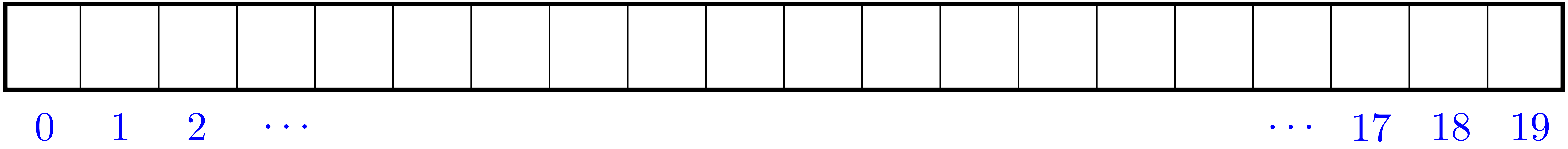
Model of Computation



Imagine all the memory of your computer as a long tape divided into small chunks of memory (think 8 bits) called words.

Each word has an integer address.

Model of Computation

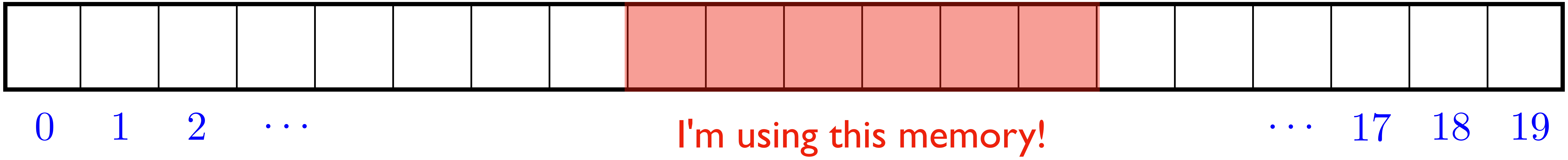


1) Random Access: we can read/write to any address in constant time.

2) We can allocate or free a block of memory in constant time.

3) We can perform arithmetic operations (plus, minus, times, divide) on addresses in constant time.

Model of Computation

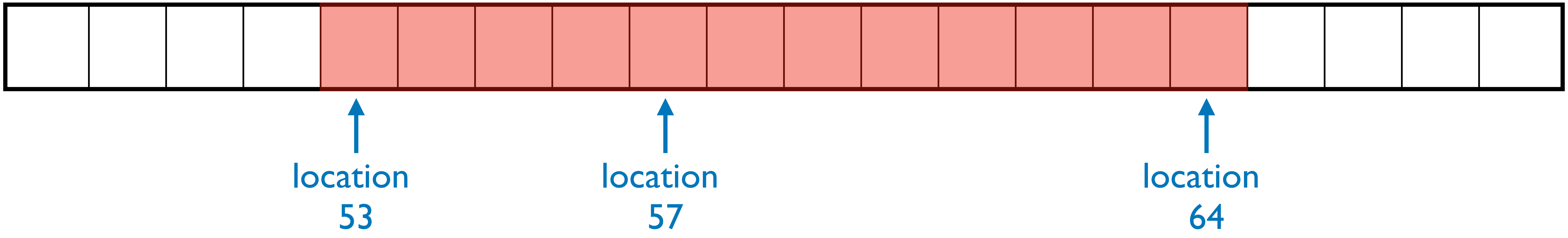


1) Random Access: we can read/write to any address in constant time.

2) We can allocate or free a block of memory in constant time.

3) We can perform arithmetic operations (plus, minus, times, divide) on addresses in constant time.

Implementation of a fixed-size array

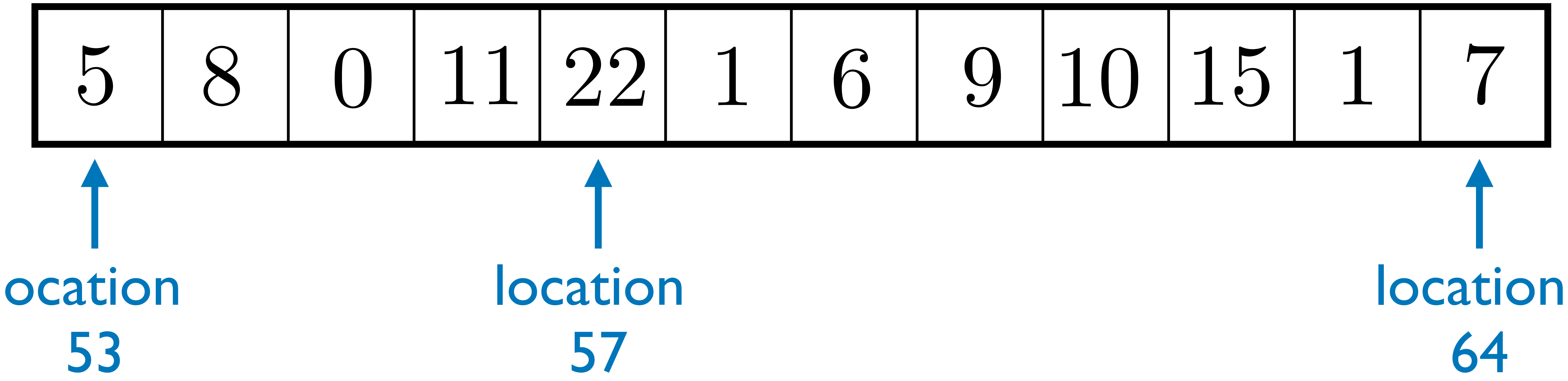


$A \leftarrow \texttt{Array(n)}$ creates an empty array that can hold n elements.

We allocate a contiguous block of memory large enough to hold n items.

This takes constant time by rule 2.

Implementation of a fixed-size array



We store the address of the first element of the array $\&\texttt{arr}[0]$.

From the address of the first element we can compute the address of $\texttt{arr}[\texttt{i}]$ with one addition and one multiplication.

$$\&\texttt{arr}[0] + \texttt{i} * \texttt{sizeof}(\texttt{type})$$

We can get/set $\texttt{arr}[\texttt{i}]$ in constant time in the RAM model.

Summary

A fixed-size array supports the following operations.

$A \leftarrow \texttt{Array(n)}$ creates an empty array that can hold n elements.

$A.\texttt{Get(i)}$ for $0 \leq i < n$ returns the i^{th} item in the array.

$A.\texttt{Set(i, x)}$ for $0 \leq i < n$ set the value of the i^{th} item to be x .

All of these operations take constant time.

Resizable Array

Resizable Array

What is the drawback of a fixed-size array?

You have to know an upper bound on the number of elements in advance.

If we are reading in data from a file, for example, we may not know this.

A resizable array allows us to add as many elements as we want, up to the memory limit of the computer.

As in a fixed-size array, in a resizable array we can access/modify any element via its index in constant time.

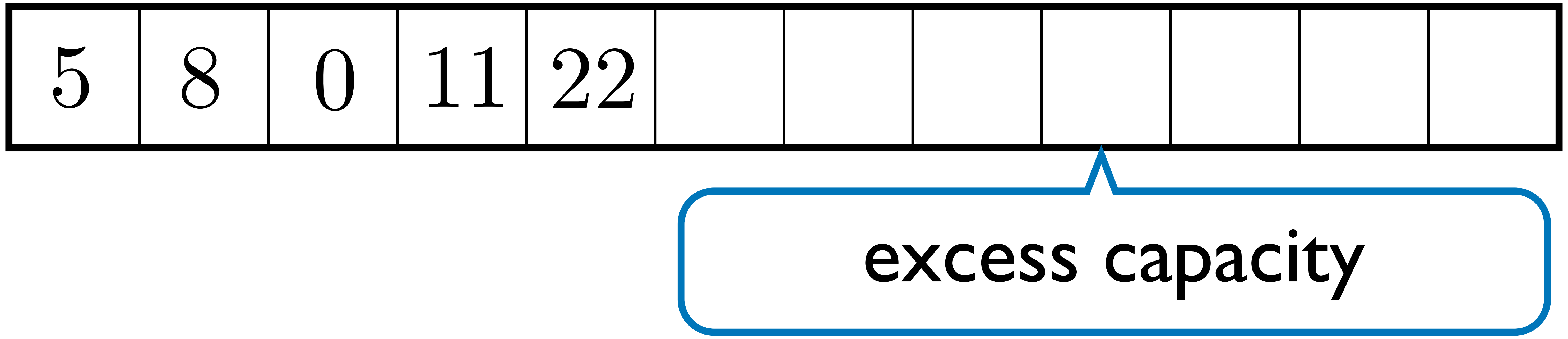
Resizable Array ADT

$A \leftarrow \texttt{Vector}()$	Creates an empty resizable array
$A.\texttt{push\_back}(\texttt{x})$	Add x to the end of A.
$A.\texttt{pop\_back}()$	Remove the last element of A.
$A.\texttt{size}()$	Return the number of elements in A.
$A.\texttt{get}(\texttt{i})$	For $0 \leq i < A.\texttt{size}()$, returns the i^{th} item in A.
$A.\texttt{set}(\texttt{i}, \texttt{x})$	For $0 \leq i < A.\texttt{size}()$, set the value of the i^{th} item to be x.

Implementing a Resizable Array

We implement a resizable array using fixed-size arrays.

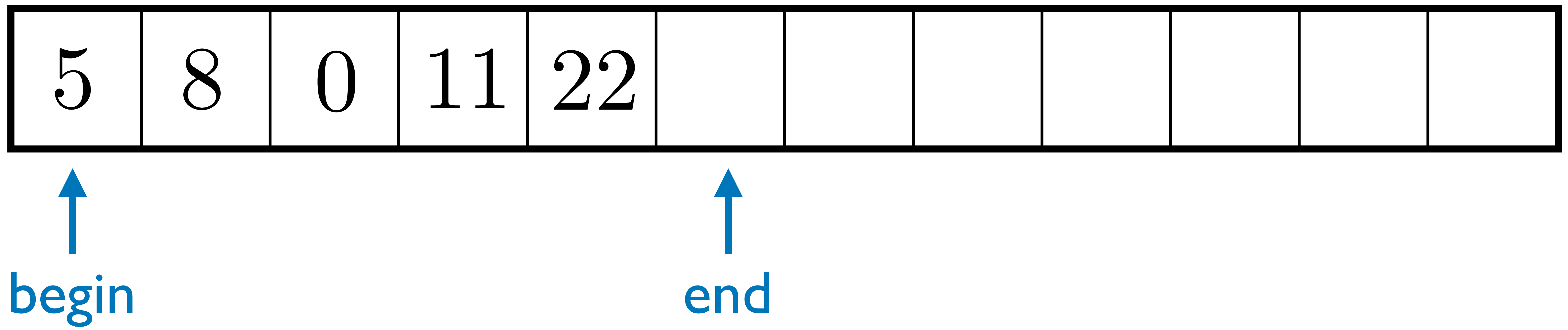
Basic idea: Allocate a fixed-size array initially. As elements are inserted, fill from left to right.



Here $A.\mathtt{size}()$ is 5. The remaining slots are yet to be used.

Implementing a Resizable Array

Basic idea: Allocate a fixed-size array initially. As elements are inserted, fill from left to right.

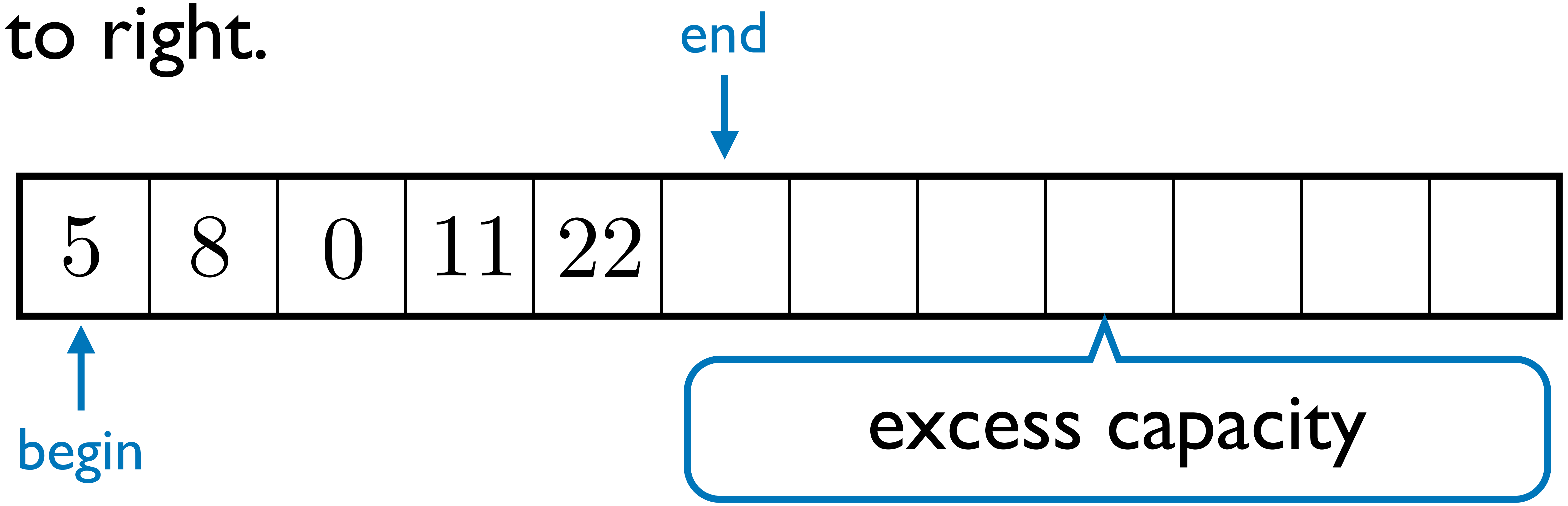


We store the address of the beginning of the array and one past the last element.

This tells us where to push back a new element, and makes it easy to compute the size.

Implementing a Resizable Array

Basic idea: Allocate a fixed-size array initially. As elements are inserted, fill from left to right.



While there is excess capacity, adding an element takes constant time.

No Capacity

The interesting question is how to do push back when there is no excess capacity.

5	8	0	11	22	1	6	9	10	15	1	7

Say now we want to `push_back` 13 but the fixed-size array is full.

What can we do in this situation?

No Capacity

5	8	0	11	22	1	6	9	10	15	1	7

Say now we want to insert 13 but the fixed-size array is full.

We allocate a new and larger fixed-size array (constant time by Rule 2).

No Capacity

5	8	0	11	22	1	6	9	10	15	1	7

Then we copy all the elements, and the new element 13, into the new array.

5	8	0	11	22	1	6	9	10	15	1	7	13				

Finally, we free the memory of the original array (constant time by Rule 2).

Complexity

Question: How much time does this method take to insert an element when there is no capacity?

It takes constant time to allocate the new array and free the old array.

The expensive part of the operation is to copy the elements from the old array to the new array.

This operation takes time proportional to the number of elements in the old array.

Complexity

Question: How should we choose the size of the new array?

There is a trade-off here between time and memory.

Transferring to a new array is a time-expensive operation.
We do not want to do it too often.

This suggest choosing the new array to be large.

On the other hand, the larger the new array, the more potentially "wasted" extra memory we use.

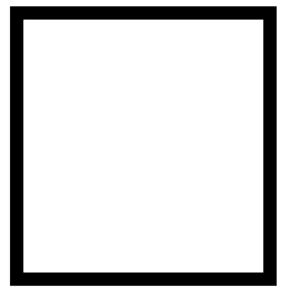
Array Doubling

A common solution to this trade-off is to set the size of the new array to be double the size of the previous one.

> Memory: this way the amount of memory we use is at most twice the minimum amount needed.
>
> Time: let's look at how much total time we use over a sequence of insertions.

As a proxy for time, let's count the number of "copy" operations as we successively push back 5, 8, 0, 11, 22.

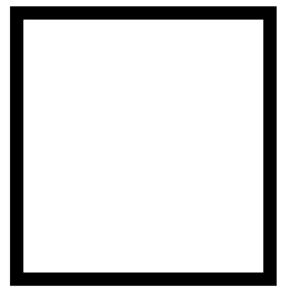
We assume that the initially allocated fixed size array has size one.



initial fixed-size array.

