Hash Tables

Contains Duplicate

Let's go back to the contains duplicate problem.

The input is a vector of integers and we want to know if any integer appears more than once.

We can solve this problem with any data structure that lets us do the operations of:

$\texttt{insert}(\texttt{x})$ —add x to the data structure.

$\texttt{contains}(\texttt{x})$ —check if x is present in the data structure.

This is the bread and butter of a hash table.

Dictionary ADT

A hash table implements the dictionary abstract data type:

$A \leftarrow \texttt{Dict}()$	Create an empty dictionary.
$A.\texttt{insert}(\texttt{x})$	Add $\texttt{x}$ to the dictionary.
$A.\texttt{contains}(\texttt{x})$	Return if $\texttt{x}$ is in the dictionary or not.
$A.\texttt{erase}(\texttt{x})$	Remove $\texttt{x}$ from the dictionary.

First Idea

Let's say that the input vector `nums` has size 10^5 and we are promised that the entries of the vector satisfy $0 \leq \texttt{nums}[i] < 10^9$.

We could initialize a boolean vector `present` of size 10^9 to all false.

We make one pass through `nums` like this:

```
for (int x : nums) {
    if (present[x]) {
        return true;
    }
    present[x] = true;
}
```

This code exceeds the time limit on Leet Code!

Problem

The problem is that we are initializing a huge vector `present` of size 10^9.

As the input `nums` has size at most 10^5, most entries in `present` will never be touched.

Can we make use of this fact?

Let's try to get by with using a smaller vector to store the entries we have seen.

Second Idea

Let's say we can only afford to initialize a vector of size 10^6.

As the input $\texttt{nums}$ has size at most 10^5, this is still 10 times the number of values we need to store.

Now we need a way to map a value $0 \leq \texttt{nums}[i] <= 10^9$ to an index i with $0 \leq i < 10^6$.

We can do this with the modulus function.

$$\texttt{index} = \texttt{x} \,\%\, \texttt{SIZE}$$

Implementation

Godbolt link

```
bool containsDuplicate(std::vector<int>& nums) {
    std::size_t SIZE = 999'863;
    std::vector<bool> present(SIZE);
    for (int x : nums) {
        std::size_t index = x % SIZE;
        if (present[index]) {
            return true;
        }
        present[index] = true;
    }
    return false;
}
```

This code now passes all the Leet Code tests*!

*Code on slide is simplified assuming $x \geq 0$, see Godbolt link for code passing Leet Code tests.

Issue

Godbolt link

```
bool containsDuplicate(std::vector<int>& nums) {
    std::size_t SIZE = 999'863;
    std::vector<bool> present(SIZE);
    for (int x : nums) {
        std::size_t index = x % SIZE;
        if (present[index]) {
            return true;
        }
        present[index] = true;
    }
    return false;
}
```

With this code we are getting very lucky!

If there are two integers $\mathtt{x} \neq \mathtt{y}$ in the input with $\mathtt{x} \ \% \ \mathtt{SIZE} = \mathtt{y} \ \% \ \mathtt{SIZE}$ we would say there is a duplicate when there might not be.

Collisions

We have $\texttt{SIZE}$ buckets labeled by integers $0, 1, 2, \ldots, \texttt{SIZE} - 1$.

Given a non-negative integer $\texttt{x}$, we place it in bucket $\texttt{x} \; \% \; \texttt{SIZE}$.

1 $\quad$ $5 * \texttt{SIZE} + 1$

Two distinct integers going into the same bucket is a collision.

Why Use The Modulus?

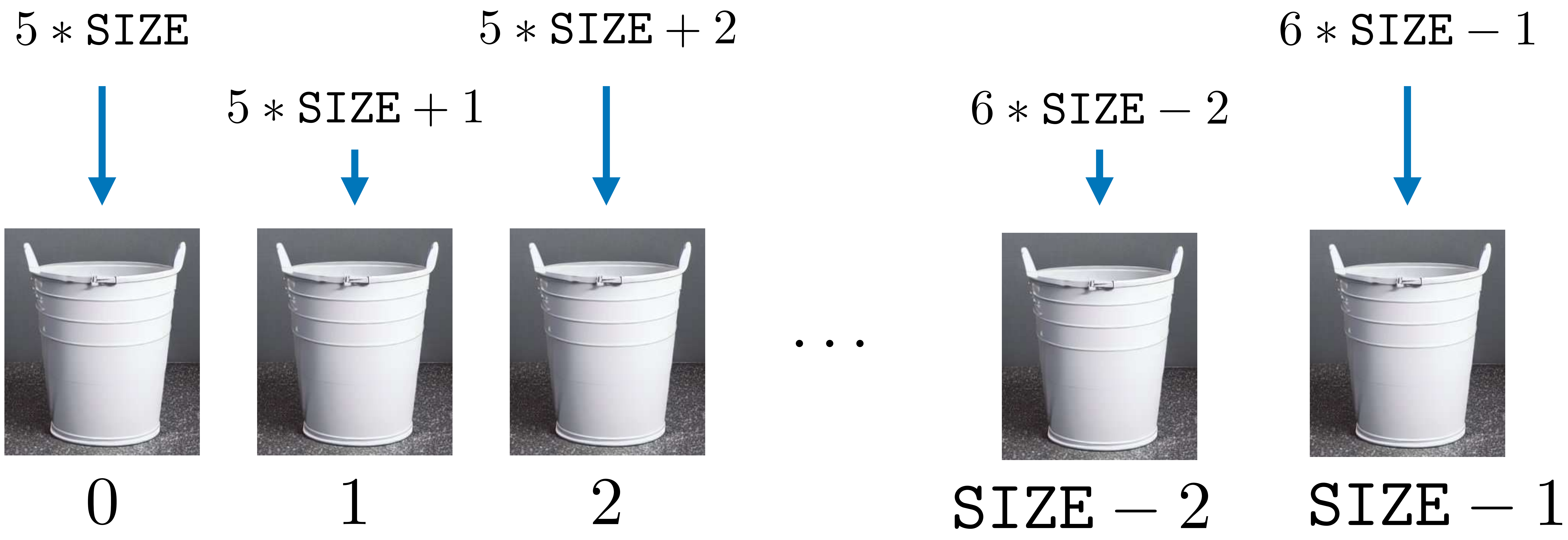
1) We need a function that maps non-negative integers to buckets.

A function which maps some data type to the label of a bucket is called a hash function.

Why Use The Modulus?

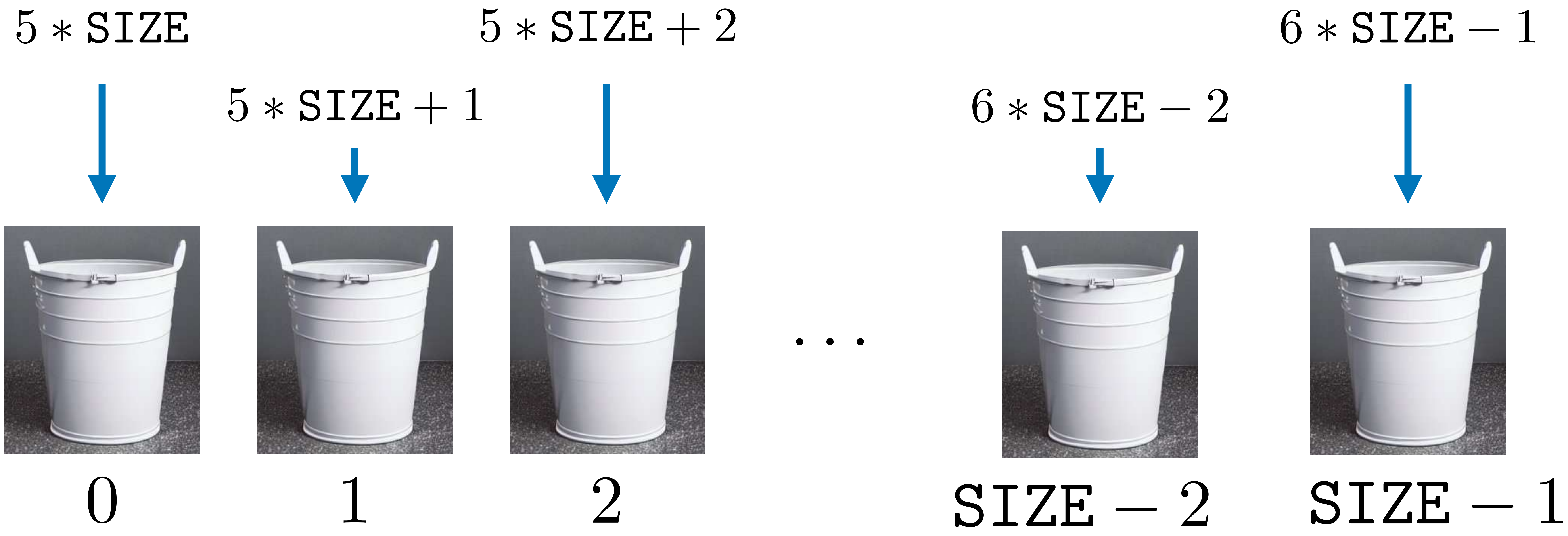
2) A nice property of the modulus function is that when we choose a non-negative integer `x` at random, `x % SIZE` is distributed uniformly over the buckets.

