Module std::collections

1.0.0 [−][src]

[−] Collection types.

Rust's standard collection library provides efficient implementations of the most common general purpose programming data structures. By using the standard implementations, it should be possible for two libraries to communicate without significant data conversion.

To get this out of the way: you should probably just use `Vec` or `HashMap`. These two collections cover most use cases for generic data storage and processing. They are exceptionally good at doing what they do. All the other collections in the standard library have specific use cases where they are the optimal choice, but these cases are borderline *niche* in comparison. Even when `Vec` and `HashMap` are technically suboptimal, they're probably a good enough choice to get started.

Rust's collections can be grouped into four major categories:

- Sequences: `Vec`, `VecDeque`, `LinkedList`
- Maps: `HashMap`, `BTreeMap`
- Sets: `HashSet`, `BTreeSet`
- Misc: `BinaryHeap`

When Should You Use Which Collection?

These are fairly high-level and quick break-downs of when each collection should be considered. Detailed discussions of strengths and weaknesses of individual collections can be found on their own documentation pages.

Use a `Vec` when:

- You want to collect items up to be processed or sent elsewhere later, and don't care about any properties of the actual values being stored.
- You want a sequence of elements in a particular order, and will only be appending to (or near) the end.
- You want a stack.
- You want a resizable array.
- You want a heap-allocated array.

Use a `VecDeque` when:

- You want a `Vec` that supports efficient insertion at both ends of the sequence.
- You want a queue.
- You want a double-ended queue (deque).

Use a `LinkedList` when:

- You want a `Vec` or `VecDeque` of unknown size, and can't tolerate amortization.
- You want to efficiently split and append lists.
- You are *absolutely* certain you *really*, *truly*, want a doubly linked list.

Use a `HashMap` when:

- You want to associate arbitrary keys with an arbitrary value.
- You want a cache.
- You want a map, with no extra functionality.

Use a `BTreeMap` when:

- You want a map sorted by its keys.
- You want to be able to get a range of entries on-demand.
- You're interested in what the smallest or largest key-value pair is.
- You want to find the largest or smallest key that is smaller or larger than something.

Use the `Set` variant of any of these `Map`s when:

- You just want to remember which keys you've seen.
- There is no meaningful value to associate with your keys.
- You just want a set.

Use a `BinaryHeap` when:

- You want to store a bunch of elements, but only ever want to process the "biggest" or "most important" one at any given time.
- You want a priority queue.

Performance

Choosing the right collection for the job requires an understanding of what each collection is good at. Here we briefly summarize the performance of different collections for certain important operations. For further details, see each type's documentation, and note that the names of actual methods may differ from the tables below on certain collections.

Throughout the documentation, we will follow a few conventions. For all operations, the collection's size is denoted by n. If another collection is involved in the operation, it contains m elements. Operations which have an *amortized* cost are suffixed with a `*`. Operations with an *expected* cost are suffixed with a `~`.

All amortized costs are for the potential need to resize when capacity is exhausted. If a resize occurs it will take $O(n)$ time. Our collections never automatically shrink, so removal operations aren't amortized. Over a sufficiently large series of operations, the average cost per operation will deterministically equal the given cost.

Only `HashMap` has expected costs, due to the probabilistic nature of hashing. It is theoretically possible, though very unlikely, for `HashMap` to experience worse performance.

Sequences

	get(i)	insert(i)	remove(i)	append	split_off(i)
`Vec`	O(1)	O(n-i)*	O(n-i)	O(m)*	O(n-i)
`VecDeque`	O(1)	O(min(i, n-i))*	O(min(i, n-i))	O(m)*	O(min(i, n-i))
`LinkedList`	O(min(i, n-i))	O(min(i, n-i))	O(min(i, n-i))	O(1)	O(min(i, n-i))

Note that where ties occur, `Vec` is generally going to be faster than `VecDeque`, and `VecDeque` is generally going to be faster than `LinkedList`.

Maps

For Sets, all operations have the cost of the equivalent Map operation.

	get	insert	remove	predecessor	append
`HashMap`	O(1)~	O(1)~*	O(1)~	N/A	N/A
`BTreeMap`	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n+m)

Correct and Efficient Usage of Collections

Of course, knowing which collection is the right one for the job doesn't instantly permit you to use it correctly. Here are some quick tips for efficient and correct usage of the standard collections in general. If you're interested in how to use a specific collection in particular, consult its documentation for detailed discussion and code examples.

Capacity Management

Many collections provide several constructors and methods that refer to "capacity". These collections are generally built on top of an array. Optimally, this array would be exactly the right size to fit only the elements stored in the collection, but for the collection to do this would be very inefficient. If the backing array was exactly the right size at all times, then every time an element is inserted, the collection would have to grow the array to fit it. Due to the way memory is allocated and managed on most computers, this would almost surely require allocating an entirely new array and copying every single element from the old one into the new one. Hopefully you can see that this wouldn't be very efficient to do on every operation.