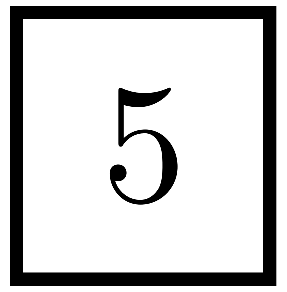
As a proxy for time, let's count the number of "copy" operations as we successively push back 5, 8, 0, 11, 22.

We assume that the initially allocated fixed size array has size one.



initial fixed-size array.

push back 5:

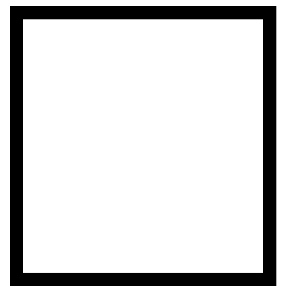


excess capacity, no copies.

0 copies

As a proxy for time, let's count the number of "copy" operations as we successively push back 5, 8, 0, 11, 22.

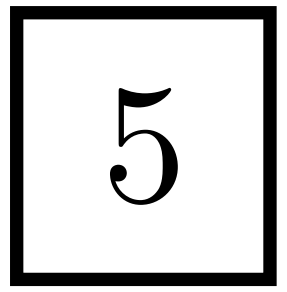
We assume that the initially allocated fixed size array has size one.



initial fixed-size array.

push back 5:

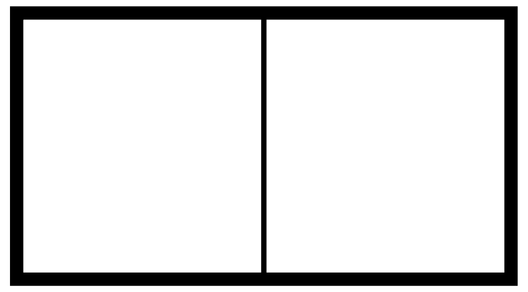
0 copies



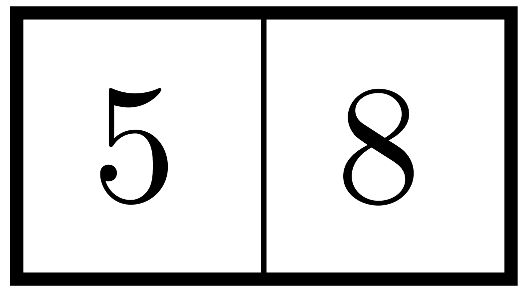
excess capacity, no copies.

push back 8:

1 copy



no capacity, allocate array of size 2.



copy 5 over, insert 8.

push back 8:

5	8

1 copy

push back 8:

5	8

1 copy

push back 0:

no capacity, allocate array of size 2.

2 copies

5	8	0	

copy over 5 and 8, insert 0.

push back 8:

5	8

1 copy

push back 0:

no capacity, allocate array of size 2.

2 copies

5	8	0	

copy over 5 and 8, insert 0.

push back 11:

5	8	0	11

excess capacity, no copies.

0 copies

push back 11:

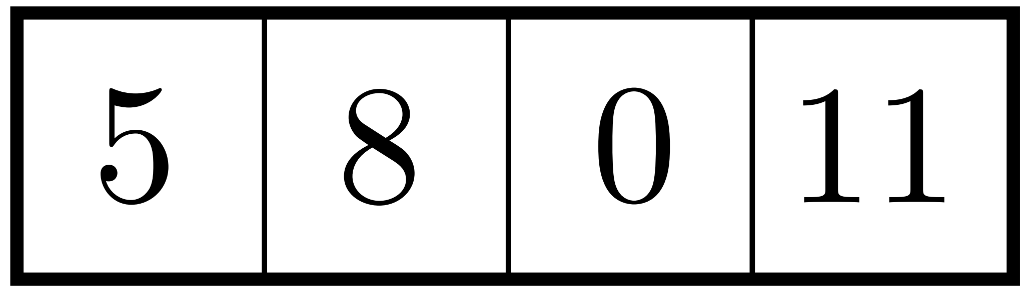
5	8	0	11

excess capacity, no copies.

0 copies

push back 11:

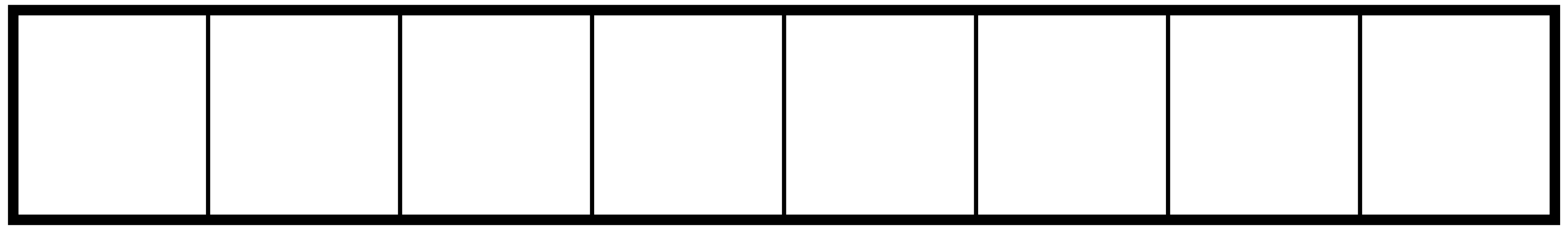
0 copies



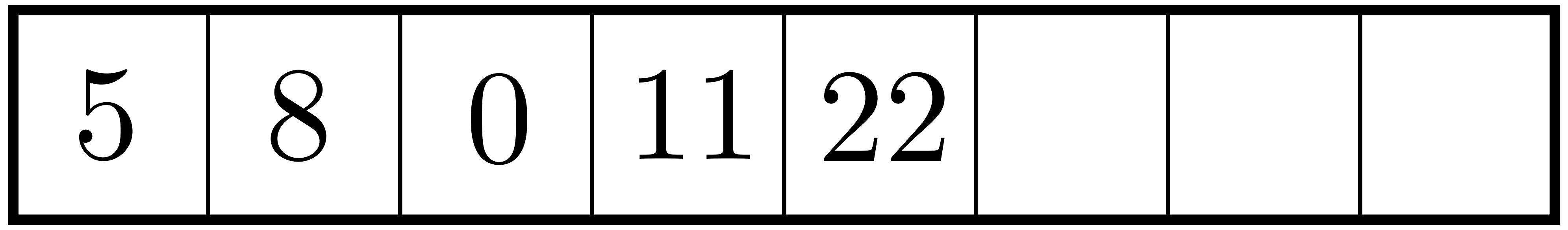
excess capacity, no copies.

push back 22:

4 copies



no capacity, allocate array of size 8.



copy 5, 8, 0, 11 and insert 22.

General Pattern

We have no excess capacity when inserting element $2^i + 1$, for $i = 0, 1, 2, 3$.

For this insertion, we have to do 2^i copy operations.

All other insertions do not require any copies and work in constant time.

Thus to insert $2^k + 1$ elements the total number of copy operations is

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

General Pattern

Thus to push back $2^k + 1$ elements the total number of copy operations is

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

This shows that the time to push back n elements is at most a constant times n.

In the "array doubling" solution, push back takes amortized constant time.

Amortized means averaged over a sequence of operations.

std::vector

std::vector ADT from en.cppreference.com

Capacity

empty	checks whether the container is empty (public member function)
size	returns the number of elements (public member function)
max_size	returns the maximum possible number of elements (public member function)
reserve	reserves storage (public member function)
capacity	returns the number of elements that can be held in currently allocated storage (public member function)
shrink_to_fit (C++11)	reduces memory usage by freeing unused memory (public member function)

Modifiers

clear	clears the contents (public member function)
insert	inserts elements (public member function)
emplace (C++11)	constructs element in-place (public member function)
erase	erases elements (public member function)
push_back	adds an element to the end (public member function)
emplace_back (C++11)	constructs an element in-place at the end (public member function)
pop_back	removes the last element (public member function)

description of push back from en.cppreference.com

std::vector<T,Allocator>::push_back

`void push_back( const T& value );`	(1)	(until C++20)
`constexpr void push_back( const T& value );`		(since C++20)
`void push_back( T&& value );`	(2)	(since C++11) (until C++20)
`constexpr void push_back( T&& value );`		(since C++20)

Appends the given element `value` to the end of the container.

1) The new element is initialized as a copy of `value`.

2) `value` is moved into the new element.

If the new `size()` is greater than `capacity()` then all iterators and references (including the past-the-end iterator) are invalidated. Otherwise only the past-the-end iterator is invalidated.

Parameters

`value` - the value of the element to append

Type requirements

- T must meet the requirements of *CopyInsertable* in order to use overload (1).
- T must meet the requirements of *MoveInsertable* in order to use overload (2).

Return value

(none)

Complexity

Amortized constant.

Notes

It is possible to implement a dynamic array with every insertion taking worst-case constant time.

See the paper "Resizable Arrays in Optimal Time and Space" by Brodnik, Carlsson, Demaine, Munro, and Sedgewick.

https://cs.uwaterloo.ca/~imunro/cs840/ResizableArrays.pdf

Linked List

Linked List

A linked list is another sequence container.

It is fast to insert/remove elements from both the front and the back of the list.

What we give up in a linked list is fast access to the i^{th} element. To get to the i^{th} element we have to iterate from the beginning (or end).

Linked List

A linked list is another sequence container.

It is fast to insert/remove elements from both the front and the back of the list.

What we give up in a linked list is fast access to the i^{th} element. To get to the i^{th} element we have to iterate from the beginning (or end).

Subtle one: It is fast to insert/remove elements to the middle of the linked list...

Linked List

A linked list is another sequence container.

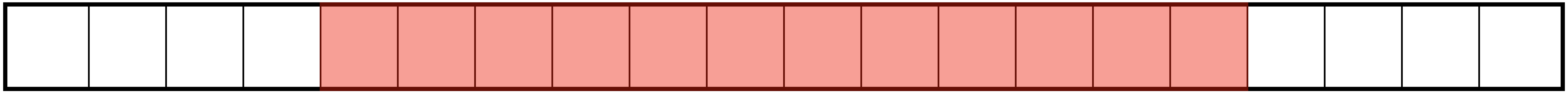
It is fast to insert/remove elements from both the front and the back of the list.

What we give up in a linked list is fast access to the i^{th} element. To get to the i^{th} element we have to iterate from the beginning (or end).

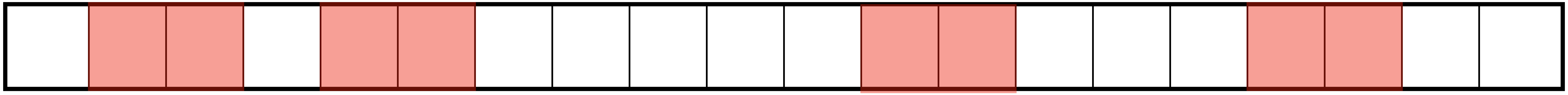
Subtle one: It is fast to insert/remove elements to the middle of the linked list... given some extra information.

Comparison with Array

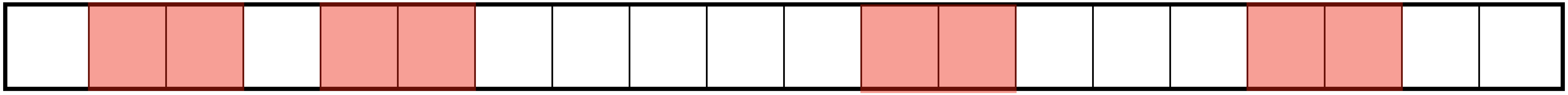
In the implementation of a resizable array, values were stored contiguously in memory.



In a linked list, values can be spread out in memory.



Node



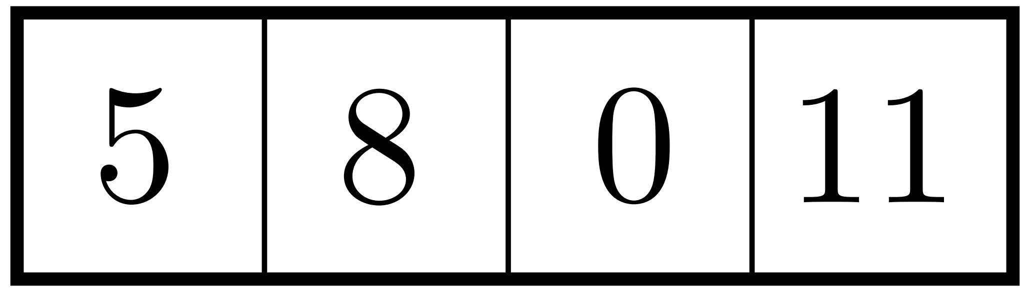
A linked list is comprised of nodes.

Each data value is stored in a separate node.

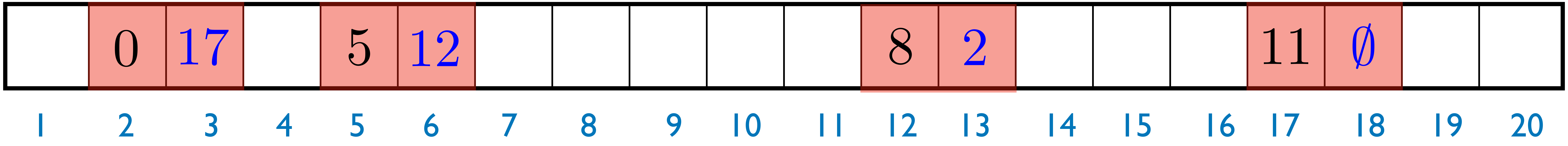
A node also stores the location in memory of the next node.

Thus a node has at least two fields:

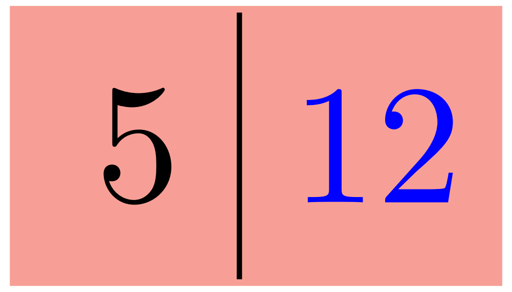
Memory Picture



array version

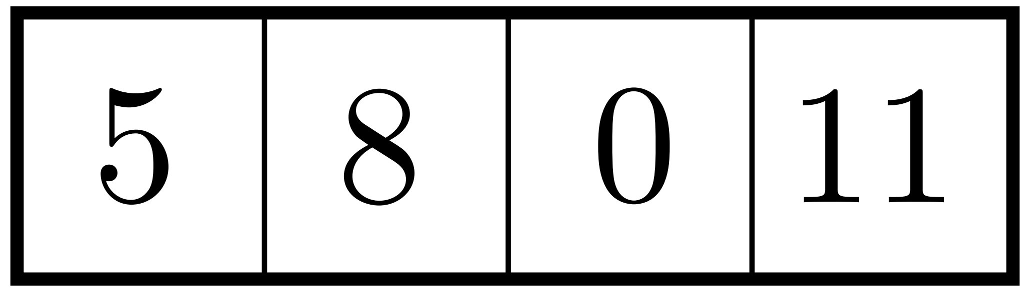


Example node:

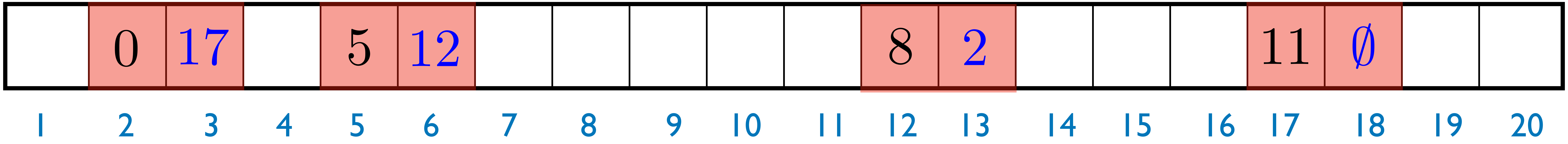


The first field holds the value, the second field holds the address of the next node (holding value 8).