→ This reduces the chance of collisions.



Why Use The Modulus?

3) A hash function should also be fast to compute, and the modulus function is relatively fast to compute.



Load Factor

We don't want to have too many buckets. If we are going to store N numbers, we would like the number of buckets to be, say, $2N$.

The ratio of the number of numbers stored to the number of buckets is called the load factor.

With a constant load factor, we expect to have collisions. We need a way of dealing with them.

Collision Handling

Rather than just saying if a bucket has a number or not, let's remember all the numbers in the bucket.

Each bucket can have its own data structure, say a deque, to store all the elements in that bucket.

For each new element we first compute what bucket it should be in. We then add it to the front of the deque stored at that bucket.

Collision Handling

For each new element we first compute what bucket it should be in. We then add it to the front of the deque stored at that bucket.

$$\texttt{SIZE} + 1$$

0 1 2 $\ldots$ $\texttt{SIZE} - 2$ $\texttt{SIZE} - 1$

Collision Handling

For each new element we first compute what bucket it should be in. We then add it to the vector stored at that bucket.

Collision Handling

For each new element we first compute what bucket it should be in. We then add it to the vector stored at that bucket.

$$10 * \mathtt{SIZE} + 1$$

0

1

2

$\ldots$

$\mathtt{SIZE} - 2$

$\mathtt{SIZE} - 1$

$$\mathtt{SIZE} + 1$$

$$5 * \mathtt{SIZE} - 1$$

Collision Handling

For each new element we first compute what bucket it should be in. We then add it to the vector stored at that bucket.

0

1

2

$\dots$

$\texttt{SIZE} - 2$

$\texttt{SIZE} - 1$

$10 * \texttt{SIZE} + 1$

$\texttt{SIZE} + 1$

$5 * \texttt{SIZE} - 1$

Hash Table

We have now built a hash table.

The basic components are:

1) Hash function to map the data to an array index.

2) Mechanism to handle hash collisions.

The method of having a data structure at each bucket to store the elements mapped there is called separate chaining.

Homemade Hash Table

Let's try this again on contains duplicate. Godbolt Link

```
bool containsDuplicate(std::vector<int>& nums) {
  const std::size_t SIZE {2*nums.size()};
  std::vector<std::deque<int> > buckets(SIZE);
  for (int x : nums) {
    std::size_t bucket = x % SIZE;
    auto iter = std::find(buckets[bucket].begin(),
                          buckets[bucket].end(), x);
    if (iter != buckets[bucket].end()) {
      return true;
    }
    buckets[bucket].push_front(x);
  }
  return false;
}
```

← load factor of 1/2

Homemade Hash Table

Let's try this again on contains duplicate.

```
bool containsDuplicate(std::vector<int>& nums) {
  const std::size_t SIZE {2*nums.size()};
  std::vector<std::deque<int> > buckets(SIZE);
  for (int x : nums) {
    std::size_t bucket = x % SIZE;
    auto iter = std::find(buckets[bucket].begin(),
                          buckets[bucket].end(), x);
    if (iter != buckets[bucket].end()) {
      return true;
    }
    buckets[bucket].push_front(x);
  }
  return false;
}
```

our hash table is a vector of deques

Homemade Hash Table

Let's try this again on contains duplicate.

```cpp
bool containsDuplicate(std::vector<int>& nums) {
  const std::size_t SIZE {2*nums.size()};
  std::vector<std::deque<int> > buckets(SIZE);
  for (int x : nums) {
    std::size_t bucket = x % SIZE;
    auto iter = std::find(buckets[bucket].begin(),
                          buckets[bucket].end(), x);
    if (iter != buckets[bucket].end()) {
      return true;
    }
    buckets[bucket].push_front(x);
  }
  return false;
}
```

compute the bucket
(assuming $x \geq 0$)

Homemade Hash Table

Let's try this again on contains duplicate.

```
bool containsDuplicate(std::vector<int>& nums) {
  const std::size_t SIZE {2*nums.size()};
  std::vector<std::deque<int> > buckets(SIZE);
  for (int x : nums) {
    std::size_t bucket = x % SIZE;
    auto iter = std::find(buckets[bucket].begin(),
                          buckets[bucket].end(), x);
    if (iter != buckets[bucket].end()) {
      return true;
    }
    buckets[bucket].push_front(x);
  }
  return false;
}
```

do linear search to see if `x` already in bucket

Homemade Hash Table

Let's try this again on contains duplicate.

```
bool containsDuplicate(std::vector<int>& nums) {
  const std::size_t SIZE {2*nums.size()};
  std::vector<std::deque<int> > buckets(SIZE);
  for (int x : nums) {
    std::size_t bucket = x % SIZE;
    auto iter = std::find(buckets[bucket].begin(),
                          buckets[bucket].end(), x);
    if (iter != buckets[bucket].end()) {
      return true;
    }
    buckets[bucket].push_front(x);
  }
  return false;
}
```

if `x` is in the bucket, return true

Homemade Hash Table

Let's try this again on contains duplicate.

```
bool containsDuplicate(std::vector<int>& nums) {
  const std::size_t SIZE {2*nums.size()};
  std::vector<std::deque<int> > buckets(SIZE);
  for (int x : nums) {
    std::size_t bucket = x % SIZE;
    auto iter = std::find(buckets[bucket].begin(),
                          buckets[bucket].end(), x);
    if (iter != buckets[bucket].end()) {
      return true;
    }
    buckets[bucket].push_front(x);
  }
  return false;
}
```

← otherwise, add it to the front of the bucket

Homemade Hash Table

Let's try this again on contains duplicate.

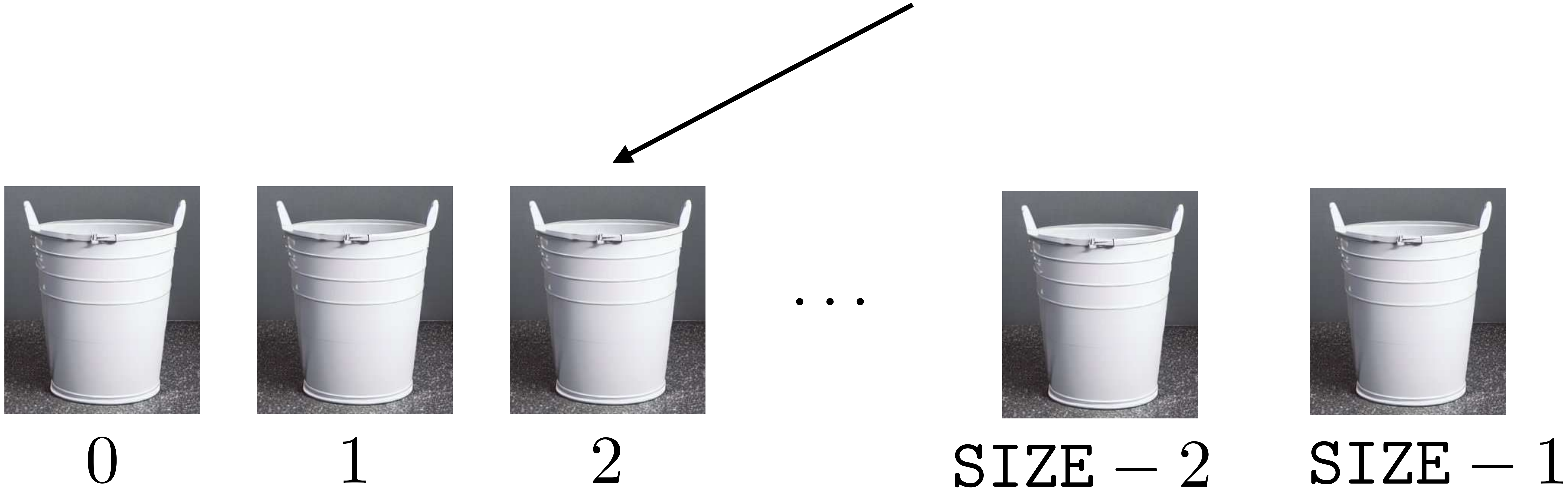
```
bool containsDuplicate(std::vector<int>& nums) {
  const std::size_t SIZE {2*nums.size()};
  std::vector<std::deque<int> > buckets(SIZE);
  for (int x : nums) {
    std::size_t bucket = x % SIZE;
    auto iter = std::find(buckets[bucket].begin(),
                          buckets[bucket].end(), x);
    if (iter != buckets[bucket].end()) {
      return true;
    }
    buckets[bucket].push_front(x);
  }
  return false;
}
```

← if never found a duplicate, return false

Efficiency

Let's focus on the contains function of our homemade hash table.