Most collections therefore use an *amortized* allocation strategy. They generally let themselves have a fair amount of unoccupied space so that they only have to grow on occasion. When they do grow, they allocate a substantially larger array to move the elements into so that it will take a while for another grow to be required. While this strategy is great in general, it would be even better if the collection *never* had to resize its backing array. Unfortunately, the collection itself doesn't have enough information to do this itself. Therefore, it is up to us programmers to give it hints.

Any `with_capacity` constructor will instruct the collection to allocate enough space for the specified number of elements. Ideally this will be for exactly that many elements, but some implementation details may prevent this. See collection-specific documentation for details. In general, use `with_capacity` when you know exactly how many elements will be inserted, or at least have a reasonable upper-bound on that number.

When anticipating a large influx of elements, the `reserve` family of methods can be used to hint to the collection how much room it should make for the coming items. As with `with_capacity`, the precise behavior of these methods will be specific to the collection of interest.

For optimal performance, collections will generally avoid shrinking themselves. If you believe that a collection will not soon contain any more elements, or just really need the memory, the `shrink_to_fit` method prompts the collection to shrink the backing array to the minimum size capable of holding its elements.

Finally, if ever you're interested in what the actual capacity of the collection is, most collections provide a `capacity` method to query this information on demand. This can be useful for debugging purposes, or for use with the `reserve` methods.

Iterators

Iterators are a powerful and robust mechanism used throughout Rust's standard libraries. Iterators provide a sequence of values in a generic, safe, efficient and convenient way. The contents of an iterator are usually *lazily* evaluated, so that only the values that are actually needed are ever actually produced, and no allocation need be done to temporarily store them. Iterators are primarily consumed using a `for` loop, although many functions also take iterators where a collection or sequence of values is desired.

All of the standard collections provide several iterators for performing bulk manipulation of their contents. The three primary iterators almost every collection should provide are `iter`, `iter_mut`, and `into_iter`. Some of these are not provided on collections where it would be unsound or unreasonable to provide them.

`iter` provides an iterator of immutable references to all the contents of a collection in the most "natural" order. For sequence collections like `Vec`, this means the items will be yielded in increasing order of index starting at 0. For ordered collections like `BTreeMap`, this means that the items will be yielded in sorted order. For unordered collections like `HashMap`, the items will be yielded in whatever order the internal representation made most convenient. This is great for reading through all the contents of the collection.

```
let vec = vec![1, 2, 3, 4];
for x in vec.iter() {
   println!("vec contained {}", x);
}
```

Run

`iter_mut` provides an iterator of *mutable* references in the same order as `iter`. This is great for mutating all the contents of the collection.

```
let mut vec = vec![1, 2, 3, 4];
for x in vec.iter_mut() {
   *x += 1;
}
```

Run

`into_iter` transforms the actual collection into an iterator over its contents by-value. This is great when the collection itself is no longer needed, and the values are needed elsewhere. Using `extend` with `into_iter` is the main way that contents of one collection are moved into another. `extend` automatically calls `into_iter`, and takes any `T: IntoIterator`. Calling `collect` on an iterator itself is also a great way to convert one collection into another. Both of these methods should internally use the capacity management tools discussed in the previous section to do this as efficiently as possible.

```
let mut vec1 = vec![1, 2, 3, 4];
let vec2 = vec![10, 20, 30, 40];
vec1.extend(vec2);
```

Run

```
use std::collections::VecDeque;

let vec = vec![1, 2, 3, 4];
let buf: VecDeque<_> = vec.into_iter().collect();
```

Run

Iterators also provide a series of *adapter* methods for performing common threads to sequences. Among the adapters are functional favorites like `map`, `fold`, `skip` and `take`. Of particular interest to collections is the `rev` adapter, that reverses any iterator that supports this operation. Most collections provide reversible iterators as the way to iterate over them in reverse order.

```
let vec = vec![1, 2, 3, 4];
for x in vec.iter().rev() {
   println!("vec contained {}", x);
}
```

Run

Several other collection methods also return iterators to yield a sequence of results but avoid allocating an entire collection to store the result in. This provides maximum flexibility as `collect` or `extend` can be called to "pipe" the sequence into any collection if desired. Otherwise, the sequence can be looped over with a `for` loop. The iterator can also be discarded after partial use, preventing the computation of the unused items.

Entries

The `entry` API is intended to provide an efficient mechanism for manipulating the contents of a map conditionally on the presence of a key or not. The primary motivating use case for this is to provide efficient accumulator maps. For instance, if one wishes to maintain a count of the number of times each key has been seen, they will have to perform some conditional logic on whether this is the first time the key has been seen or not. Normally, this would require a `find` followed by an `insert`, effectively duplicating the search effort on each insertion.