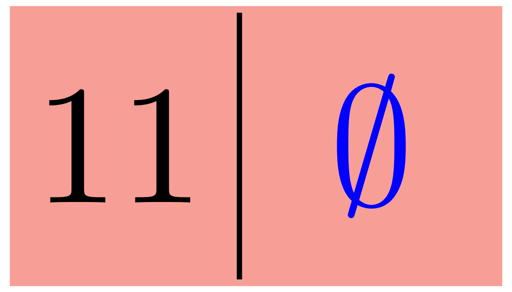
End of the List



array version

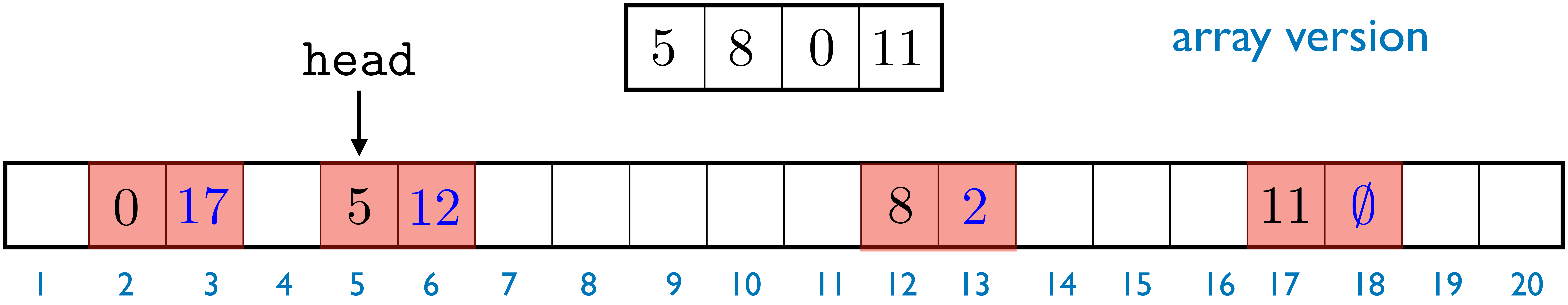


Last node:



The second word of the last node indicates that it is the last node by holding an invalid address (denoted $\emptyset$), a null pointer.

Head of the list

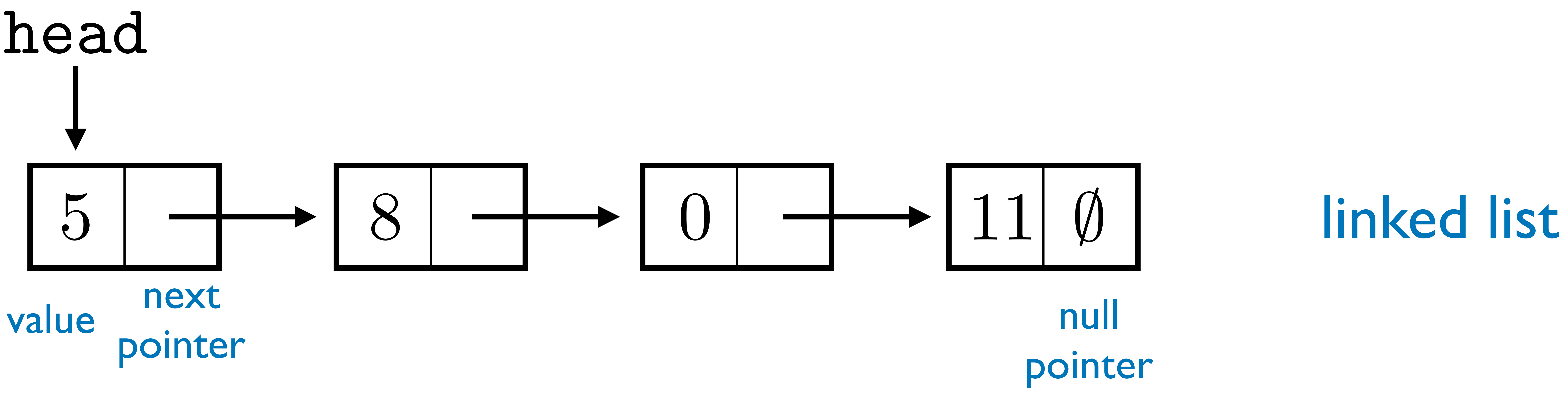


We also need to know where the list starts, the address of the first node.

We have a variable `head` with the address of the first node.

$$\texttt{head} = 5$$

Alternative picture



Usually we draw a linked list like this, drawing an arrow to the next node rather than writing the address.

This makes the sequence of values clear.

Implementation

Let's be more concrete about the implementation of a linked list in C++.

```
struct Node {
    int val = 0;
    Node* next = nullptr;
};
```

https://godbolt.org/z/Pbb5hYnc3

The godbolt link has all the code for a singly linked list we discuss.

Create some nodes

```
Node first {1};
Node second {2};
Node third {3};
Node fourth {4};
```

```
struct Node {
    int val = 0;
    Node* next = nullptr;
};
```

We are using aggregate initialization here, see https://www.learncpp.com/cpp-tutorial/struct-aggregate-initialization/

Aggregate Initialization

```
Node first {1};
Node second {2};
Node third {3};
Node fourth {4};
```

```
struct Node {
    int val = 0;
    Node* next = nullptr;
};
```

In aggregate initialization, variables are initialized according to the order in which they are declared in the struct.

If we were to declare a new variable in Node before val, this would mess up our code!

Aggregate Initialization

```
Node first {1};
Node second {2};
Node third {3};
Node fourth {4};
```

```
struct Node {
    Node* prev = nullptr;
    int val = 0;
    Node* next = nullptr;
};
```

If later I changed my code to have a pointer to a Node declared first, it would no longer compile.

```
error: invalid conversion from 'int' to 'Node*'
```

Now the compiler is trying to convert 1 into a Node* and can't do it.

C++ 20

```
Node first {.val {1}};
Node second {.val {2}};
Node third {.val {3}};
Node fourth {.val {4}};
```

```
struct Node {
    int val = 0;
    Node* next = nullptr;
};
```

https://godbolt.org/z/Pf3sYan5a

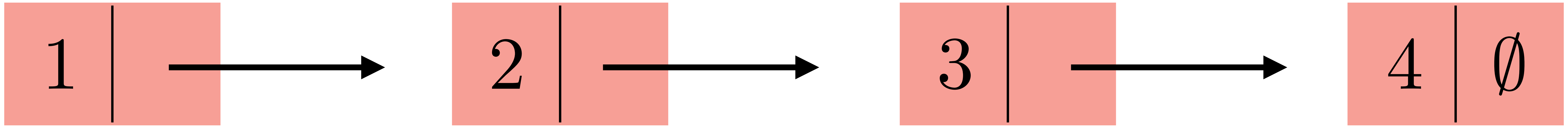
C++20 added a nice feature to avoid this problem. We can specify which variable the value in the initialization list applies to.

We still have to respect the declaration order, but now variables in between can be dropped.

Linking Nodes

```
first.next = &second;
second.next = &third;
third.next = &fourth;
```

```
struct Node {
    int val = 0;
    Node* next = nullptr;
};
```



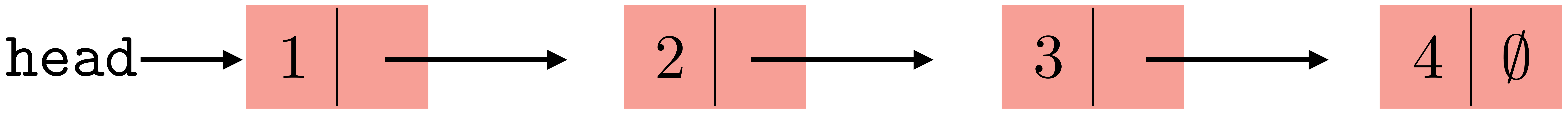
We set `first.next` to be equal to the address of the second node, and similarly for the second and third nodes.

We don't need to update `fourth.next` as it is already `nullptr`.

Iterating through a list

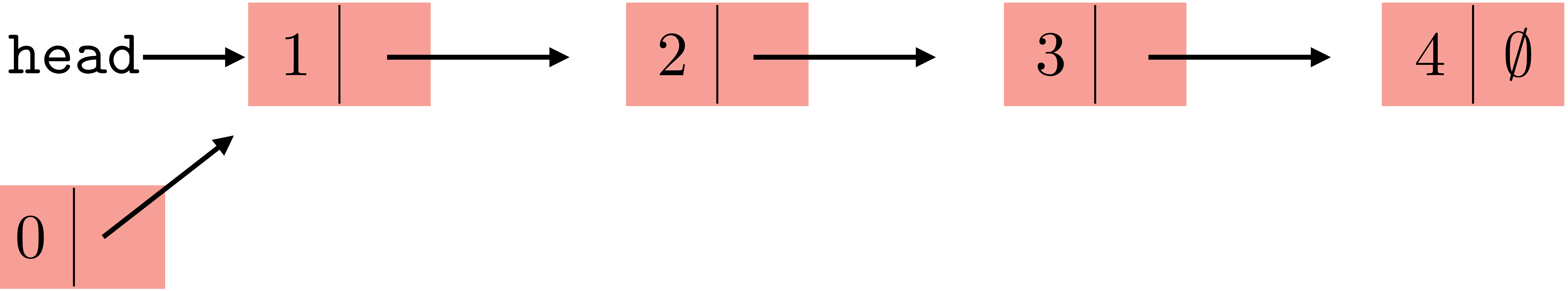
There is a standard idiom for iterating through a list:

```
for(Node* current = head; current != nullptr; current = current->next) {
    std::cout << current->val << ' ';
```



$\texttt{current} \to \texttt{next}$ is syntactic sugar for $(*\texttt{current}).\texttt{next}$.

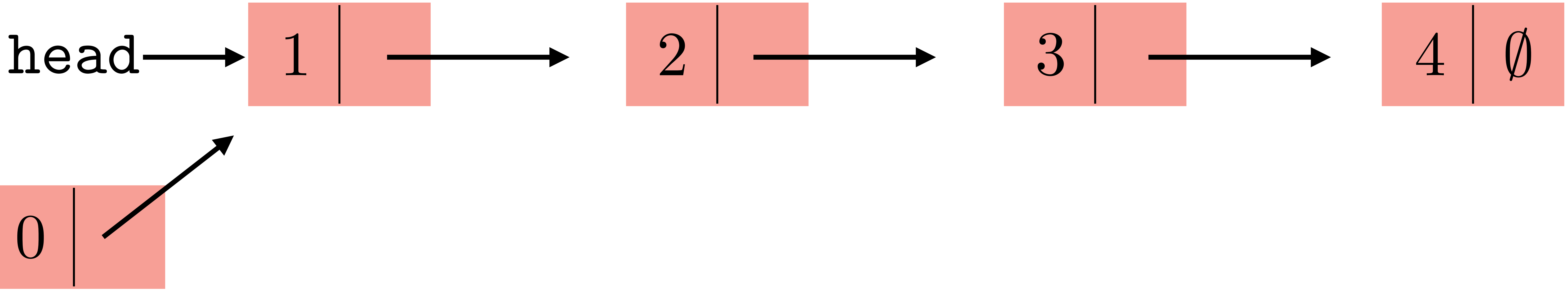
push_front



Create a new node whose `next` pointer points to the first node.

```
Node zero {0, head};
```

push_front

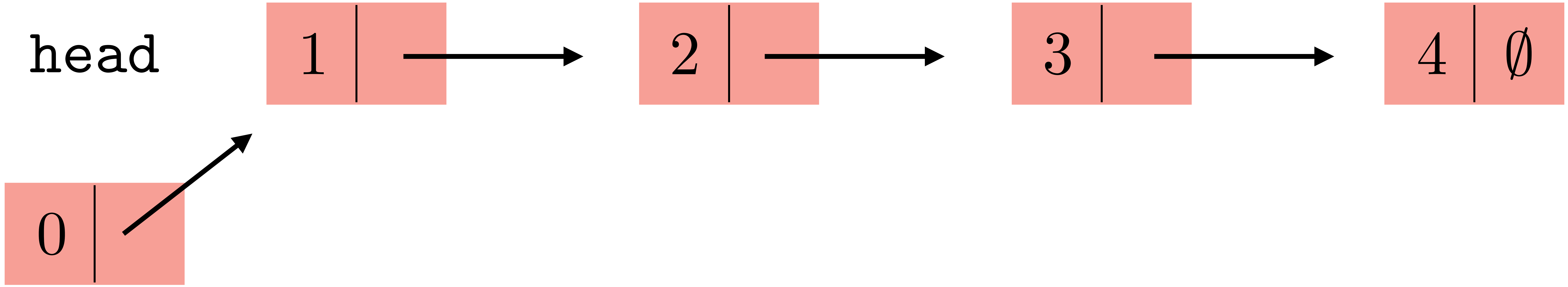


Now update the `head` pointer to point to the zero node.

```
head = &zero;
```

This whole process takes constant time.

push_front

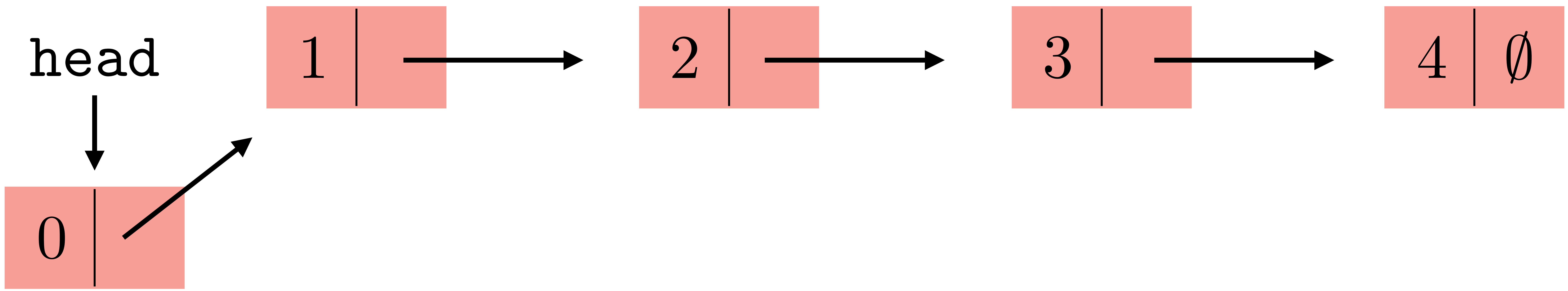


Now update the `head` pointer to point to the zero node.

```
head = &zero;
```

This whole process takes constant time.

push_front

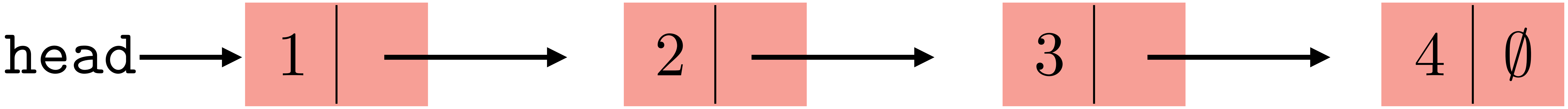


Now update the `head` pointer to point to the zero node.

```
head = &zero;
```

This whole process takes constant time.

push_back



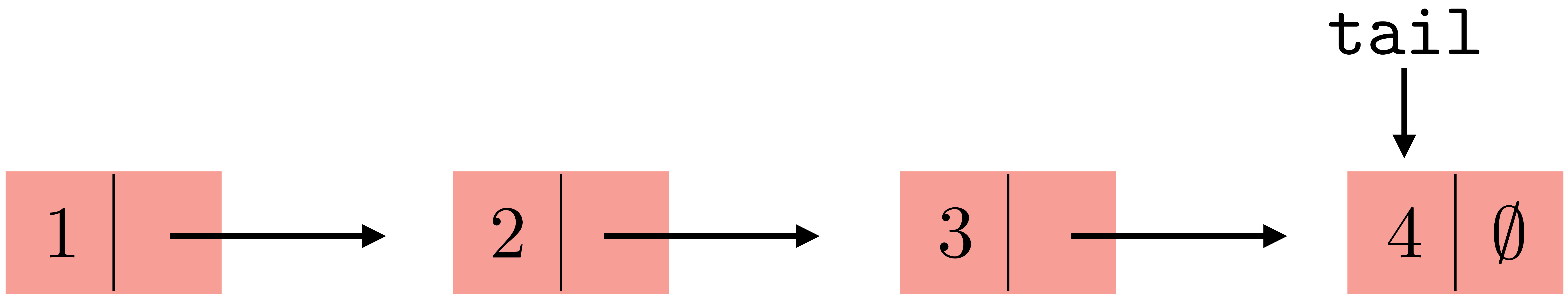
Say we want to add a node with 5 to the end of this list.