$\texttt{contains}(x)$

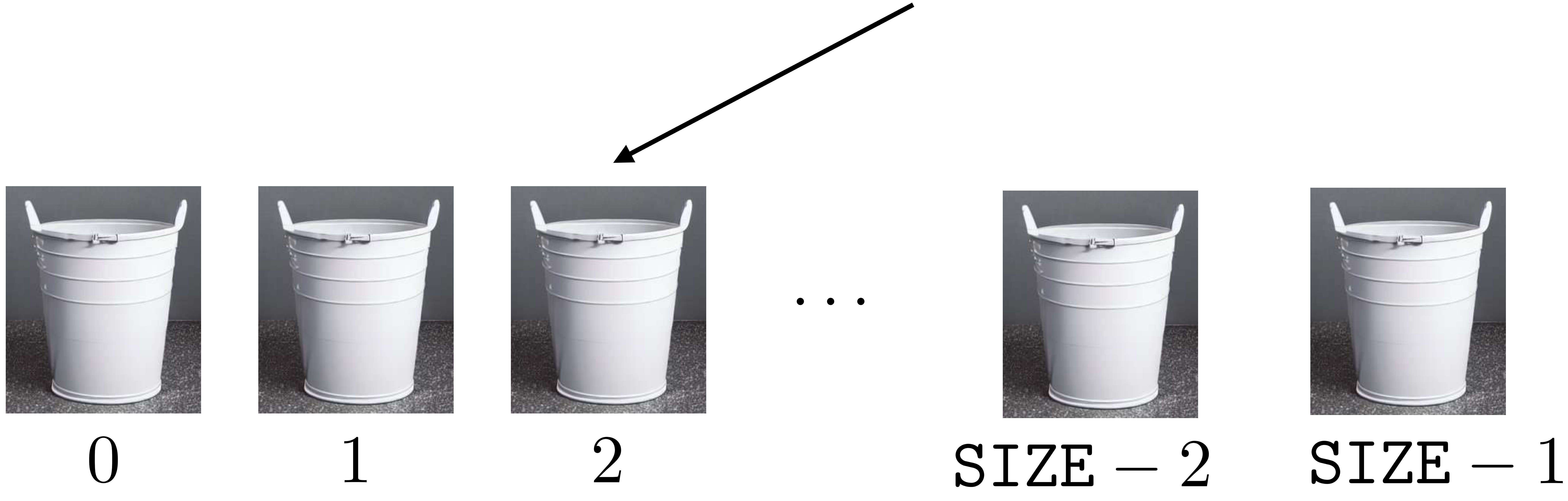


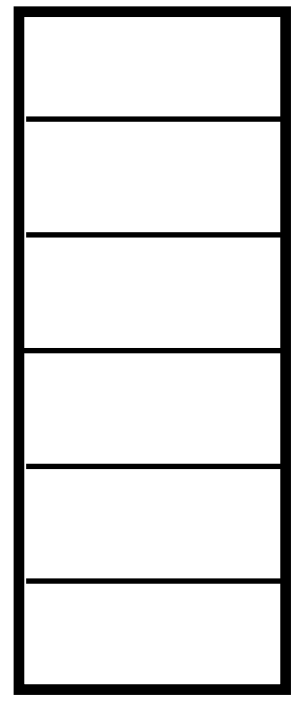
Step 1: compute $\texttt{hash}(x)$ in constant time.

Efficiency

Let's focus on the contains function of our homemade hash table.

$$\texttt{contains}(x)$$





Step 2: linear search through elements stored at the bucket.

time proportional to size of the bucket.

Efficiency

The complexity of contains is proportional to how many elements are stored in a bucket.

This is why we want a hash function which evenly distributes the data across the buckets.

In the worst-case, all the inputs are mapped to the same bucket! Then our hash table degrades to a deque and contains takes worst-case time $\Theta(n)$.

To argue a hash table has good performance we need to make some assumption about the inputs.

Simple Uniform Hashing

Simple Uniform Hashing (SUH): Any given element is equally likely to hash to any of the $\mathtt{SIZE}$ buckets.

This is true under modular hashing when the elements are random non-negative integers.

With SUH, after inserting n elements, the expected size of each bucket is $n/\mathtt{SIZE}$.

With a constant load factor, under SUH the average time taken by contains is constant.

Insertion

There are several additional considerations that go into the running time of hash table insertion.

- Is the hash table of fixed size or is it dynamically resized?

  To dynamically resize we can use an array doubling technique.

  Then we can only hope to get constant amortized insertion time.

- Do we search for the element before inserting?

  We may want to so that we do not have duplicate elements, or to keep records with equal keys grouped together.

In a fixed-size hash table that does not search before insertion we can achieve worst-case constant time insertion.

Hash Table in C++

Finally, let's see how to solve contains duplicate using a hash table in the standard library.

We can use `std::unordered_set` in the library `<unordered_set>`.

This maintains a set of unique keys—if you try to insert the same key again nothing happens.

```
bool containsDuplicate(std::vector<int>& nums) {
  std::unordered_set<int> setto {};
  for (int x : nums) {
    if (setto.contains(x)) {
      return true;
    }
    setto.insert(x);
  }
  return false;
}
```

Godbolt Link

Hash Tables in C++

Hash tables in C++ are required by the standard to store elements in buckets, and that elements that hash to the same value are kept in the same bucket.

This essentially describes a separate chaining implementation.

Bucket interface

begin(size_type) **cbegin**(size_type) (C++11)	returns an iterator to the beginning of the specified bucket (public member function)
end(size_type) **cend**(size_type) (C++11)	returns an iterator to the end of the specified bucket (public member function)
bucket_count (C++11)	returns the number of buckets (public member function)
max_bucket_count (C++11)	returns the maximum number of buckets (public member function)
bucket_size (C++11)	returns the number of elements in specific bucket (public member function)
bucket (C++11)	returns the bucket for specific key (public member function)

cppreference

In `std::unordered_set` of contains and erase take average case constant time, and insert takes amortized average case constant time, under the input assumption mentioned earlier.

They have worst-case running time $\Theta(n)$.

Valid Parentheses

Valid Parentheses

Leetcode 20 (easy, Blind75) Valid Parentheses:

Given a string over the 6 characters ()[]{}, determine if it is a valid parenthesization.

Examples:

(){}	valid
((){[]})	valid
}()[]	invalid: first closing } has no partner
{[}]	invalid: closing } is matched by [

Valid Parentheses

Leetcode 20 (easy, Blind75) Valid Parentheses:

Given a string over the 6 characters ()[]{}, determine if it is a valid parenthesization.

Formal definition:

The empty string is valid.

If s is valid then $(s), [s], \{s\}$ are valid.

If s, t are valid then the concatenation st is valid.

Key Idea

Once we find a valid substring, we can remove it from the string. The original is valid if and only if the remainder is.

Examples:

(()**{[]}**) → (())

valid → still valid

[(**{[]}**)} → [()}

invalid → still invalid

Easy Valid Substrings

The easiest valid substrings to find are an opening symbol followed by a closing symbol of the same type.

Examples: () {} [ ]

If an opening symbol is followed by a closing symbol of a different type, we immediately know the string is invalid.

Example: A valid string cannot contain the substring (}

First Algorithm

This suggests a first algorithm:

Go through the string looking for an opening symbol immediately followed by a closing symbol.

If they match, remove them. If they don't match, return invalid.

Repeat until the string is empty. If you end up with the empty string, return valid.

Example

First pass:

((){[]})

Example

First pass:

((){[]})

↓

({[]})

Example

First pass:

({[]})

↓

({})

Example

Second pass:

({})

↓

()

Example

Third pass:

()

↓

empty string

We have reached the empty string, so we return valid.

Efficiency

This algorithm can be slow!

(((((((())))))))

On a valid string like this of size n we have to make $n/2$ passes.

Maybe if we remember some information along the way, we don't have to start back at the beginning after each pass?

Remember the openers

Goal: Remember the most recent opening symbol we have seen dynamically, as we modify the string by removing matching pairs.

Example:

(() {[]})

Read Symbol: (

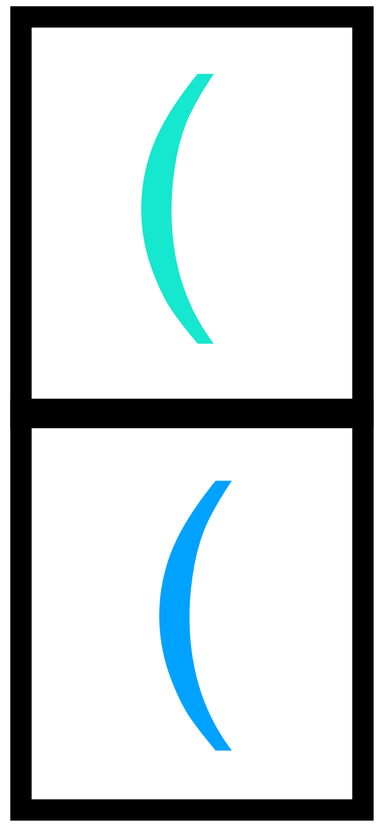
(

Remember the openers

Goal: Remember the most recent opening symbol we have seen dynamically, as we modify the string by removing matching pairs.