When a user calls `map.entry(&key)`, the map will search for the key and then yield a variant of the `Entry` enum.

If a `Vacant(entry)` is yielded, then the key *was not* found. In this case the only valid operation is to `insert` a value into the entry. When this is done, the vacant entry is consumed and converted into a mutable reference to the value that was inserted. This allows for further manipulation of the value beyond the lifetime of the search itself. This is useful if complex logic needs to be performed on the value regardless of whether the value was just inserted.

If an `Occupied(entry)` is yielded, then the key *was* found. In this case, the user has several options: they can `get`, `insert` or `remove` the value of the occupied entry. Additionally, they can convert the occupied entry into a mutable reference to its value, providing symmetry to the vacant `insert` case.

Examples

Here are the two primary ways in which `entry` is used. First, a simple example where the logic performed on the values is trivial.

Counting the number of times each character in a string occurs

```
use std::collections::btree_map::BTreeMap;

let mut count = BTreeMap::new();
let message = "she sells sea shells by the sea shore";

for c in message.chars() {
    *count.entry(c).or_insert(0) += 1;
}

assert_eq!(count.get(&'s'), Some(&8));

println!("Number of occurrences of each character");
for (char, count) in &count {
    println!("{}: {}", char, count);
}
```

Run

When the logic to be performed on the value is more complex, we may simply use the `entry` API to ensure that the value is initialized and perform the logic afterwards.

Tracking the inebriation of customers at a bar

```
use std::collections::btree_map::BTreeMap;

// A client of the bar. They have a blood alcohol level.
struct Person { blood_alcohol: f32 }

// All the orders made to the bar, by client ID.
let orders = vec![1, 2, 1, 2, 3, 4, 1, 2, 2, 3, 4, 1, 1, 1];

// Our clients.
let mut blood_alcohol = BTreeMap::new();

for id in orders {
    // If this is the first time we've seen this customer, initialize them
    // with no blood alcohol. Otherwise, just retrieve them.
    let person = blood_alcohol.entry(id).or_insert(Person { blood_alcohol: 0.0 });

    // Reduce their blood alcohol level. It takes time to order and drink a beer!
    person.blood_alcohol *= 0.9;

    // Check if they're sober enough to have another beer.
    if person.blood_alcohol > 0.3 {
        // Too drunk... for now.
        println!("Sorry {}, I have to cut you off", id);
    } else {
        // Have another!
        person.blood_alcohol += 0.1;
    }
}
```

Run

Insert and complex keys

If we have a more complex key, calls to `insert` will not update the value of the key. For example:

```
use std::cmp::Ordering;
use std::collections::BTreeMap;
use std::hash::{Hash, Hasher};

#[derive(Debug)]
struct Foo {
    a: u32,
    b: &'static str,
}

// we will compare `Foo`s by their `a` value only.
impl PartialEq for Foo {
    fn eq(&self, other: &Self) -> bool { self.a == other.a }
}

impl Eq for Foo {}

// we will hash `Foo`s by their `a` value only.
impl Hash for Foo {
    fn hash<H: Hasher>(&self, h: &mut H) { self.a.hash(h); }
}

impl PartialOrd for Foo {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> { self.a.partial_cmp(&other.a) }
}

impl Ord for Foo {
    fn cmp(&self, other: &Self) -> Ordering { self.a.cmp(&other.a) }
}

let mut map = BTreeMap::new();
map.insert(Foo { a: 1, b: "baz" }, 99);

// We already have a Foo with an a of 1, so this will be updating the value.
map.insert(Foo { a: 1, b: "xyz" }, 100);

// The value has been updated...
assert_eq!(map.values().next().unwrap(), &100);

// ...but the key hasn't changed. b is still "baz", not "xyz".
assert_eq!(map.keys().next().unwrap().b, "baz");
```

Run

Modules

binary_heap	A priority queue implemented with a binary heap.
btree_map	A map based on a B-Tree.
btree_set	A set based on a B-Tree.
hash_map	A hash map implemented with quadratic probing and SIMD lookup.
hash_set	A hash set implemented as a `HashMap` where the value is `()`.
linked_list	A doubly-linked list with owned nodes.
vec_deque	A double-ended queue implemented with a growable ring buffer.

Structs

BTreeMap	A map based on a B-Tree.
BTreeSet	A set based on a B-Tree.
BinaryHeap	A priority queue implemented with a binary heap.
HashMap	A hash map implemented with quadratic probing and SIMD lookup.
HashSet	A hash set implemented as a `HashMap` where the value is `()`.
LinkedList	A doubly-linked list with owned nodes.
VecDeque	A double-ended queue implemented with a growable ring buffer.

Enums

TryReserveError	Experimental The error type for `try_reserve` methods.