How much time would this take?

The main problem is that we do not know the address of the last node.

In order to find this address we have to walk through the whole list, making this a slow operation.

push_back

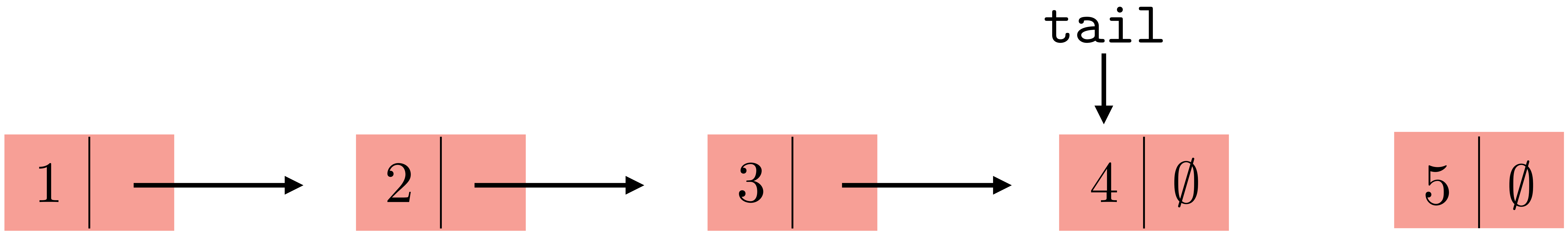


We can easily fix this by adding a pointer `tail` to the last node of the list.

Then push_back can be implemented in constant time.

```
Node fifth {5};
tail->next = &fifth;
tail = &fifth;
```

push_back

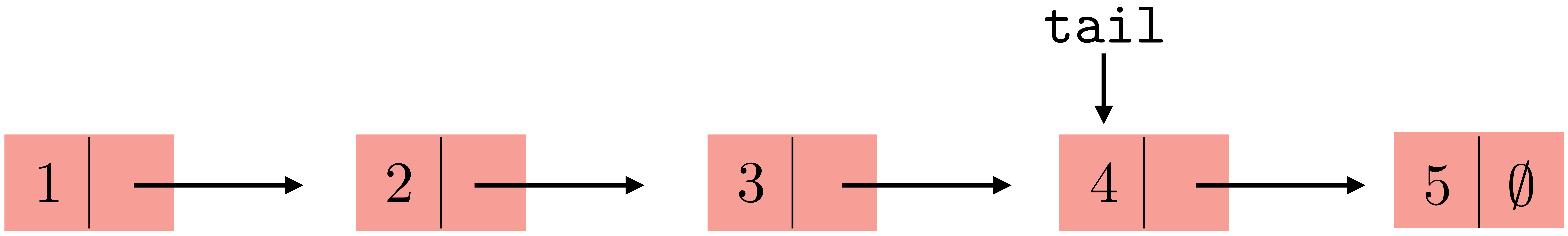


We can easily fix this by adding a pointer `tail` to the last node of the list.

Then push_back can be implemented in constant time.

```
Node fifth {5};
tail->next = &fifth;
tail = &fifth;
```

push_back

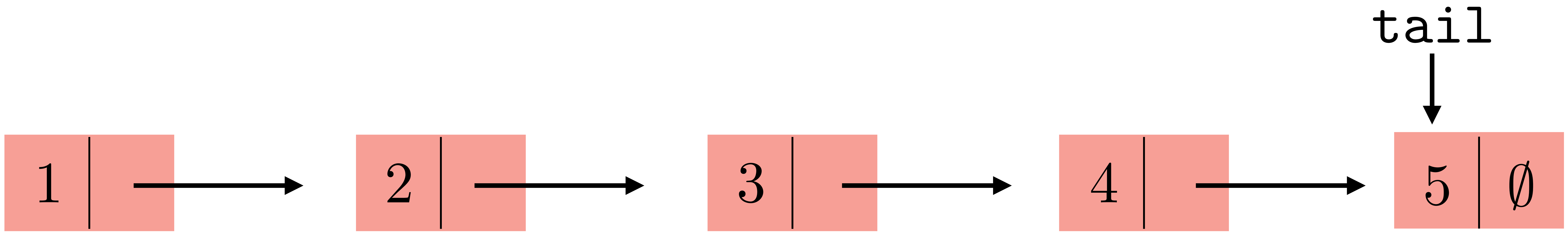


We can easily fix this by adding a pointer `tail` to the last node of the list.

Then push_back can be implemented in constant time.

```
Node fifth {5};
tail->next = &fifth;
tail = &fifth;
```

push_back

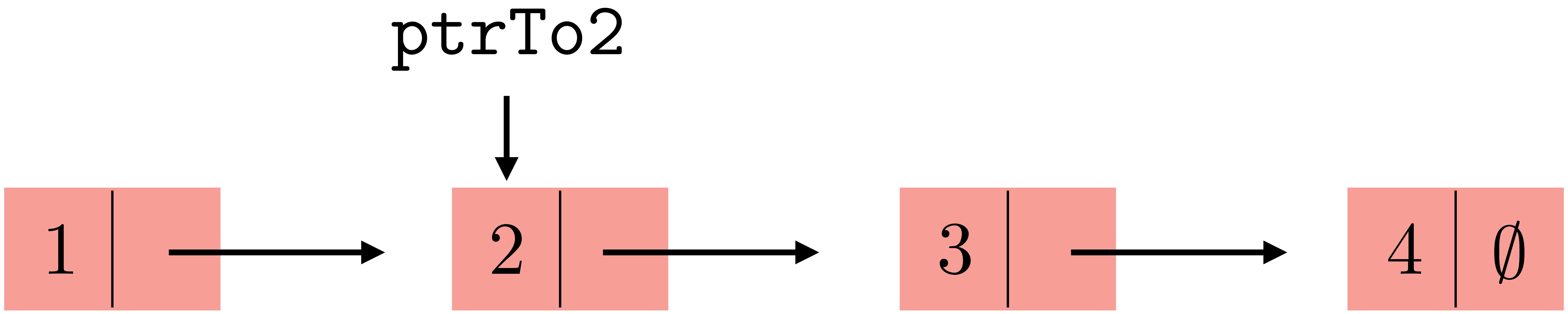


We can easily fix this by adding a pointer `tail` to the last node of the list.

Then push_back can be implemented in constant time.

```
Node fifth {5};
tail->next = &fifth;
tail = &fifth;
```

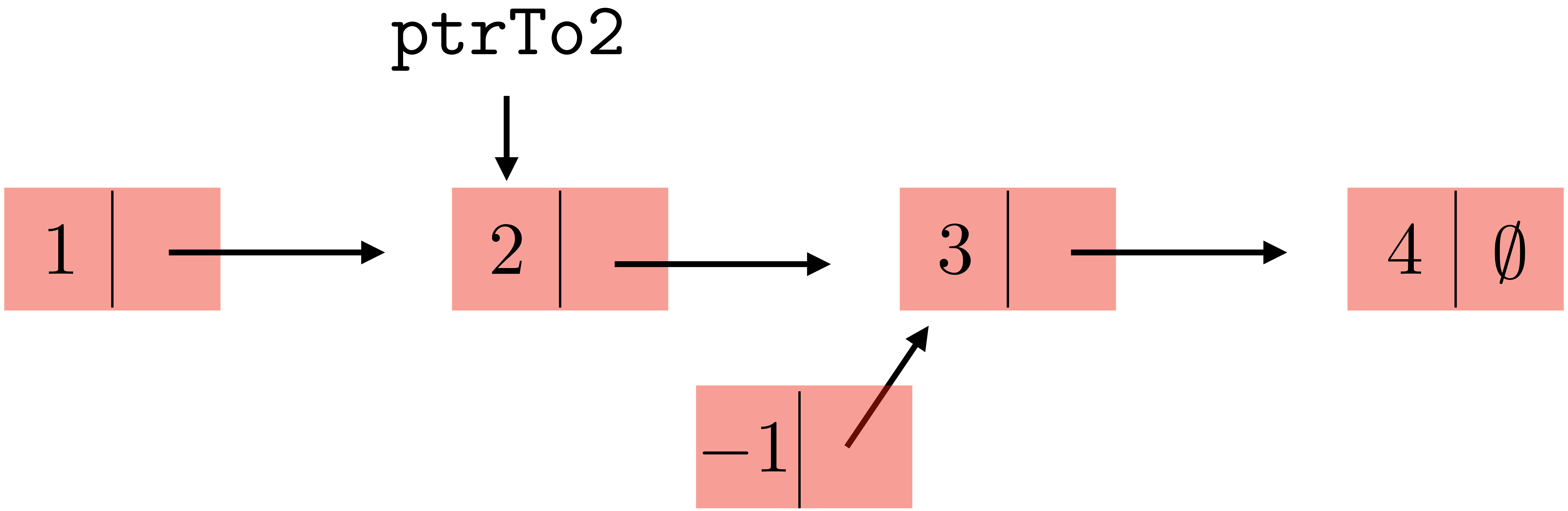
insert in the middle



Say we want to insert a node with value -1 in between the second and third nodes.

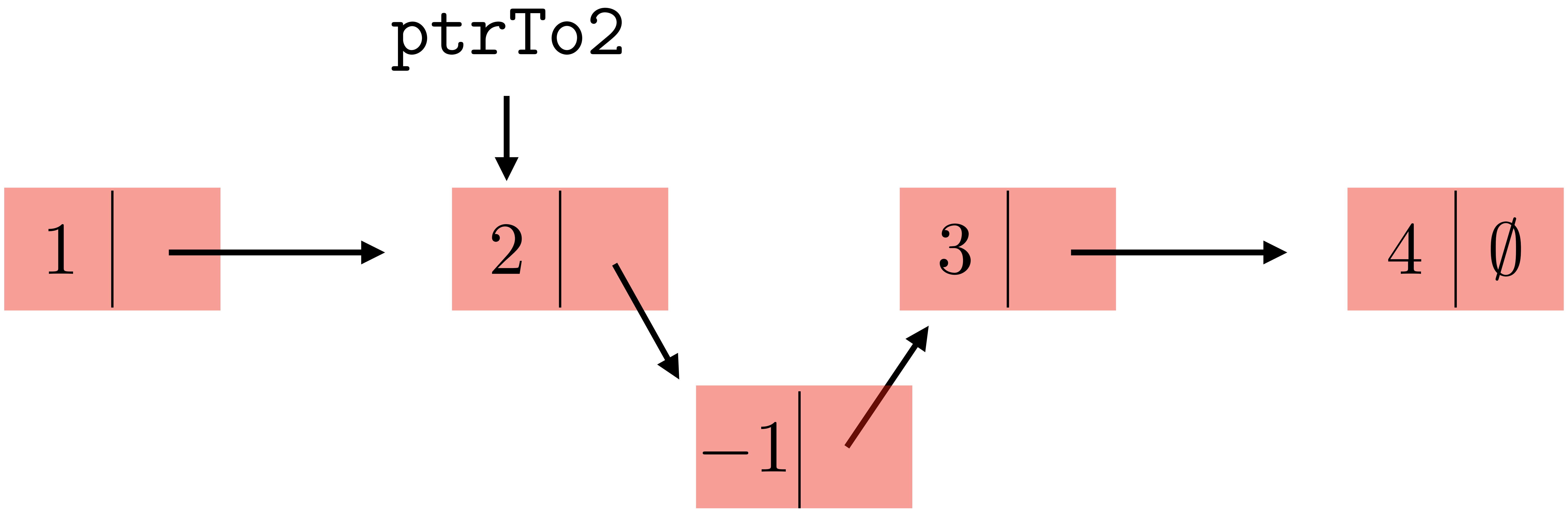
This is easy to do given the address of the second node.

insert in the middle




```
Node* ptrTo2 = &second;
Node twoAndAHalf {-1, ptrTo2->next};
ptrTo2->next = &twoAndAHalf;
```

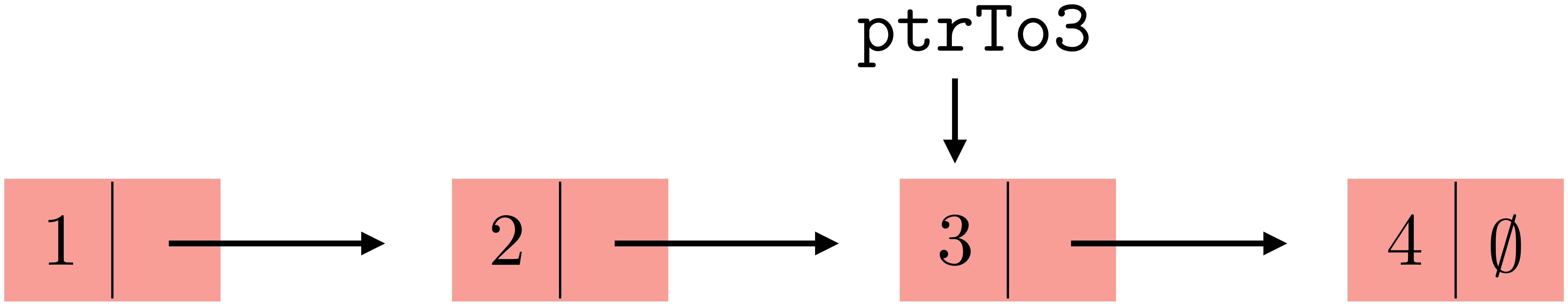
insert in the middle




```
Node* ptrTo2 = &second;
Node twoAndAHalf {-1, ptrTo2->next};
ptrTo2->next = &twoAndAHalf;
```

Given `ptrTo2`, this is constant time.

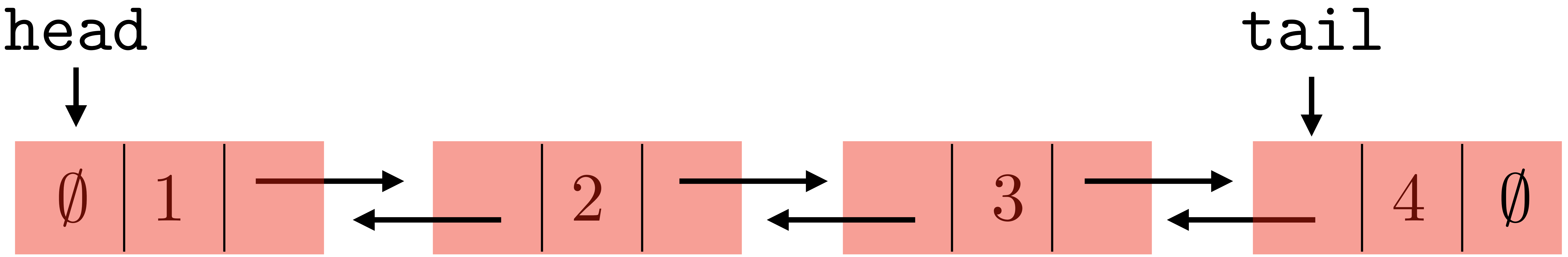
insert before



What if we want to insert a node between 2 and 3 and we are given a pointer to the third node?

Now we are stuck! We have no way to find the address of the node before 3, other than iterating through the list.