Example: ((){[]})

Read Symbol: (



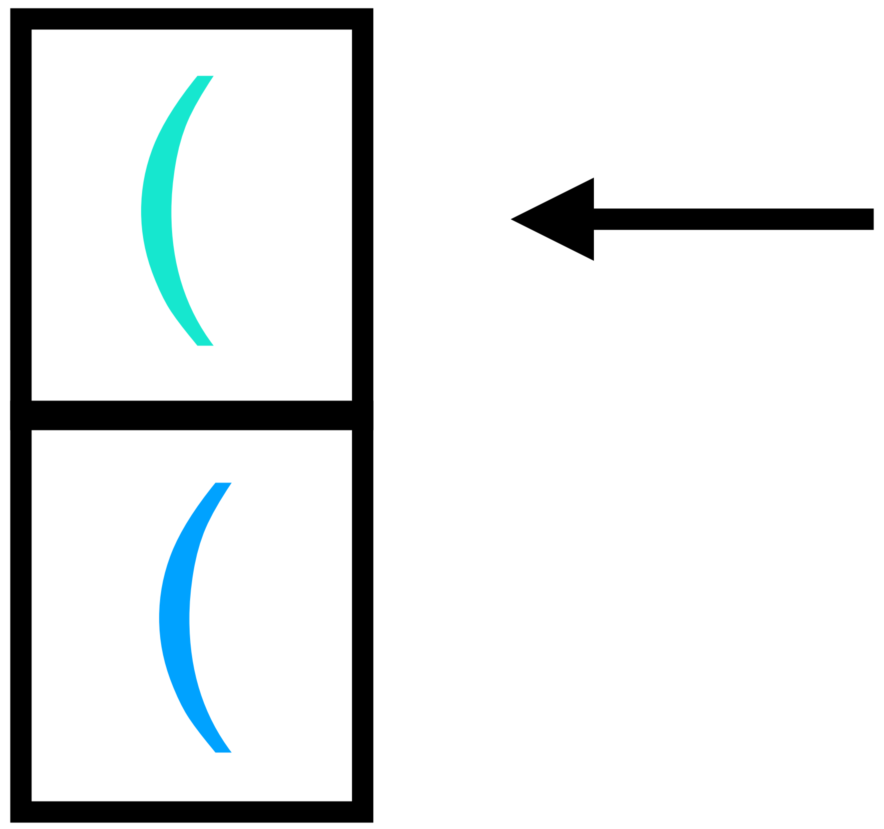
Remember the openers

Goal: Remember the most recent opening symbol we have seen dynamically, as we modify the string by removing matching pairs.

Example:

(( ){[]})

Read Symbol:)



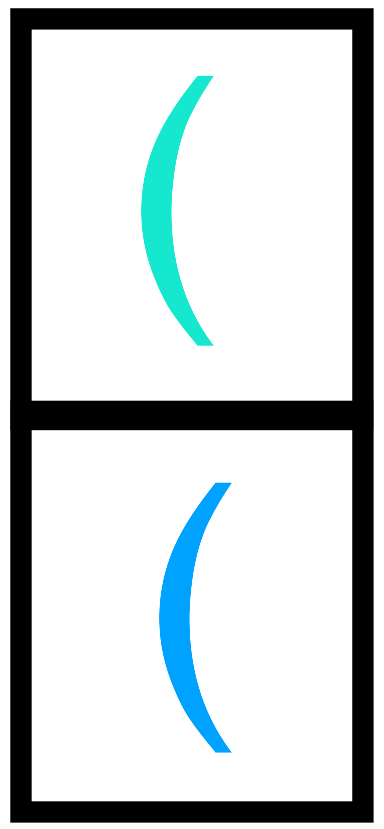
matches the most recent opener.

Remember the openers

Goal: Remember the most recent opening symbol we have seen dynamically, as we modify the string by removing matching pairs.

Example: (()\{[]\}) → (\{[]\})

Read Symbol:)



Remove matched pair from the string.

← in the new string, this is the most recent opener.

Remember the openers

Goal: Remember the most recent opening symbol we have seen dynamically, as we modify the string by removing matching pairs.

Example: ({[]})

Read Symbol: {

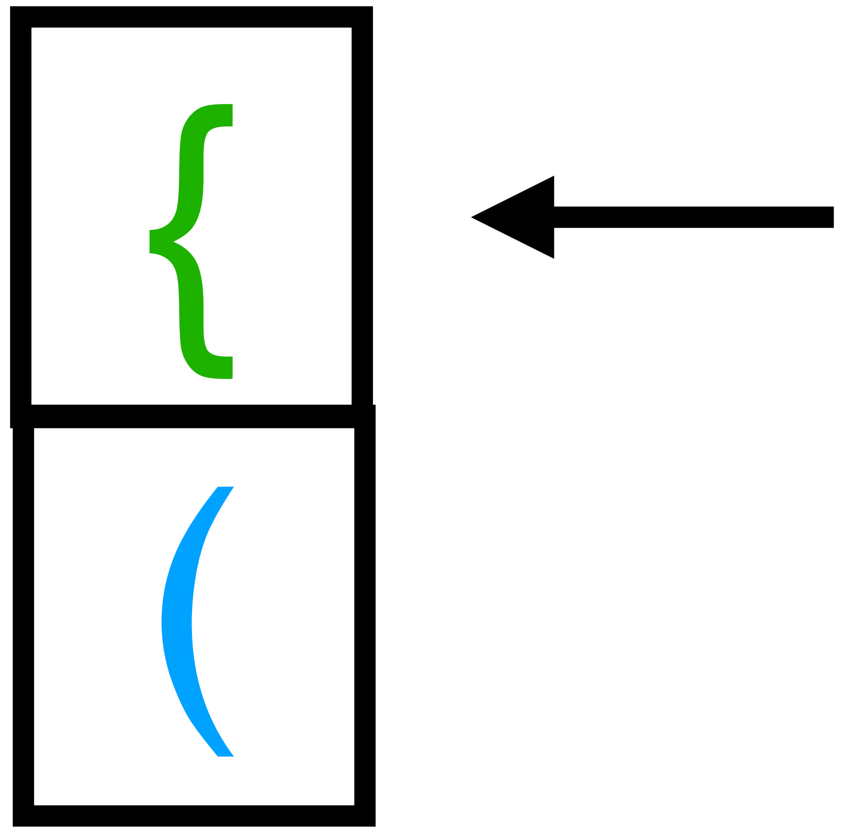
(← in the new string, this is the most recent opener.

Remember the openers

Goal: Remember the most recent opening symbol we have seen dynamically, as we modify the string by removing matching pairs.

Example: ({[]})

Read Symbol: {



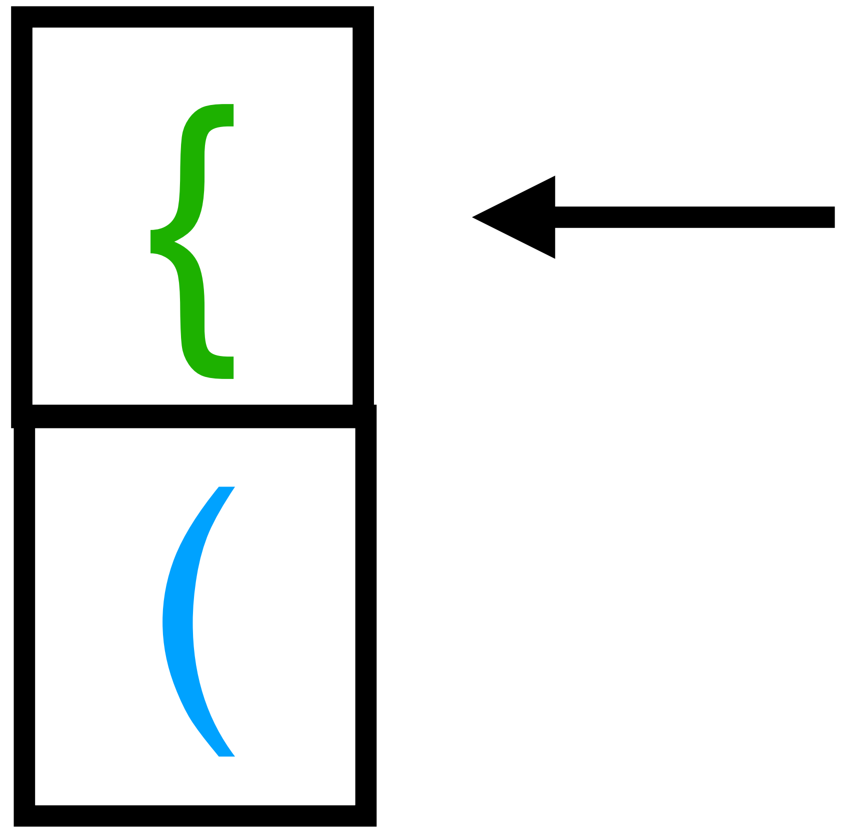
most recent opener

Remember the openers

Goal: Remember the most recent opening symbol we have seen dynamically, as we modify the string by removing matching pairs.