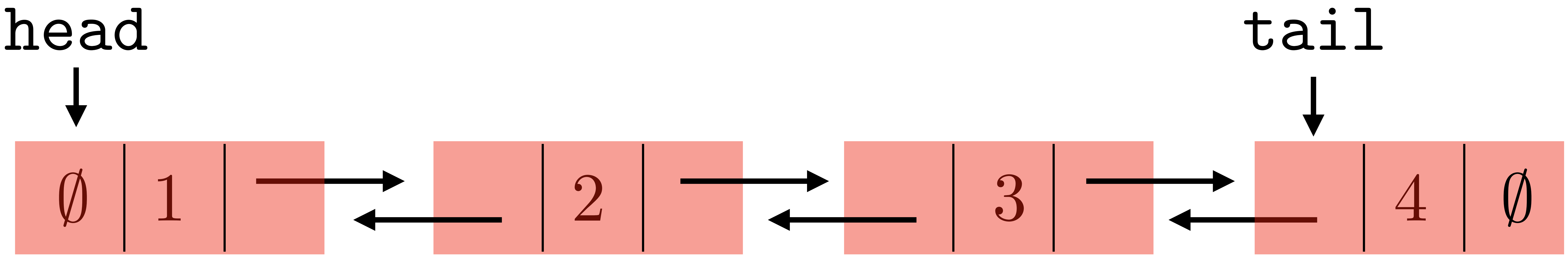
Doubly Linked List



We can solve this by using a doubly linked list.

Each node has a pointer to the next node and the previous node.

Doubly Linked List

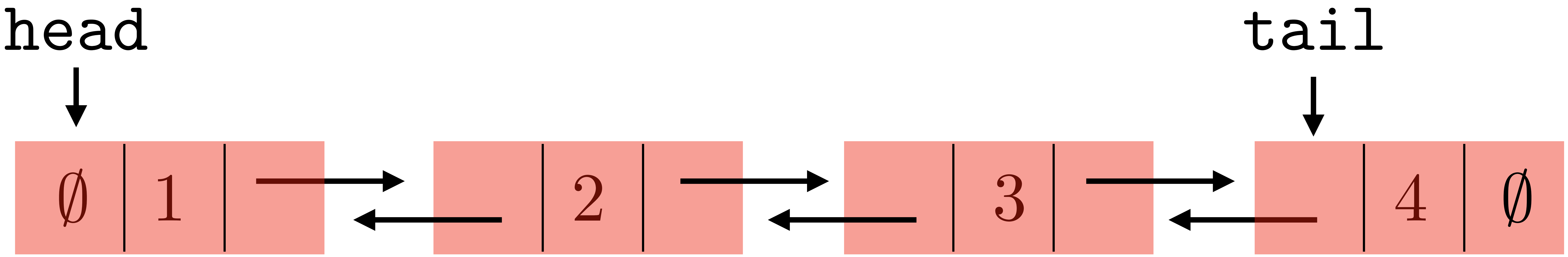


The struct for a node now looks like this:

```
struct Node {
    int val = 0;
    Node* next = nullptr;
    Node* prev = nullptr;
};
```

https://godbolt.org/z/h5T353cxs

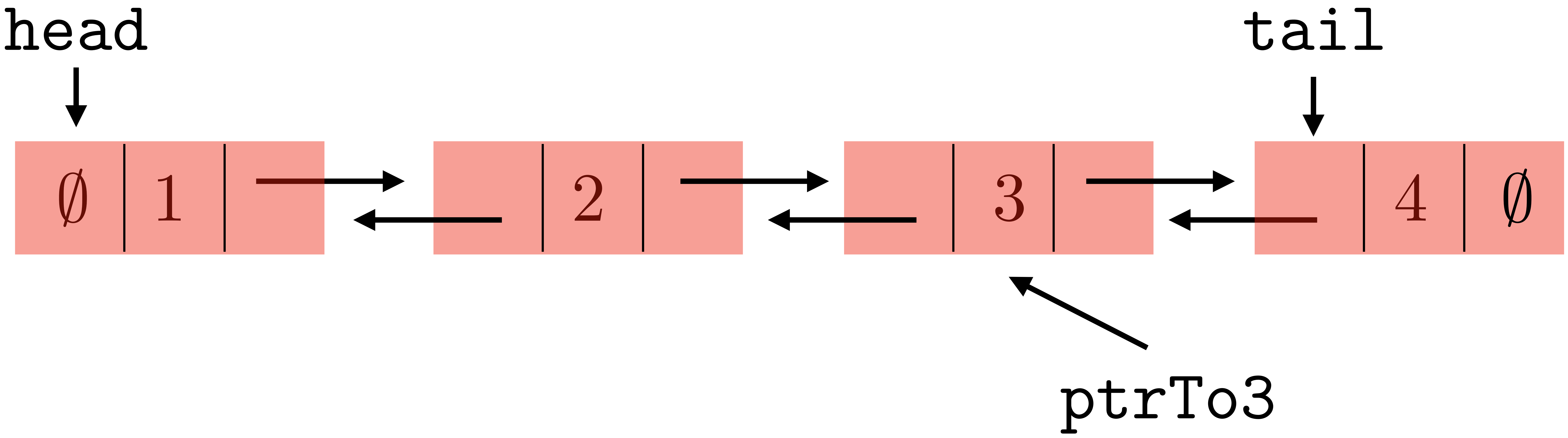
Doubly Linked List



Now we can iterate backwards through the list:

```
for(Node* current = tail; current != nullptr; current = current->prev) {
    std::cout << current->val << ' ';
```

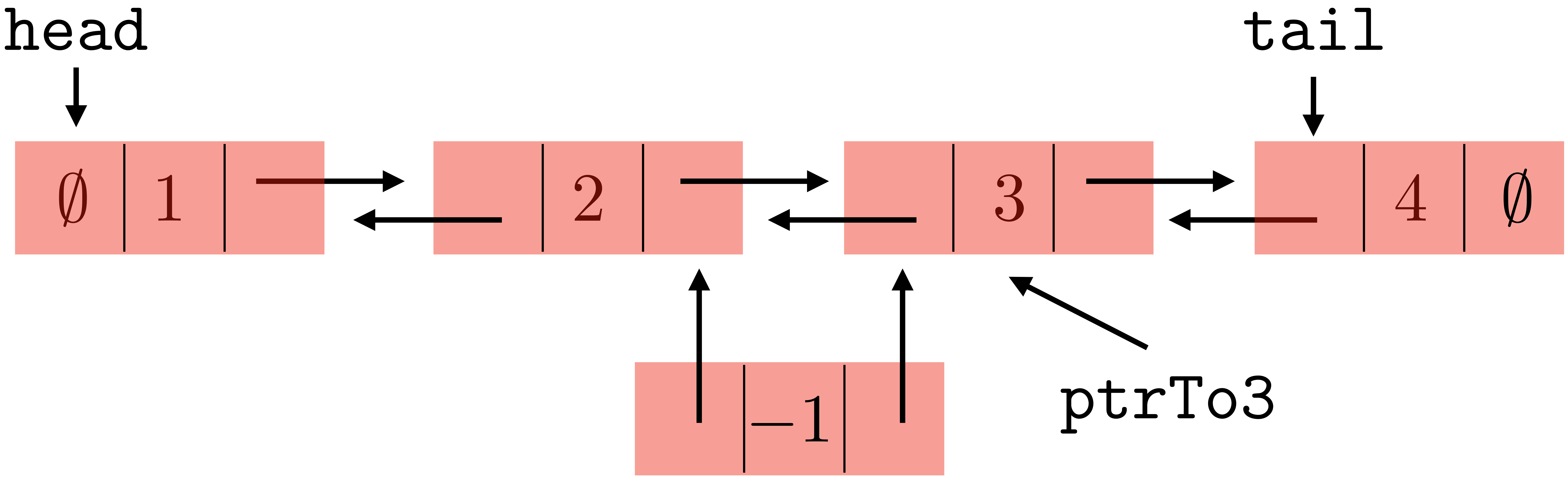
insert before a node



Given a pointer to the node with value 3, we can insert a node with value -1 before it.

```
Node twoAndAHalf {-1, ptrTo3, ptrTo3->prev};
```

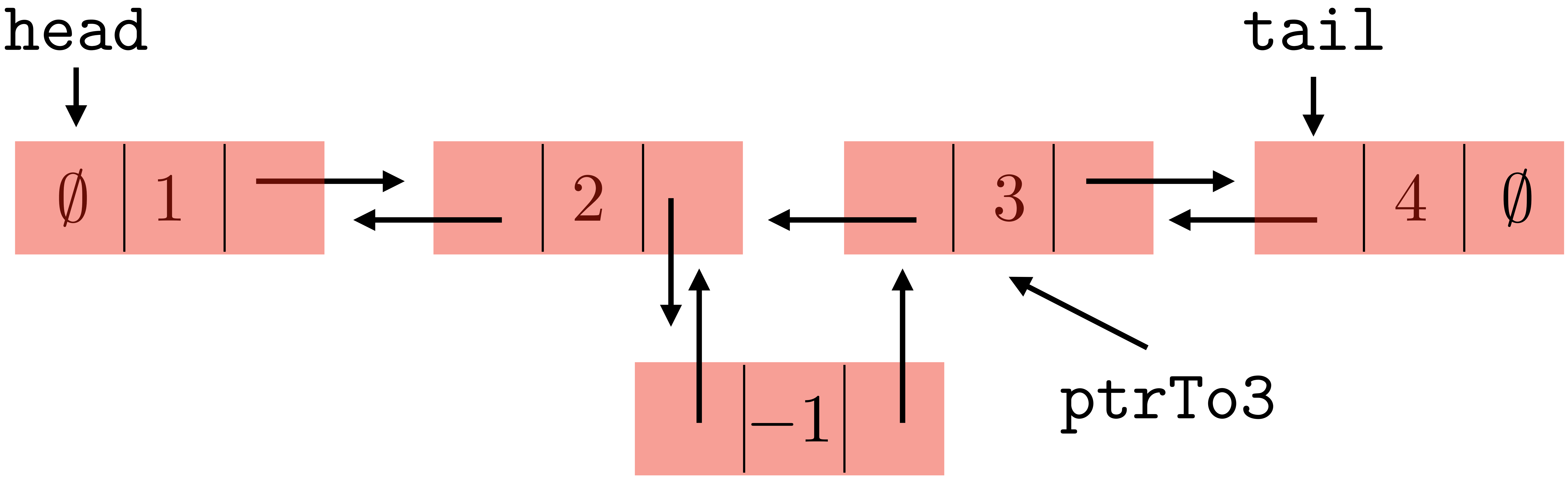
insert before a node



Given a pointer to the node with value 3, we can insert a node with value -1 before it.

```
Node twoAndAHalf {-1, ptrTo3, ptrTo3->prev};
```

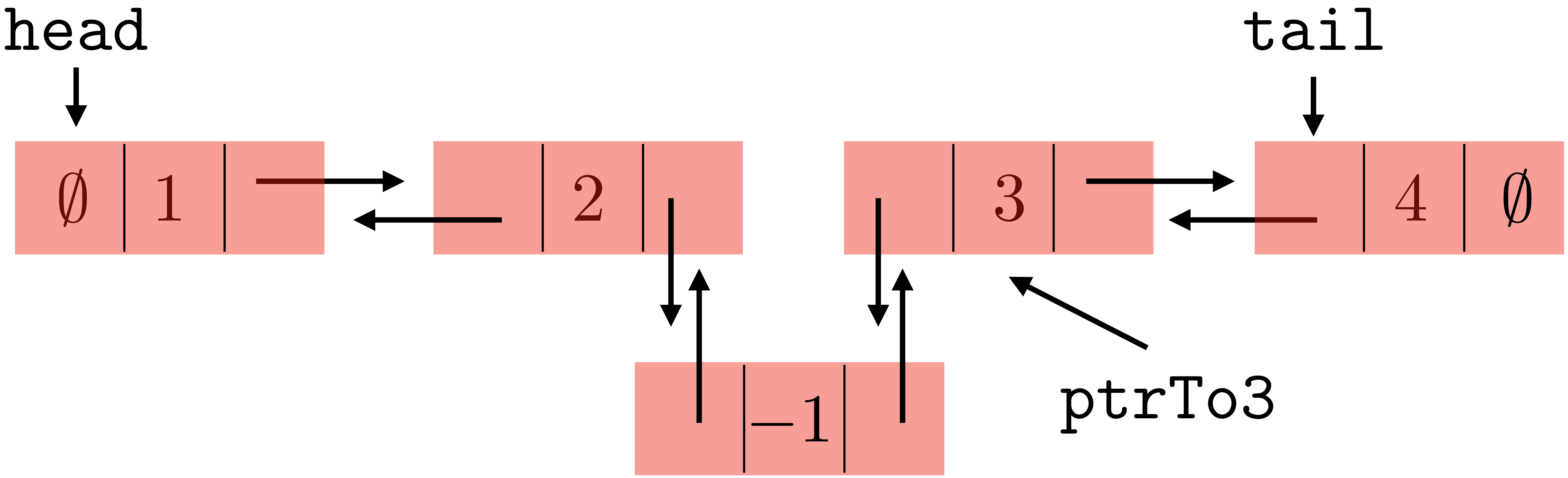
insert before a node



The `next` variable of the node before 3 should now point to `twoAndAHalf`.

```
(ptrTo3->prev)->next = &twoAndAHalf;
ptrTo3->prev = &twoAndAHalf;
```

insert before a node



The `prev` variable of node 3 should point to `twoAndAHalf`.

```
(ptrTo3->prev)->next = &twoAndAHalf;
ptrTo3->prev = &twoAndAHalf;
```

Singly Linked List ADT

$A \leftarrow \texttt{SinglyLinkedList}()$	Creates an empty list.
$A.\texttt{push\_front}(\texttt{x})$	Add x to front of the list.
$A.\texttt{pop\_front}()$	Remove the first element.
$A.\texttt{insert\_after}(\texttt{loc}, \texttt{x})$	Insert x into the list after the node with address `loc`.
$A.\texttt{erase\_after}(\texttt{loc})$	Remove the element stored in the node after the node with address `loc`.

Singly Linked List ADT

$A \leftarrow \texttt{SinglyLinkedList}()$

$A.\texttt{push\_front}(\texttt{x})$

$A.\texttt{pop\_front}(\texttt{x})$

$A.\texttt{insert\_after}(\texttt{loc}, \texttt{x})$

$A.\texttt{erase\_after}(\texttt{loc})$

All of these operations can be implemented in constant time.

std::forward_list

There is an implementation of a singly linked list in the standard library.

We have modelled our Singly Linked ADT after this implementation.

This is a bare bones singly linked list:

- There is no $A.\texttt{push\_back}(\texttt{x})$ function.

  This saves having a `tail` variable.

- There is no $A.\texttt{size}()$ function.

Doubly Linked List ADT

$A \leftarrow \texttt{DoublyLinkedList}()$ Creates an empty list.

$A.\texttt{front}(), \texttt{A.back}()$ Return the first/last element.

$A.\texttt{push\_front}(\texttt{x}), \texttt{A.push\_back}(\texttt{x})$ Add x to the front/back of the list.

$A.\texttt{pop\_front}(), \texttt{A.pop\_back}()$ Remove the first/last item of the list.

$A.\texttt{insert}(\texttt{loc}, \texttt{x})$ Insert x into the list before the node with address `loc`.

Deque

Deque

In a resizable array, we can only add and remove items at the end of the array.

Sometimes we might want to also add and remove items at the beginning of the array.

When we always add items to the end, the oldest item in the array is at the front.

In some applications, we want to access and remove the front item.

A deque allows us to add/remove items at both the front and back.

Deque ADT

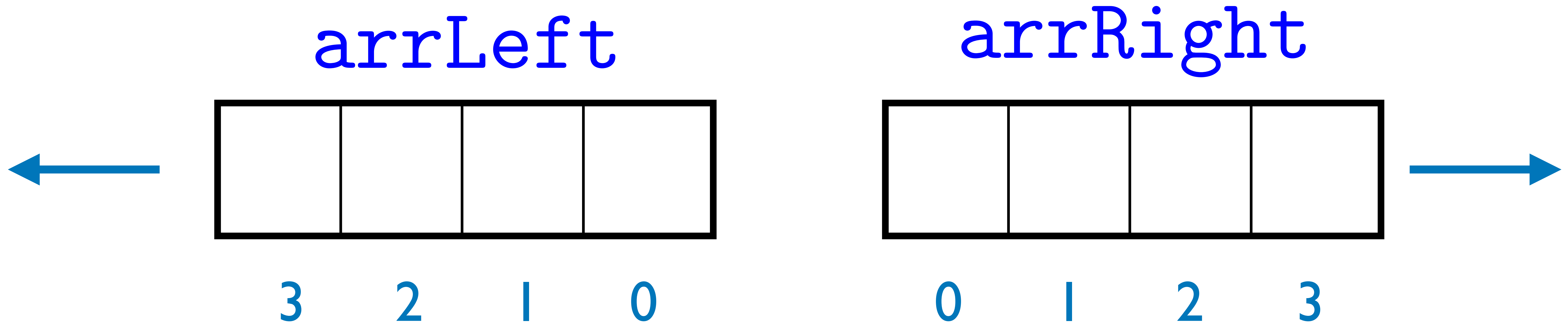
A deque is an extension of a resizable array.