Example: ({[]})

Read Symbol: [



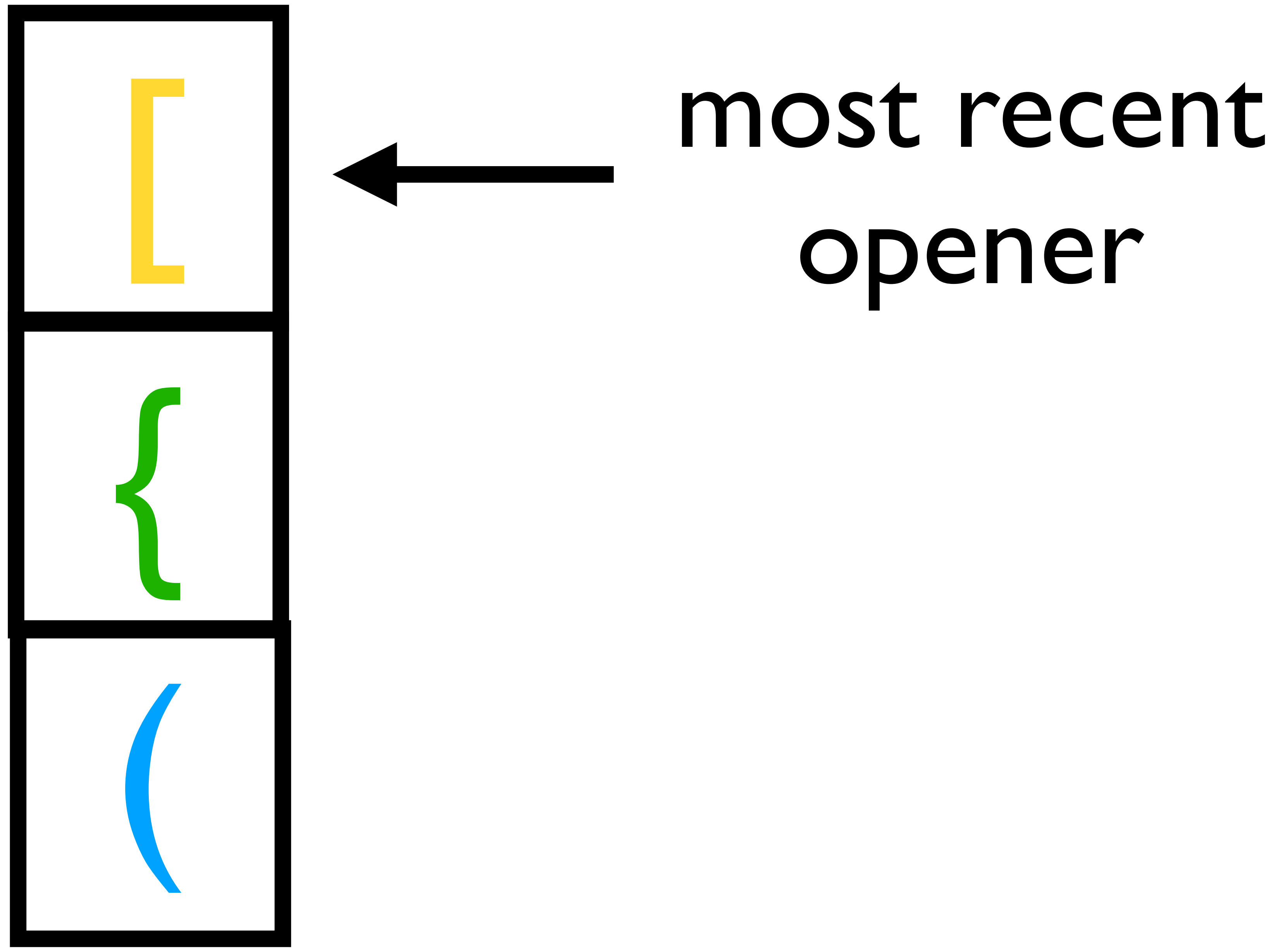
most recent opener

Remember the openers

Goal: Remember the most recent opening symbol we have seen dynamically, as we modify the string by removing matching pairs.

Example:

({[]})



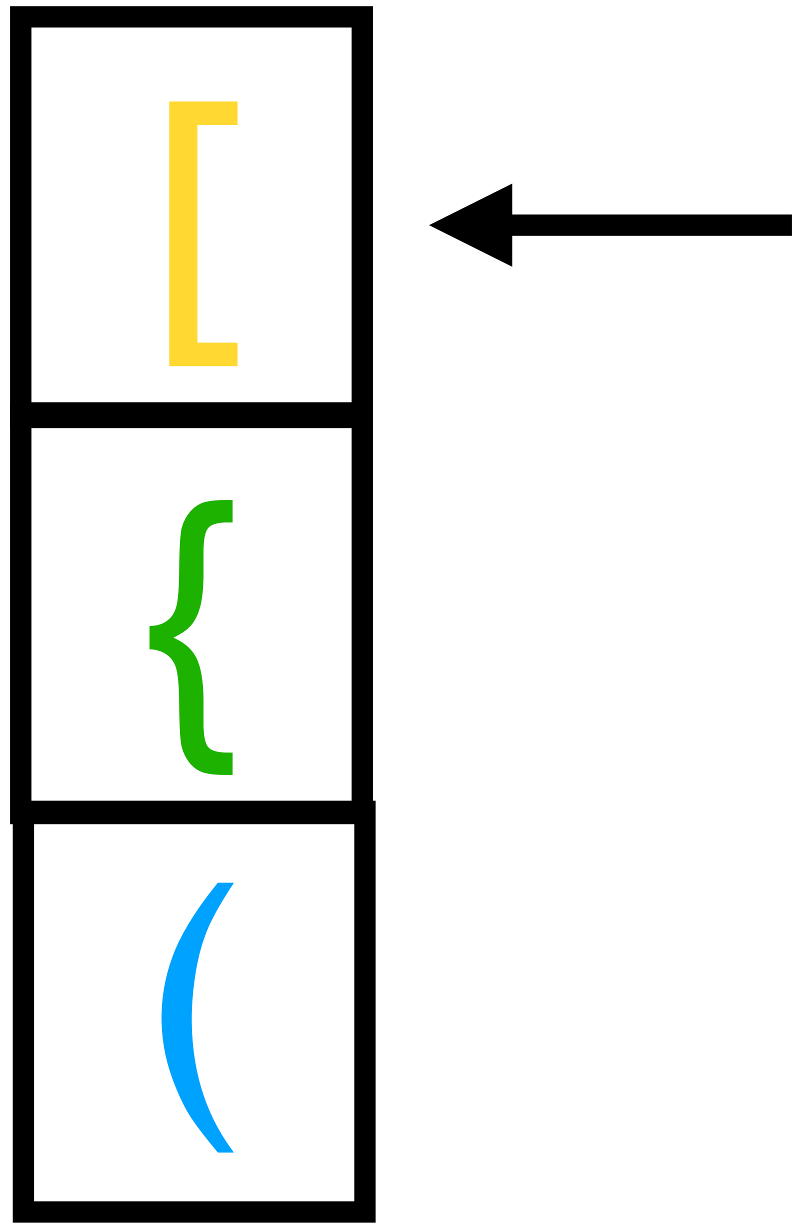
Read Symbol: [

Remember the openers

Goal: Remember the most recent opening symbol we have seen dynamically, as we modify the string by removing matching pairs.

Example:

({[]})



most recent opener

Read Symbol:]

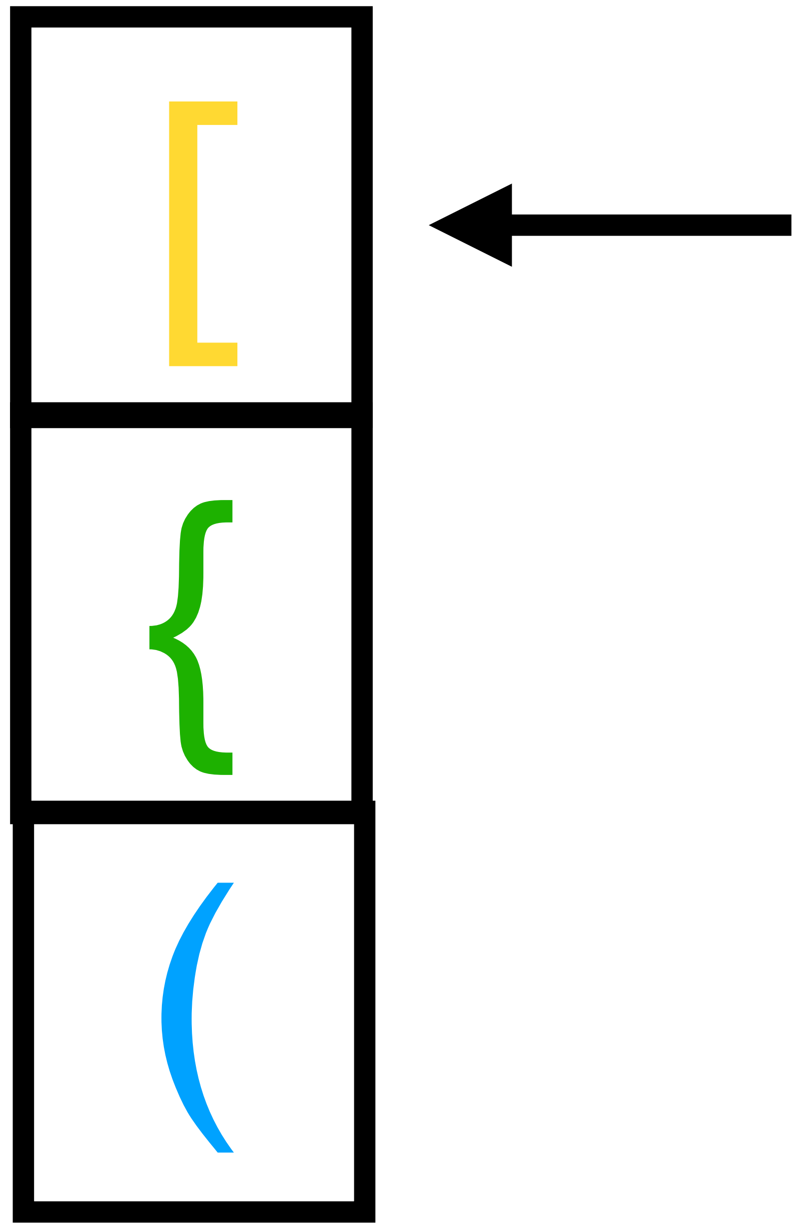
Closing symbol. Does it match most recent opener?