It has all the operations of a resizable array plus:

$A.\texttt{push\_front}(\texttt{x})$	Add x to the front of A .
$A.\texttt{pop\_front}()$	Remove the first element of A.

Implementation of a Deque

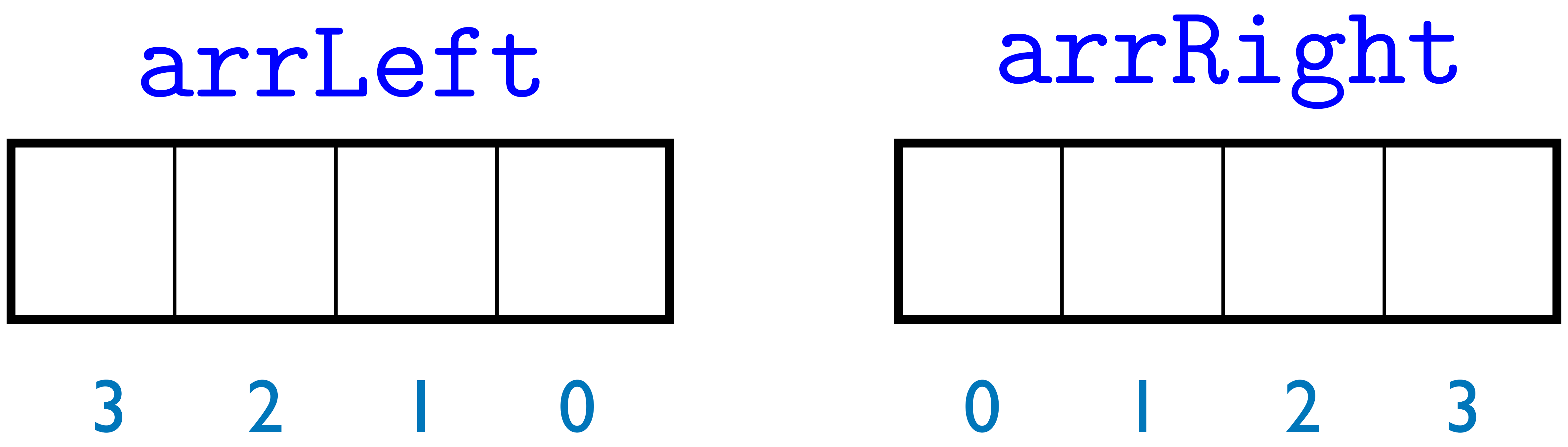
We can implement a deque with two resizable arrays put "front to front".



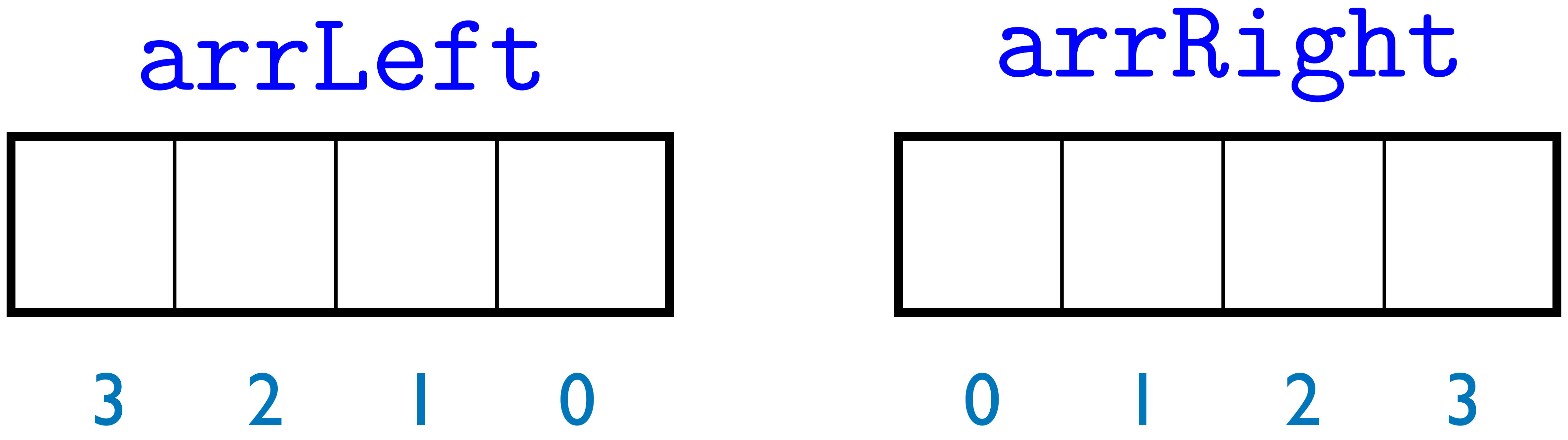
`arrLeft` is drawn backwards---it grows to the left.

In normal operation, push_back/pop_back are done on `arrRight`, and push_front/pop_front are done on `arrLeft`.

Example

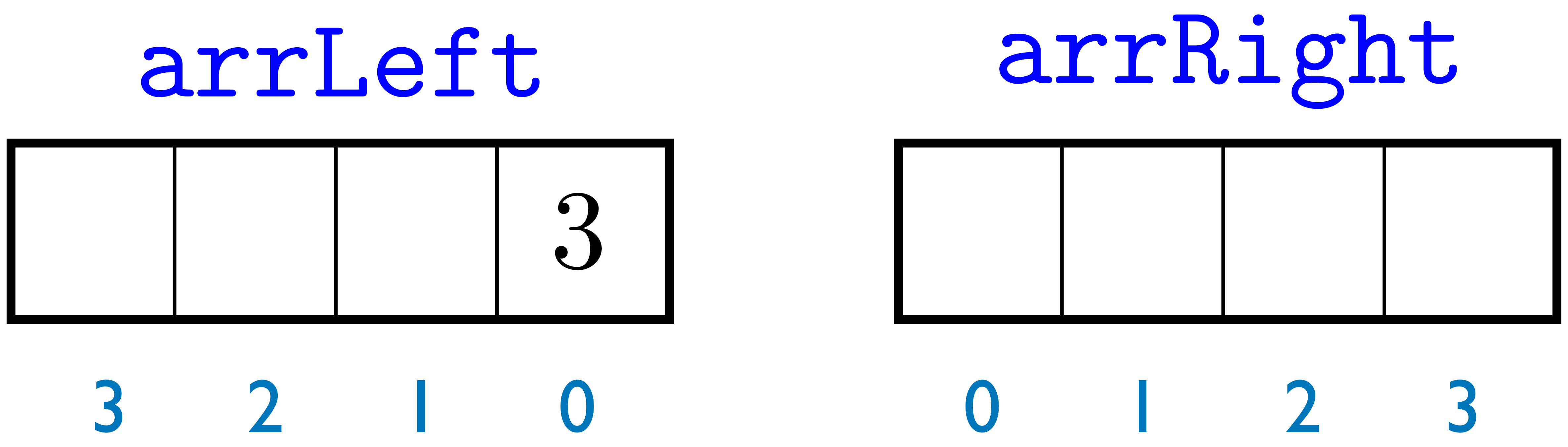


Example



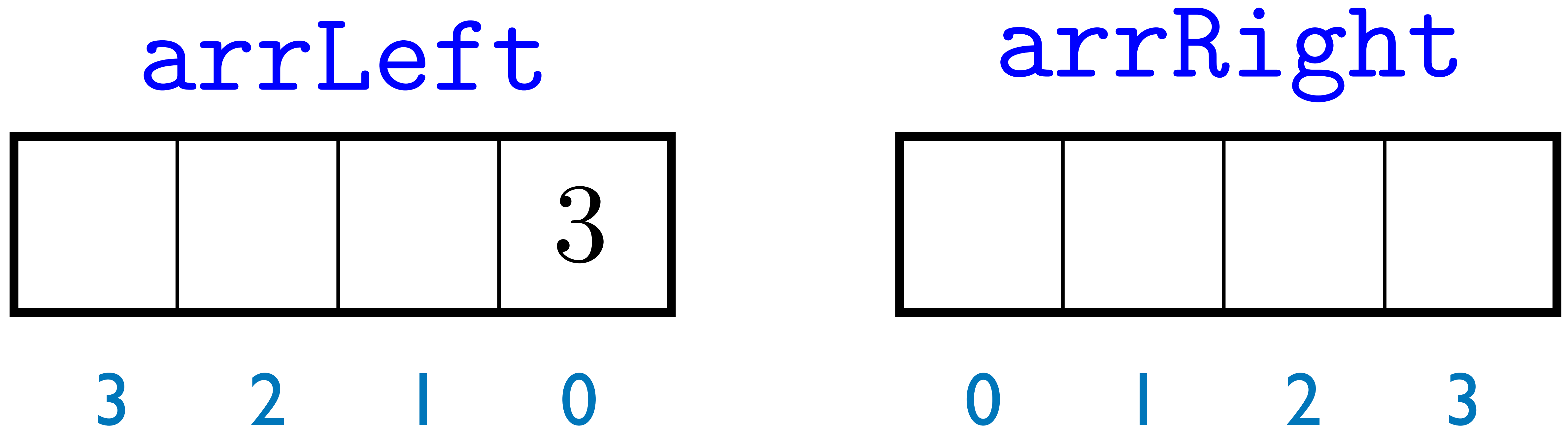
A.push_front(3)

Example



$A.\texttt{push\_front}(3)$

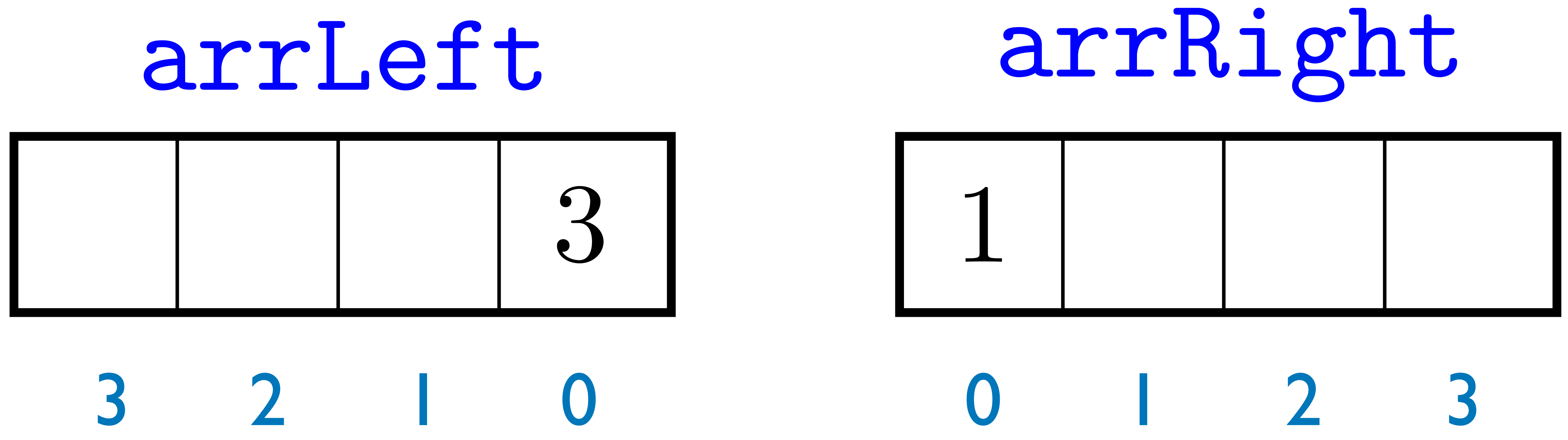
Example



A.push_front(3)

A.push_back(1)

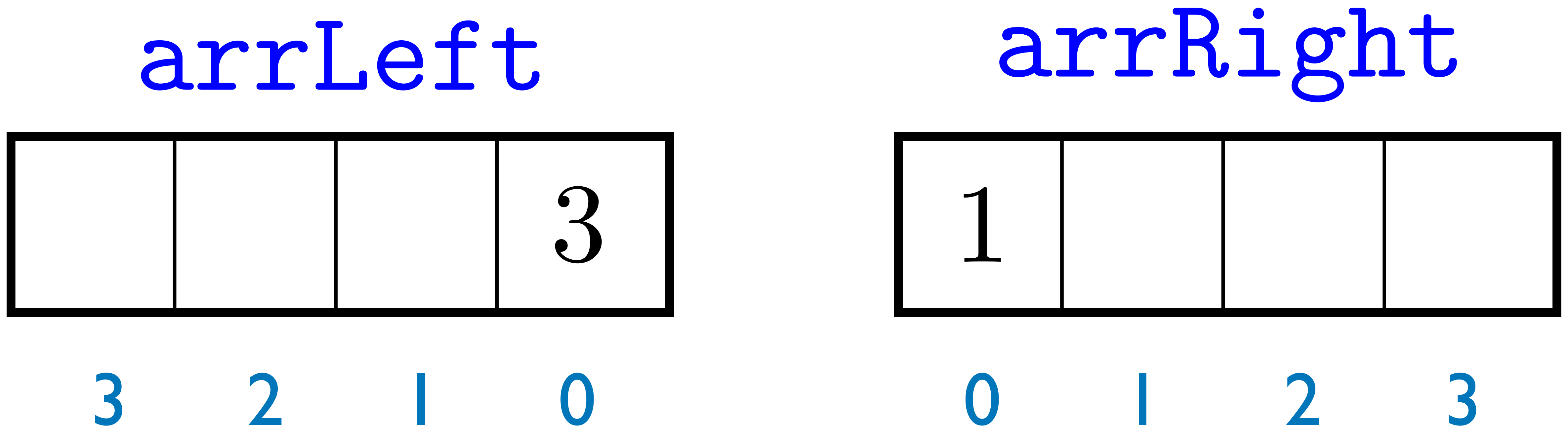
Example



A.push_front(3)

A.push_back(1)

Example

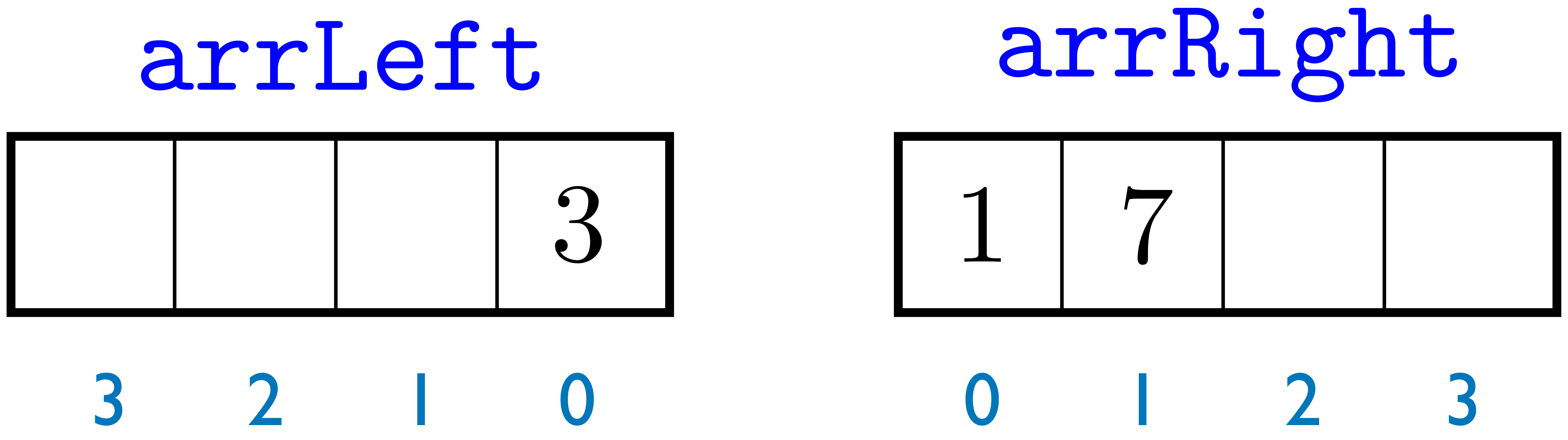


A.push_front(3)