Remember the openers

Goal: Remember the most recent opening symbol we have seen dynamically, as we modify the string by removing matching pairs.

Example:

({[]}) → ({})



most recent opener

Read Symbol:]

Removing matching pair.

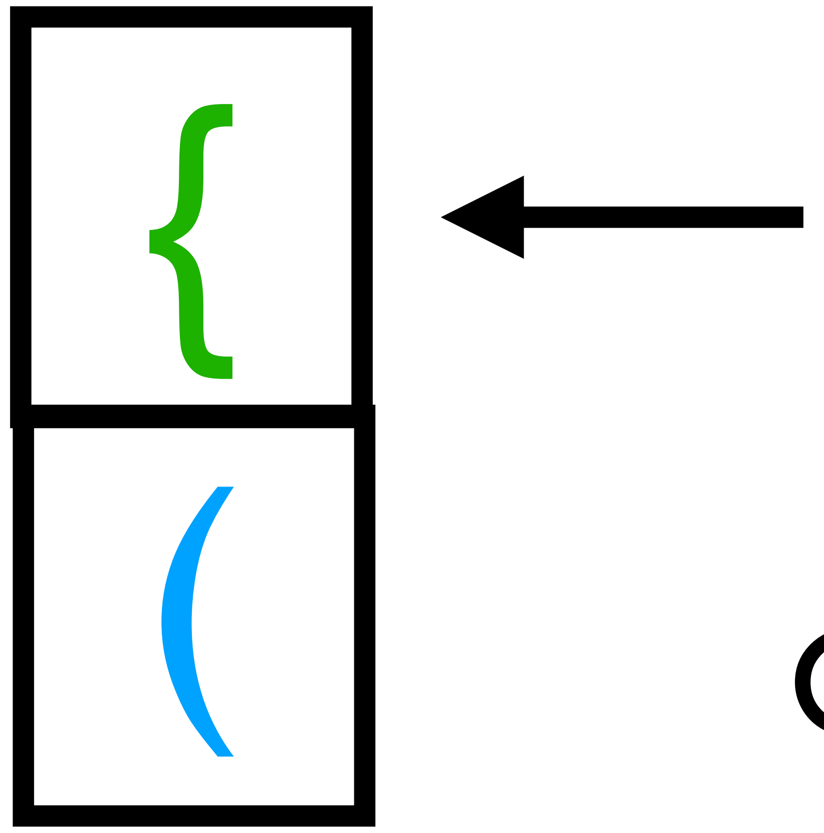
Remember the openers

Goal: Remember the most recent opening symbol we have seen dynamically, as we modify the string by removing matching pairs.

Example:

({})

Read Symbol: }



most recent opener
in new string

Closing symbol: does it match most recent opener?

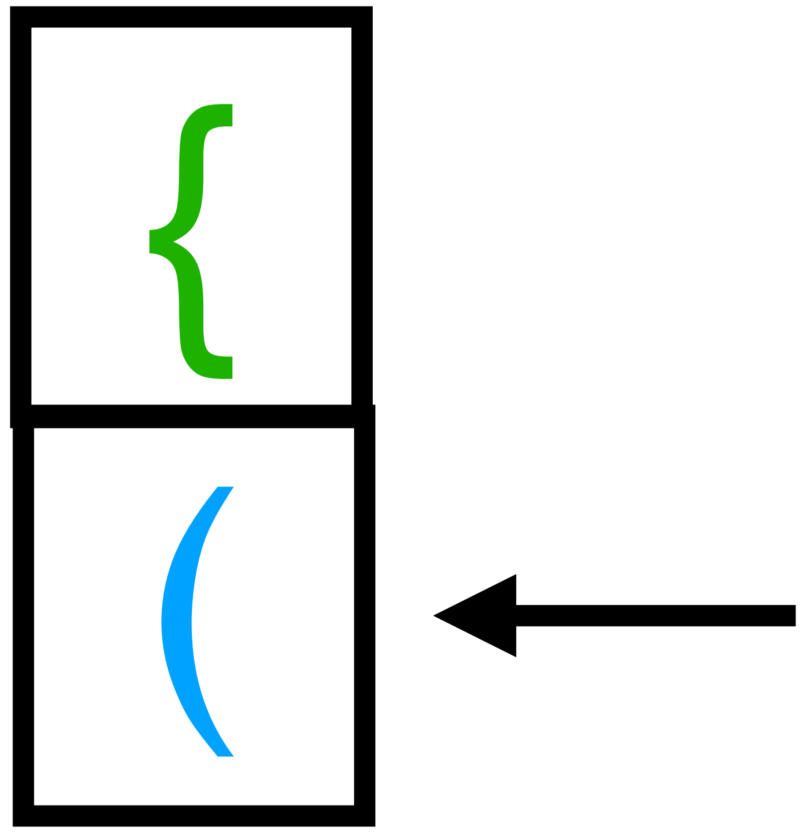
Remember the openers

Goal: Remember the most recent opening symbol we have seen dynamically, as we modify the string by removing matching pairs.

Example:

({}) → ()

Read Symbol: }



most recent opener
in new string

Remember the openers

Goal: Remember the most recent opening symbol we have seen dynamically, as we modify the string by removing matching pairs.

Example:

() → empty string!

Read Symbol:)

(← most recent opener in new string

Summary

We keep at data structure with the most recent opening symbol we have seen at the top.

For each closing symbol we read, we check if it matches the top opening symbol.

- If not then we return invalid.
- If so, then we remove the top opening symbol from the data structure.
  - The new top symbol is the most recent opening symbol read in the new string after removing the matched pair.

Summary

We keep at data structure with the most recent opening symbol we have seen at the top.

For each closing symbol we read, we check if it matches the top opening symbol.

If not then we return invalid.

If so, then we remove the top opening symbol from the data structure.

If after processing the last symbol in the string the data structure is empty we return valid.

Efficiency

This algorithm just makes one pass through the input: the running time is $\Theta(n)$.

(((((((())))))))

On an input like this of size n we have to remember $n/2$ symbols.

The worst-case memory use is $\Theta(n)$ as well.

Abstract Data Type

What operations did we need to perform for this algorithm to work?

We wanted to add items to our collection of values and keep the most recently added item on top.

We wanted to check the value of the top item.

We wanted to remove the top item. Then the second most recently added item became the top item.

This ADT is called a stack!

Stack

Stack ADT

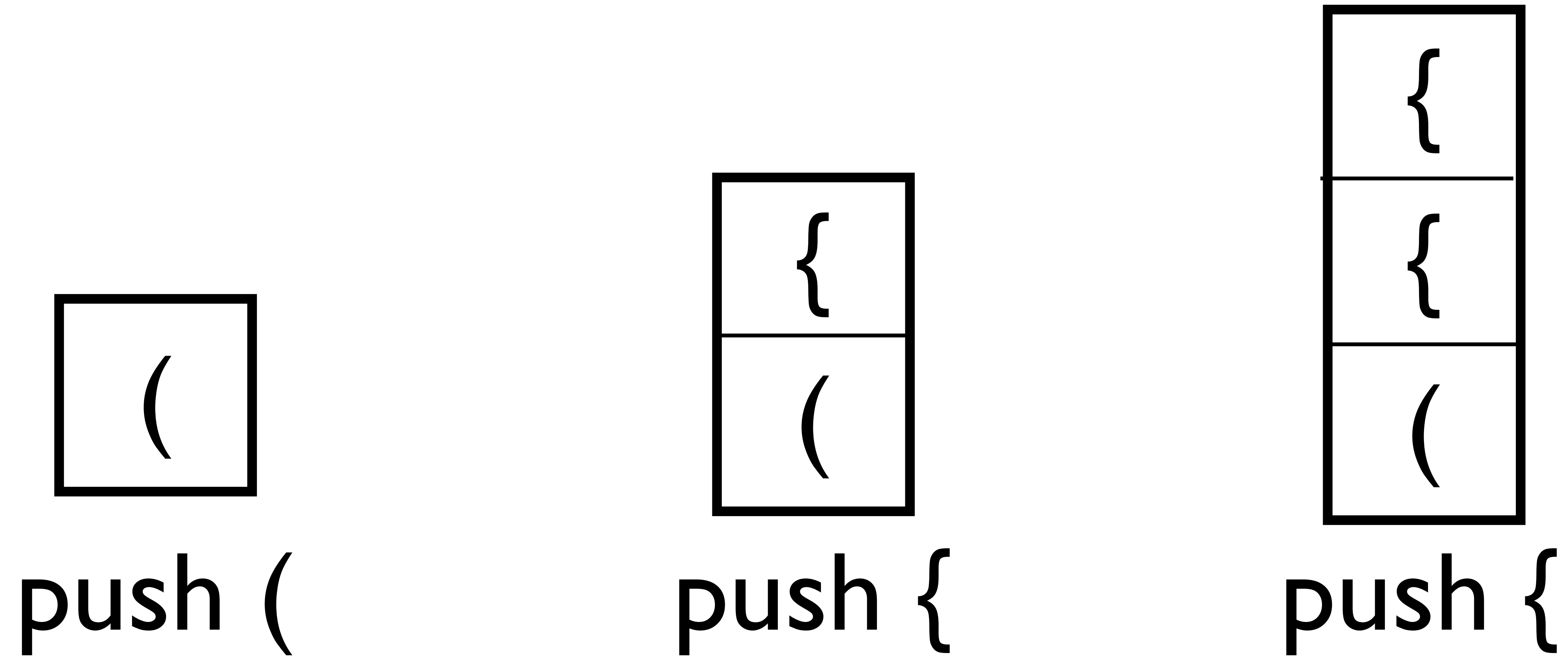
$A \leftarrow \texttt{Stack}()$	Creates an empty stack
$A.\texttt{push}(\texttt{x})$	Add x to A.
$A.\texttt{top}()$	Return the most recently added item in A.
$A.\texttt{pop}()$	Remove the top element from A.
$A.\texttt{size}()$	Return the number of elements in A.

Stack ADT: push

The push operation is how we add items to a stack:

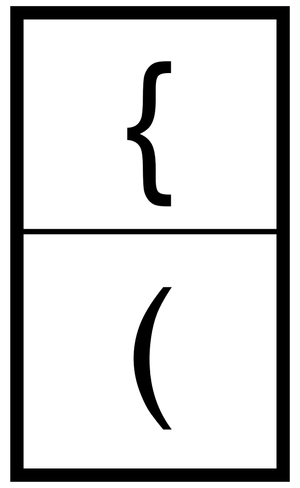
We picture the stack growing "upwards".

Pushed items are added to the top of the stack.

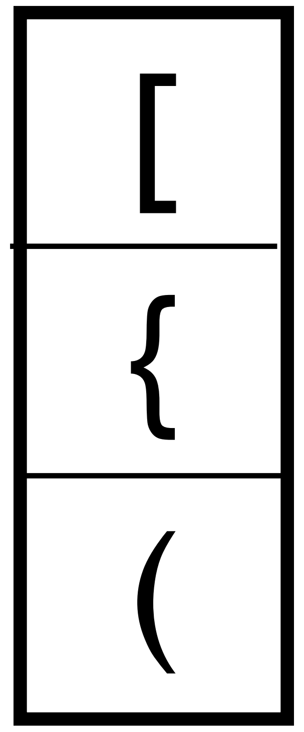


Stack ADT: top

The top operation returns the element at the top of the stack.



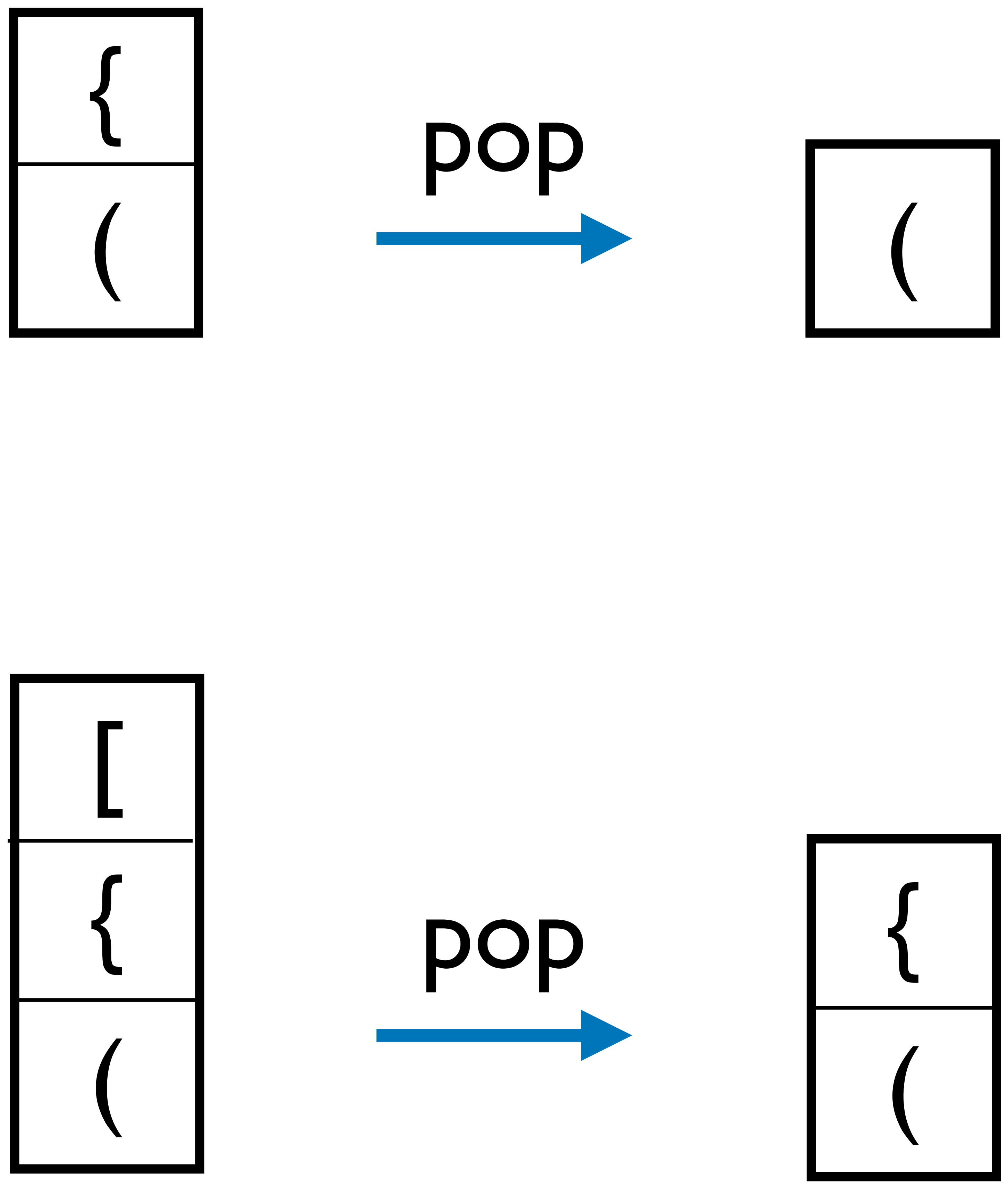
top returns {



top returns [

Stack ADT: pop

The pop operation removes the top element from the stack.



A stack has Last In First Out (LIFO) behavior.

Stack ADT

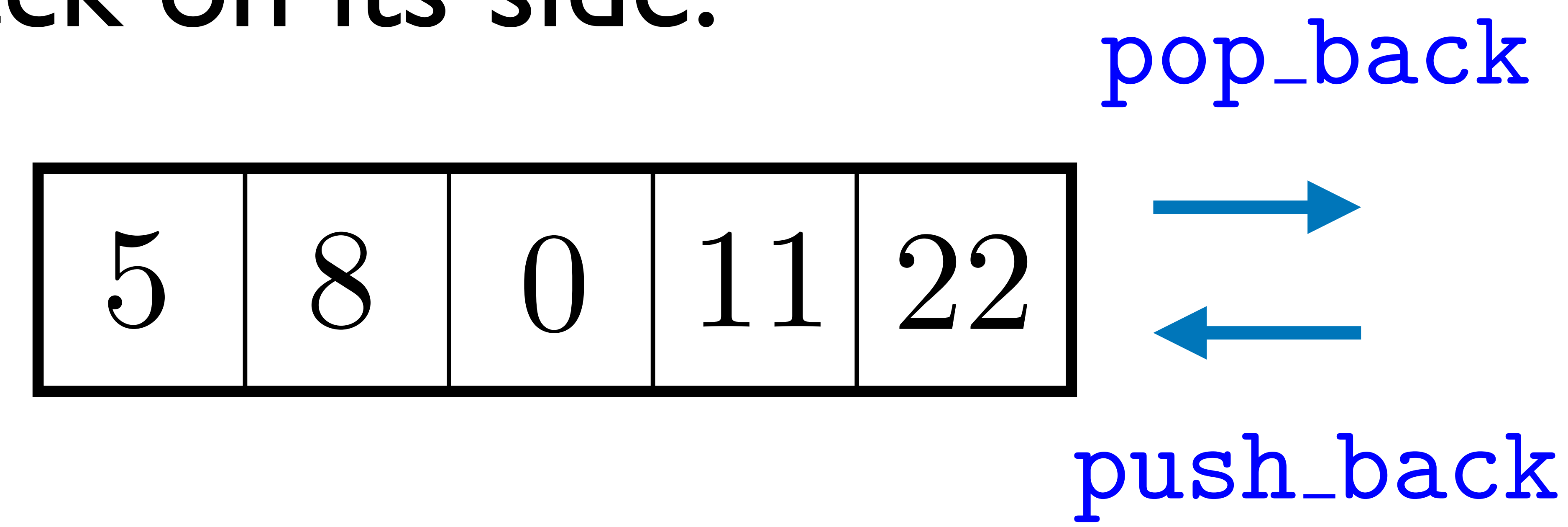
We have used the same names for the operations as in std::stack
image from https://en.cppreference.com/w/cpp/container/stack

Element access	
top	accesses the top element (public member function)
Capacity	
empty	checks whether the underlying container is empty (public member function)
size	returns the number of elements (public member function)
Modifiers	
push	inserts element at the top (public member function)
emplace (C++11)	constructs element in-place at the top (public member function)
pop	removes the top element (public member function)

Stack Implementation

The ADT of a stack is a subset of that of a resizable array.