A.push_back(1)

A.push_back(7)

Example

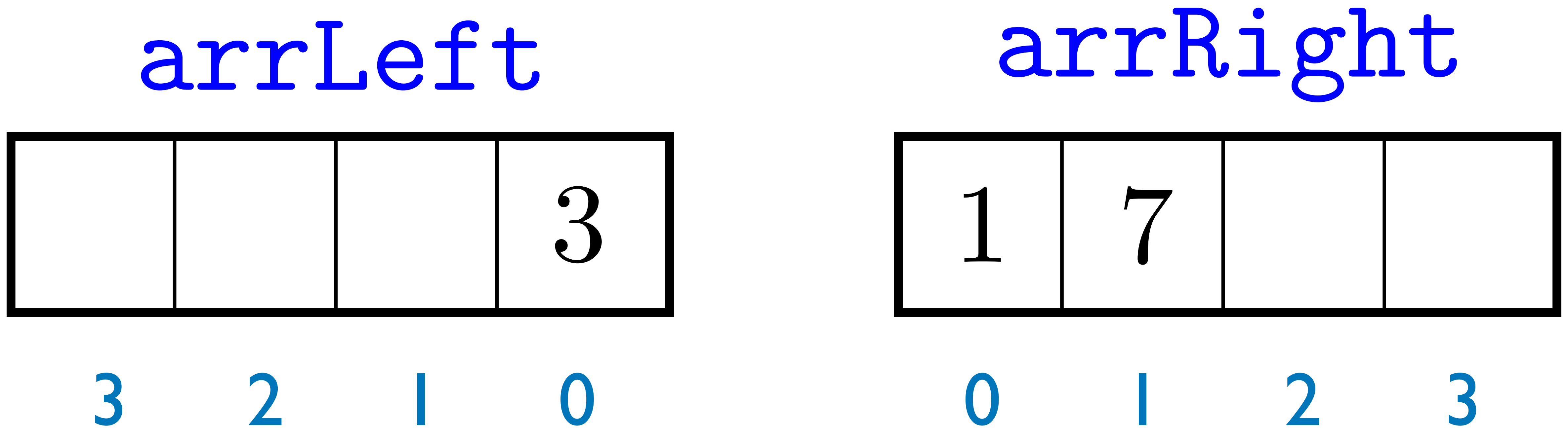


A.push_front(3)

A.push_back(1)

A.push_back(7)

Example



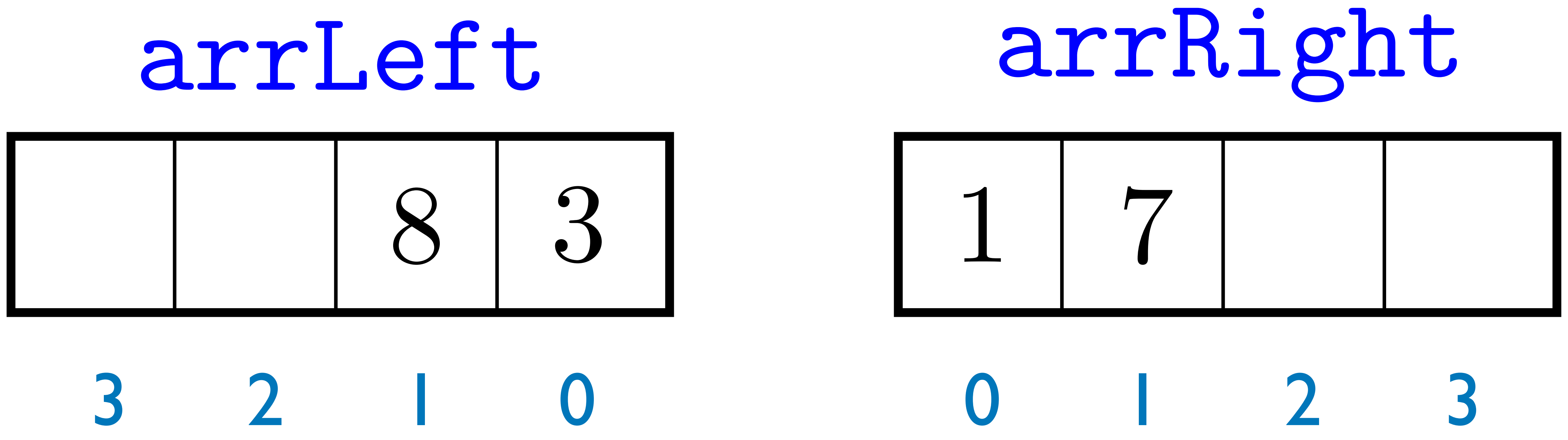
A.push_front(3)

A.push_back(1)

A.push_back(7)

A.push_front(8)

Example



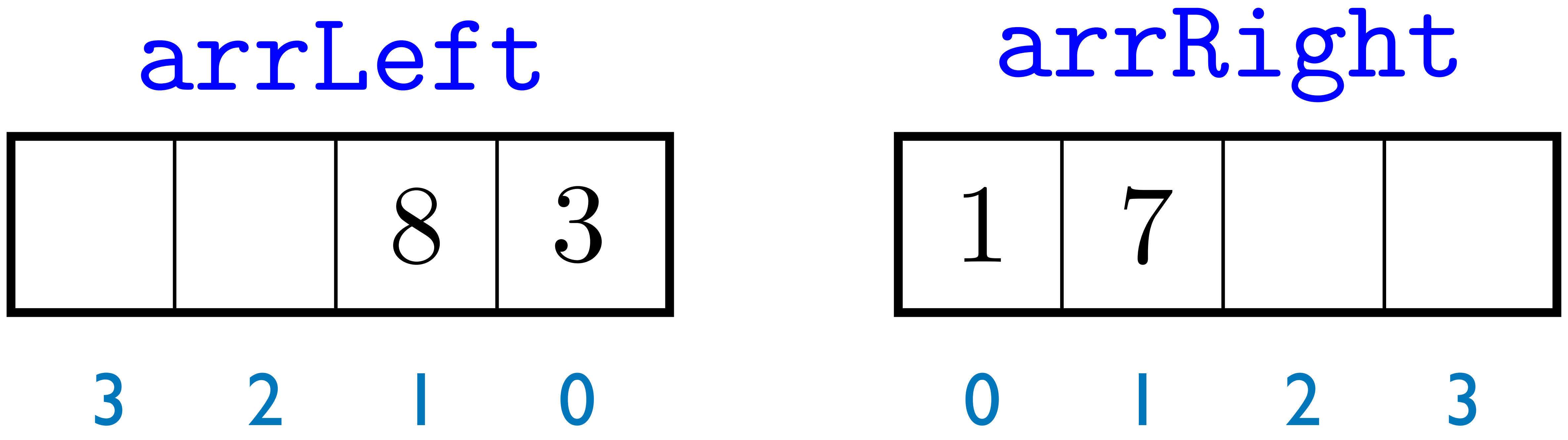
A.push_front(3)

A.push_back(1)

A.push_back(7)

A.push_front(8)

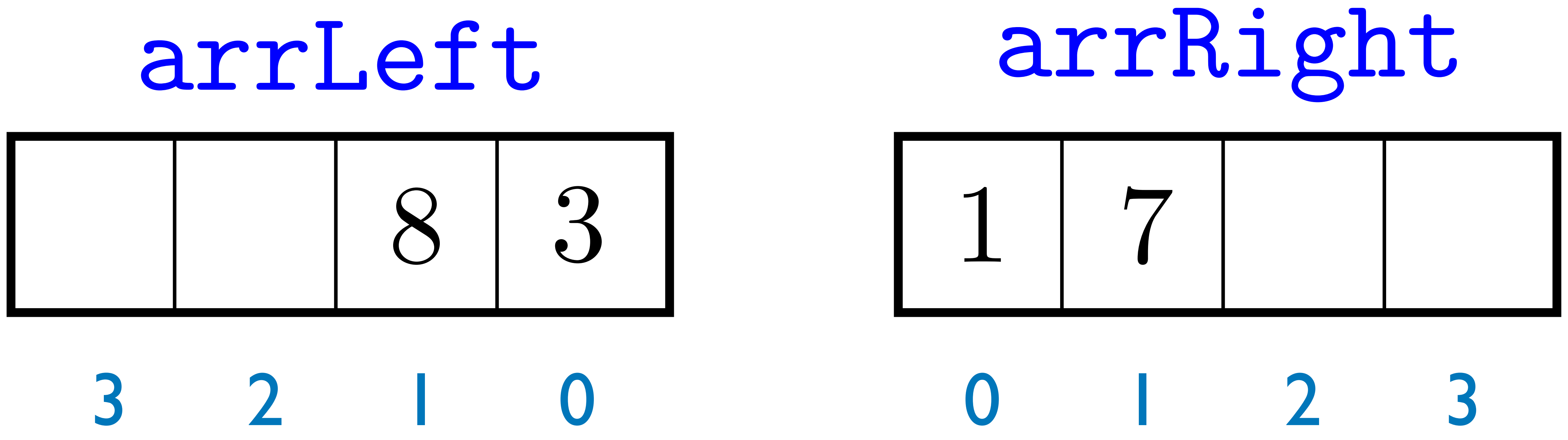
Size



$$A.\texttt{size}() == \texttt{arrLeft.size}() + \texttt{arrRight.size}()$$

This is constant time as getting the size of a resizable array is constant time.

Get/Set



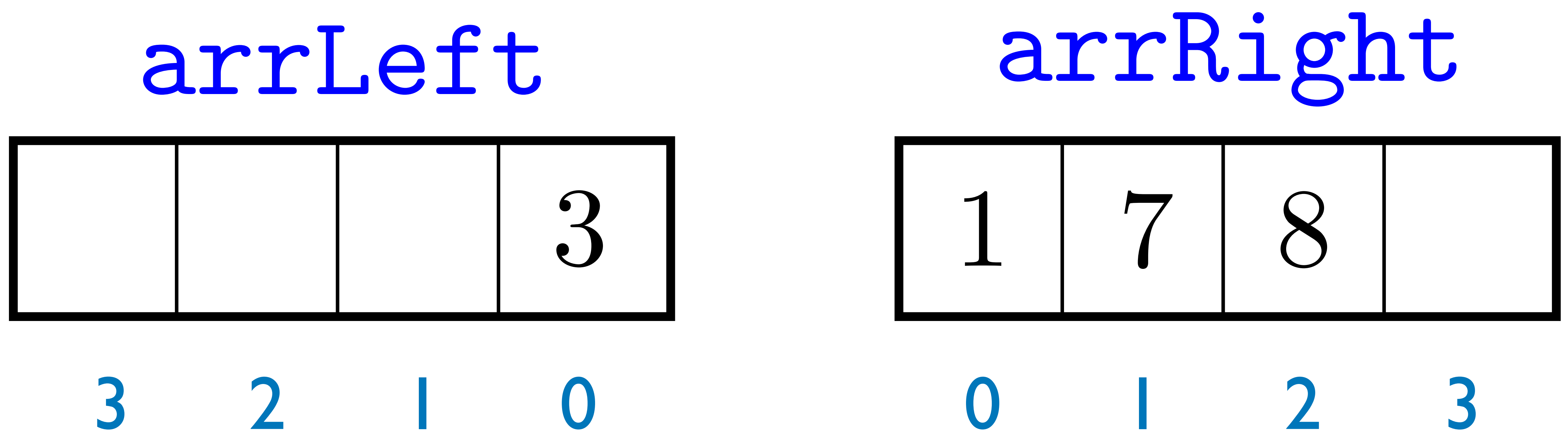
if $i < \texttt{arrLeft.size}()$ then

$$A.\texttt{get(i)} == \texttt{arrLeft.get}(\texttt{arrLeft.size}() - \texttt{i} - 1)$$

else

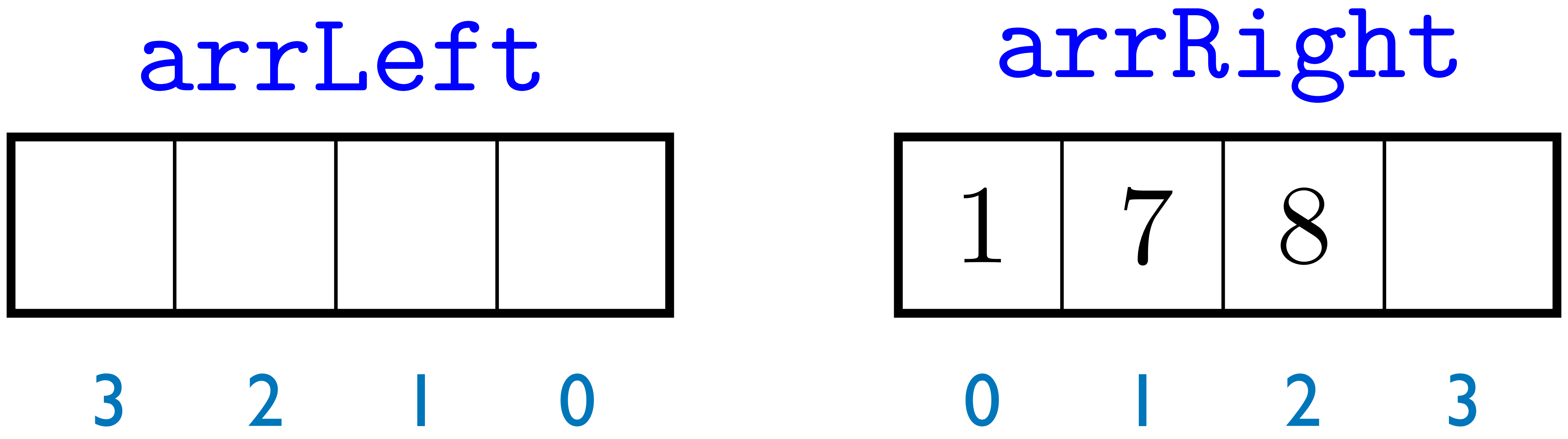
$$A.\texttt{get(i)} == \texttt{arrRight.get}(\texttt{i} - \texttt{arrLeft.size}())$$