Turn the stack on its side:



Then push and pop are exactly like push_back and pop_back.

We can implement top by $A.\texttt{get}(\texttt{A.size}() - 1)$.

C++ std::stack

The standard library implements a stack in std::stack.

This is a container adaptor: just a thin wrapper around another container like std::vector.

You can specify which container you would like std::stack to use, the default is a std::deque.

Stack in Practice

The downside of a stack is that we cannot iterate through the elements without popping them off, thereby modifying the stack.

This can make debugging difficult.

In practice, with std::stack we give up operations compared to a std::vector, without gaining any performance benefits.

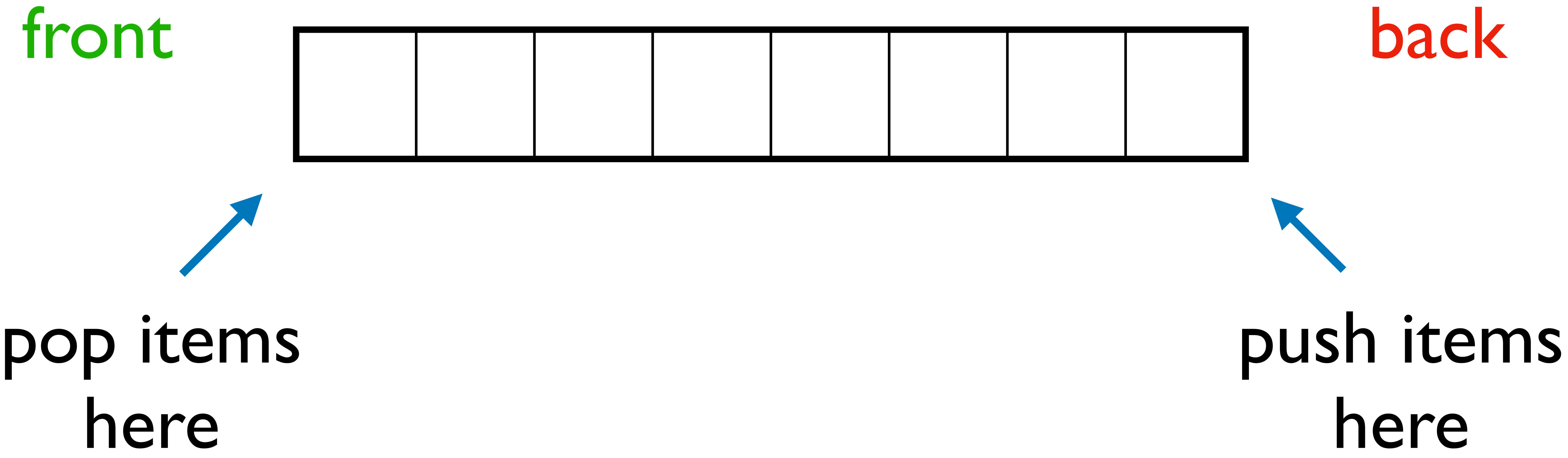
The only advantage is to communicate to readers of your code "I only use the reduced functionality of a stack here".

When I am coding I will usually just use a std::vector instead.

Queue

Queue

The dynamics of a queue is familiar from waiting in line.



We add items to the back of the line, and remove them from the front.

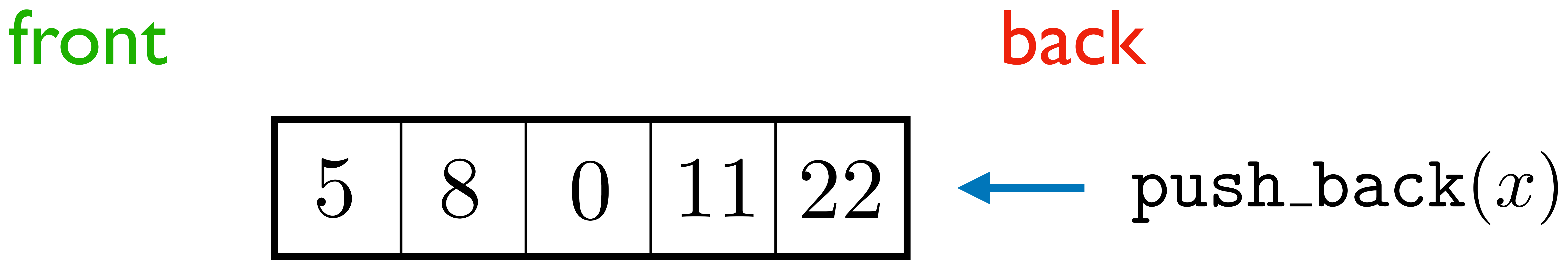
This has First In First Out (FIFO) behavior.

Queue ADT

$A \leftarrow \texttt{Queue}()$	Creates an empty queue
$A.\texttt{push}(\texttt{x})$	Add x to A.
$A.\texttt{front}()$	Return the item oldest item in A .
$A.\texttt{pop}()$	Remove the front element from A .
$A.\texttt{size}()$	Return the number of elements in A .

Queue Implementation

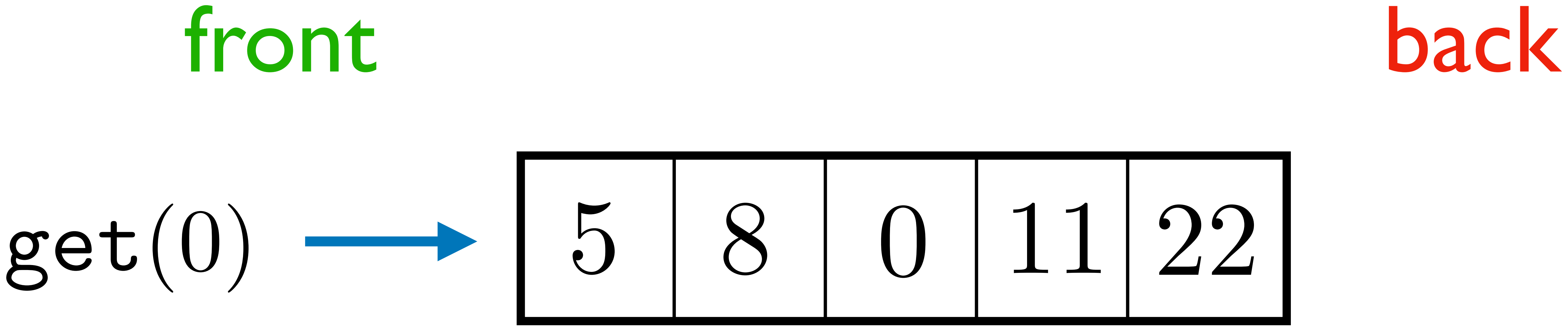
The ADT of a queue is a subset of that of a deque or linked list.



Queue	Deque
push(x)	push_back(x)

Queue Implementation

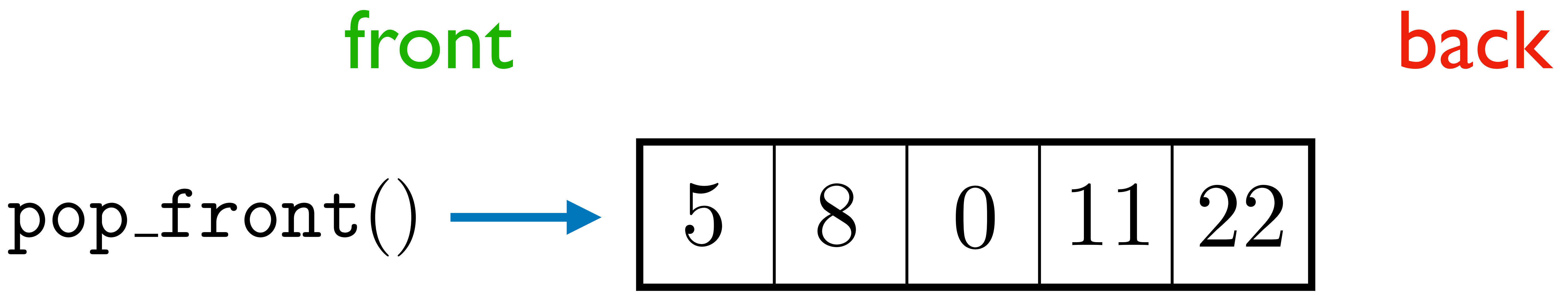
The ADT of a stack is a subset of that of a deque or linked list.



Queue	Deque
front()	get(0)

Queue Implementation

The ADT of a stack is a subset of that of a deque or linked list.



Queue	Deque
pop()	pop_front()

C++ std::queue

The standard library implements a queue in std::queue.

As with std::stack, this is a container adaptor: just a thin wrapper around another container, either std::deque or std::list.

Using std::queue we give up operations compared to a std::deque, without any gain in efficiency.

It can also make our program hard to debug as we cannot see what is in the queue without destroying it.