Interesting Case



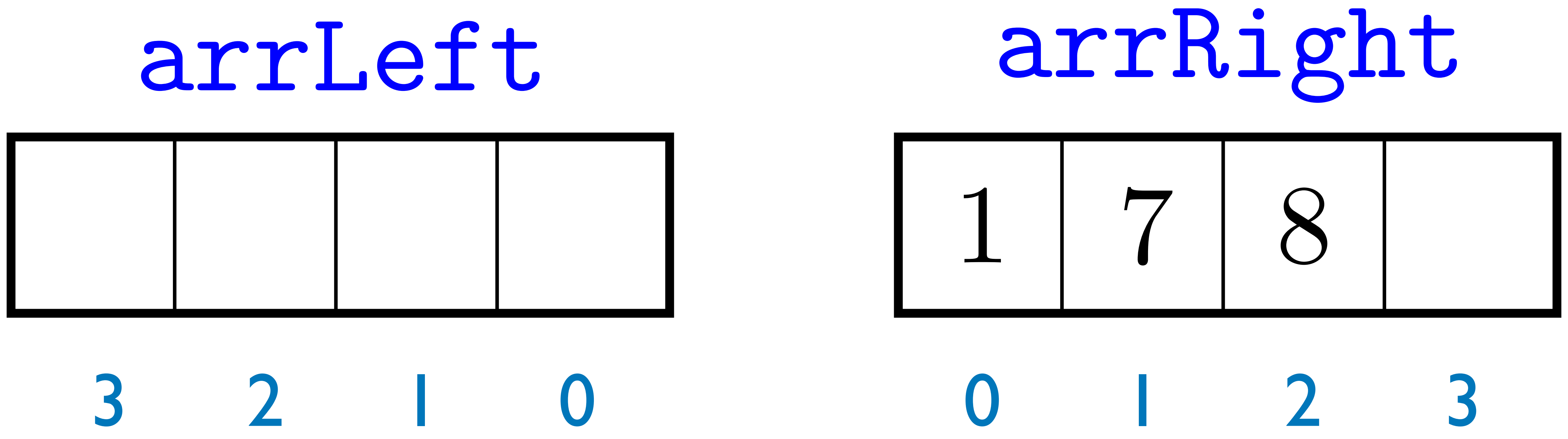
$A.\texttt{pop\_front}()$

Interesting Case



$A.\texttt{pop\_front}()$

Interesting Case

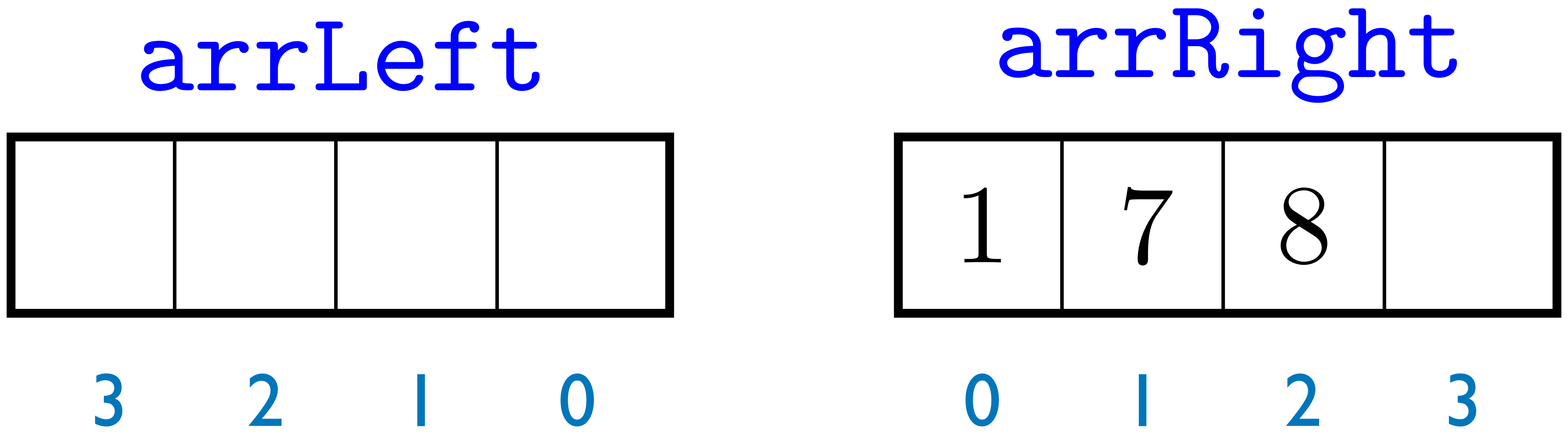


$A.\texttt{pop\_front}()$

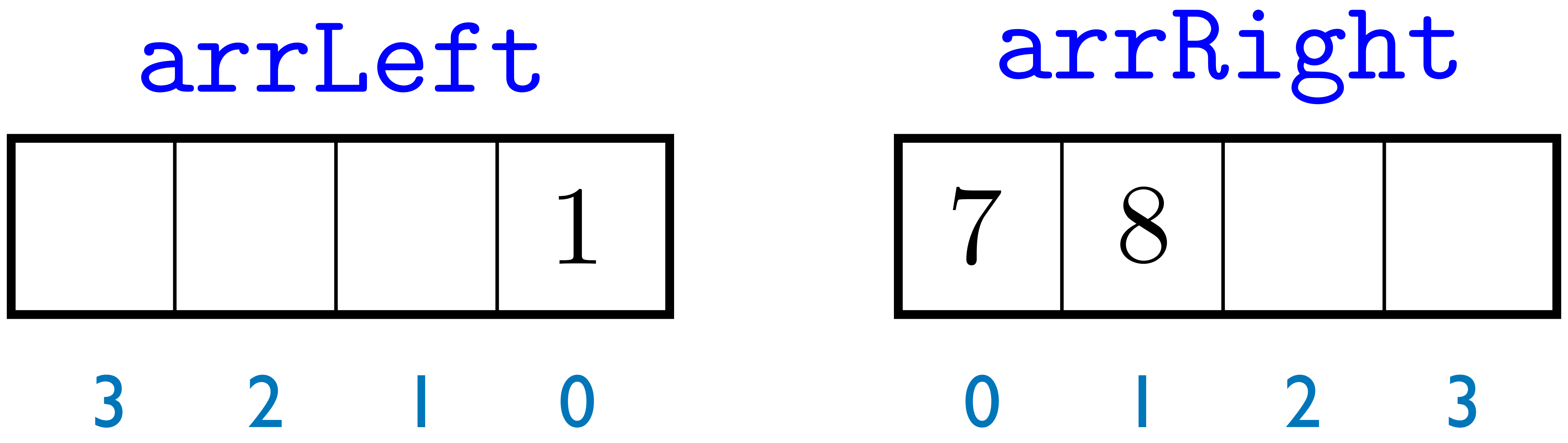
The left array is empty. What do we do now?

We cannot remove the first item from the right array because that is not an allowed operation on a resizable array.

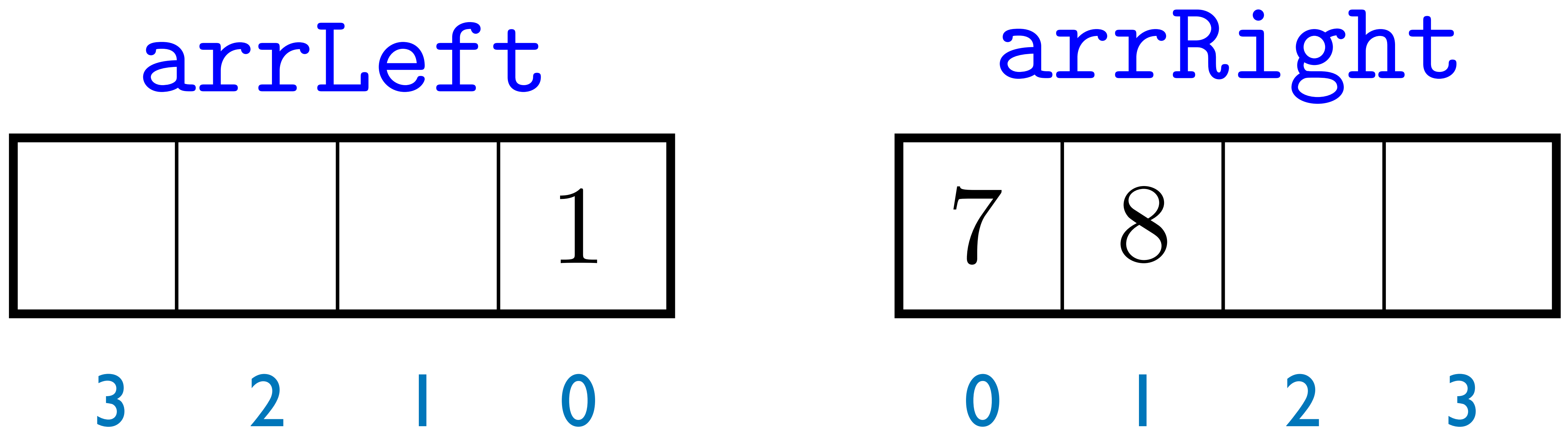
Rebalance



We can rebalance the n elements so that $\lfloor n/2 \rfloor$ are in the left array and $\lceil n/2 \rceil$ are in the right array.



Rebalance

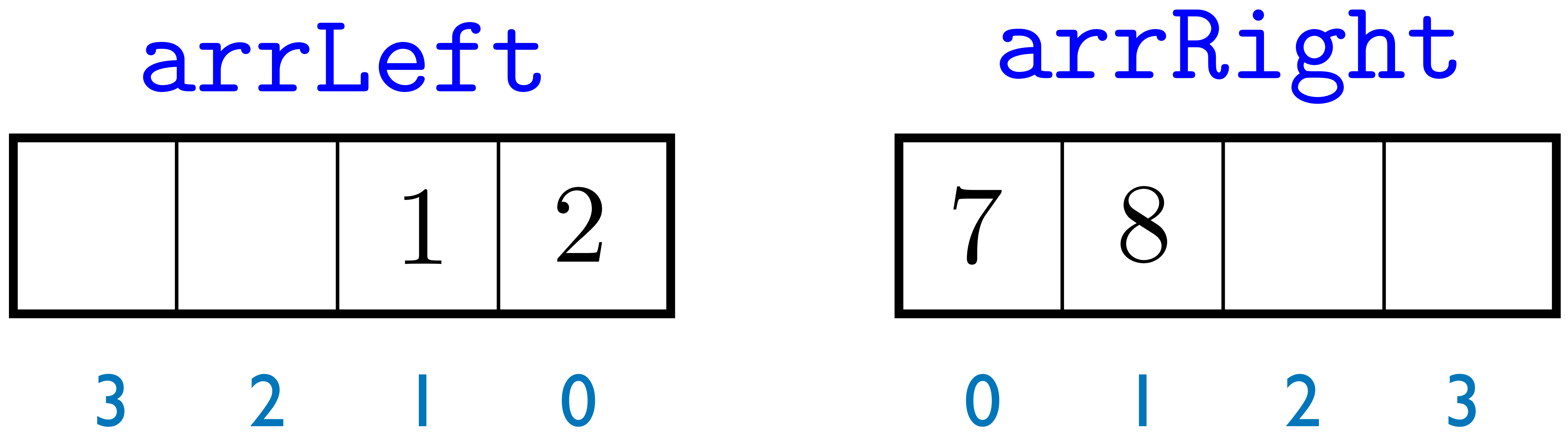


Like `push_back` when there is no excess capacity, rebalancing is an expensive operation.

The time is proportional to the number of elements in the container at the time of rebalancing.

We can still argue that the amortized complexity of `pop_front` is constant.

Rebalance: Amortized

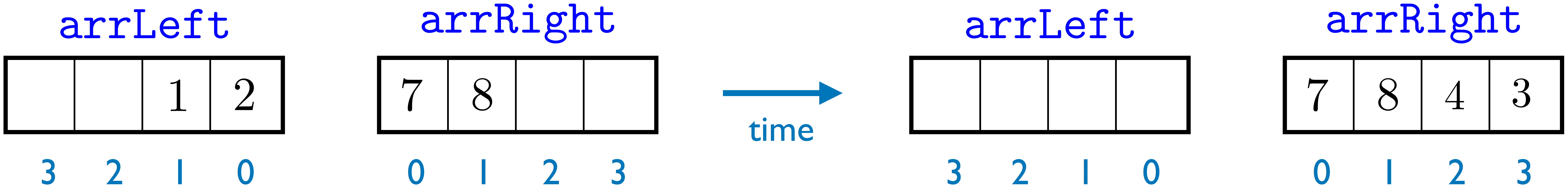


After reblancing, `arrLeft` has $\lfloor n/2 \rfloor$ elements and `arrRight` has $\lceil n/2 \rceil$ elements.

How many push/pop operations T must we do until the next rebalance?

Say that `arrLeft` is the one that becomes empty. At the next rebalancing the size of the deque will be $S = \texttt{arrRight.size}()$.

Rebalance: Amortized



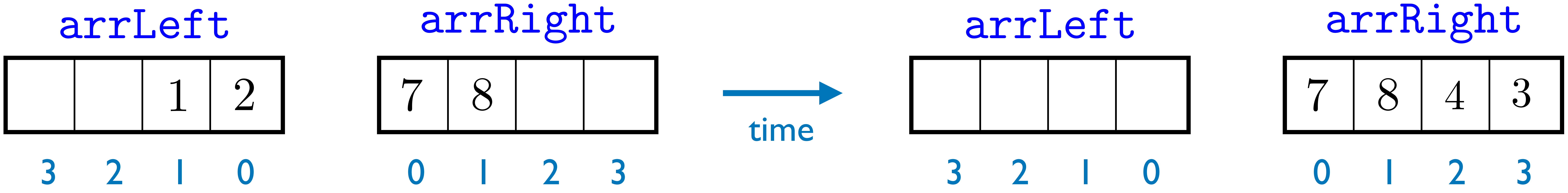
We do T push/pop operations until the next rebalancing.

The size of the deque at the time of rebalancing is $S = \texttt{arrRight.size}()$.

The total time of operations plus rebalancing is at most a constant times

$$S + T$$

Rebalance: Amortized

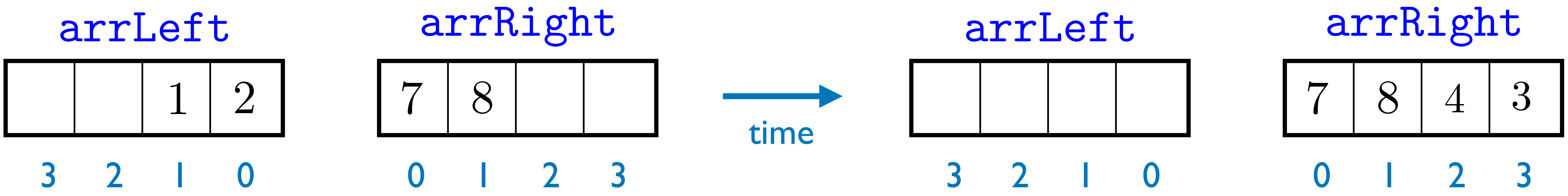


We do T push/pop operations until the next rebalancing.

The size of the deque at the time of rebalancing is $S = \texttt{arrRight.size}()$.

We claim that $T \geq S - 1$.

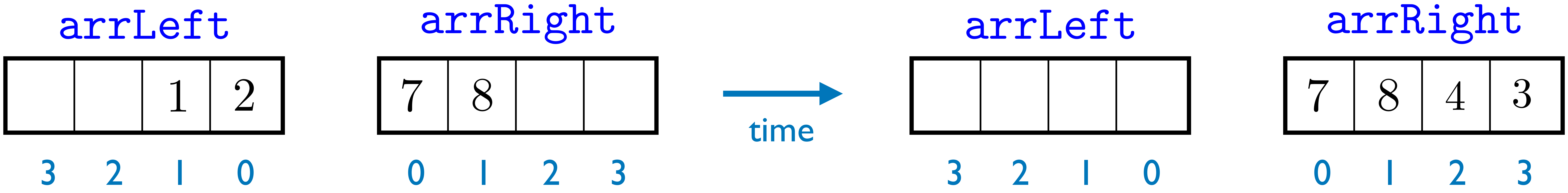
Rebalance: Amortized



To empty the left array we must do at least $\lfloor n/2 \rfloor$ `pop_front` operations.

We must also do at least $|S - \lceil n/2 \rceil|$ push/pop back operations to change the size of `arrRight` to S.

Thus $T \geq \lfloor n/2 \rfloor + S - \lceil n/2 \rceil \geq S - 1$.



To recap: Say we do T push/pop front/back operations before the next rebalancing and let S be the size of the deque when we rebalance.

We have argued that $T \geq S - 1.$

Each push/pop front/back (without rebalancing) is constant amortized time, so the total time for all these is at most a constant times T. The time to rebalance is at most a constant times S.

The total time for the T operations is at most a constant times T.

Deque Summary

We have seen how to implement a deque in a black-box way with two resizable arrays.

The time for get/set and size is constant.

The time for push/pop front and push/pop back is amortized constant.

For push front/back this is inherited from our implementation of a resizable array.

For pop front/back this is due to the need to periodically rebalance.

std:deque

The way we have described implementing a deque with resizable arrays is not compatible with the C++ standard.

The C++ standard states that no references in a deque are invalidated when doing, for example, a push front operation.

When using a resizable array, references would be invalidated when we have to double the size of the array.

A typical C++ implementation of a deque uses a linked list of pointers to fixed sized arrays.