Table of Contents

Deque

Deque is a double-ended queue implemented as a round-robin queue, supporting O(1) head or tail insertion and querying, just like double-ended queues in other languages(C++ std::deque / Rust VecDeque), here deque also supports random access.

Usage

Create

You can create a deque manually via the new() or construct it using the of().

```
1
2   test {
3     let _dv : @deque.Deque[Int] = @deque.new()
4     let _dv = @deque.of([1, 2, 3, 4, 5])
5
6   }
```

If you want to set the length at creation time to minimize expansion consumption, you can add parameter capacity to the new() function.

```
1
2   test {
3     let _dv : @deque.Deque[Int] = @deque.new(capacity=10)
4
5   }
```

Length & Capacity

A deque is an indefinite-length, auto-expandable datatype. You can use length() Tj T* () to get the number of elen current capacity.

```
1
2   test {
3     let dv = @deque.of([1, 2, 3, 4, 5])
4     assert_eq(dv.length(), 5)
5     assert_eq(dv.capacity(), 5)
6   }
```

Similarly, you can use the is_empty to determine whether the queue is empty.

```
1
2   test {
3     let dv : @deque.Deque[Int] = @deque.new()
4     assert_eq(dv.is_empty(), true)
5   }
```

You can use reserve_capacity to reserve capacity, ensures that it can hold at least the number of elements specified by the capacity argument.

```
1
2    test {
3      let dv = @deque.of([1])
4      dv.reserve_capacity(10)
5      assert_eq(dv.capacity(), 10)
6    }
```

Also, you can use shrink_to_fit to shrink the capacity of the deque.

```
1
2    test {
3      let dv = @deque.new(capacity=10)
4      dv.push_back(1)
5      dv.push_back(2)
6      dv.push_back(3)
7      assert_eq(dv.capacity(), 10)
8      dv.shrink_to_fit()
9      assert_eq(dv.capacity(), 3)
10   }
```

Front & Back & Get

You can use front() and back() to get the head and tail elements of the queue, respectively. Since the queue may be empty, their return values are both Option, or None if the queue is empty.

```
1
2    test {
3      let dv = @deque.of([1, 2, 3, 4, 5])
4      assert_eq(dv.front(), Some(1))
5      assert_eq(dv.back(), Some(5))
6    }
```

You can also use get to access elements of the queue directly, but be careful not to cross the boundaries!

```
1
2    test {
3      let dv = @deque.of([1, 2, 3, 4, 5])
4      assert_eq(dv[0], 1)
5      assert_eq(dv[4], 5)
6    }
```

Push & Set

Since the queue is bi-directional, you can use push_front() and push_back() to add values to the head or tail of the queue, respectively.

```
1
2    test {
3        let dv = @deque.of([1, 2, 3, 4, 5])
4        dv.push_front(6)
5        dv.push_front(7)
6        dv.push_back(8)
7        dv.push_back(9)
8
9    }
```

You can also use Deque::set or operator _[_]=_to set elements of the queue direc
tly, but be careful not to cross the boundaries!

```
1
2    test {
3        let dv = @deque.of([1, 2, 3, 4, 5])
4        dv[0] = 5
5        assert_eq(dv[0], 5)
6    }
```

Pop

You can use pop_front() and pop_back() to pop the element at the head or tai
l of the queue, respectively, and like [Front & Back](#Front & Back & Get), th
eir return values are Option, loaded with the value of the element being popped.

```
1
2    test {
3        let dv = @deque.of([1, 2, 3, 4, 5])
4        let _back = dv.pop_back()
5        assert_eq(dv.back(), Some(4))
6        let _front = dv.pop_front()
7        assert_eq(dv.front(), Some(2))
8        assert_eq(dv.length(), 3)
9    }
```

If you only want to pop an element without getting the return value, you can use
 unsafe_pop_front() with unsafe_pop_back(). These two functions will panic i
f the queue is empty.

```
1
2    test {
3        let dv = @deque.of([1, 2, 3, 4, 5])
4        dv.unsafe_pop_front()
5        assert_eq(dv.front(), Some(2))
6        dv.unsafe_pop_back()
7        assert_eq(dv.back(), Some(4))
8    }
```

Clear

You can use clear to clear a deque. But note that the memory it already occupies
 does not change.

```
1
2    test {
3      let dv = @deque.of([1, 2, 3, 4, 5])
4      dv.clear()
5      assert_eq(dv.is_empty(), true)
6    }
```

Equal

deque supports comparing them directly using equal.

```
1
2    test {
3      let dqa = @deque.of([1, 2, 3, 4, 5])
4      let dqb = @deque.of([1, 2, 3, 4, 5])
5      assert_eq(dqa, dqb)
6    }
```

Iter & Map

deque supports vector-like iter/iteri/map/mapi functions and their inverse forms
.

```
1
2    test {
3      let dv = @deque.of([1, 2, 3, 4, 5])
4      let arr = []
5      dv.each(elem => arr.push(elem))
6      assert_eq(arr, [1, 2, 3, 4, 5])
7      let arr2 = []
8      dv.eachi((i, _elem) => arr2.push(i))
9      assert_eq(arr2, [0, 1, 2, 3, 4])
10     let arr3 = []
11     let _ = dv.map(elem => arr3.push(elem + 1))
12     assert_eq(arr3, [2, 3, 4, 5, 6])
13     let arr4 = []
14     let _ = dv.mapi((i, elem) => arr4.push(elem + i))
15     assert_eq(arr4, [1, 3, 5, 7, 9])
16   }
```

Search & Contains

You can use contains() to find out if a value is in the deque, or search() t
o find its index in the deque.

```
1
2    test {
3      let dv = @deque.of([1, 2, 3, 4, 5])
4      assert_eq(dv.contains(1), true)
5      assert_eq(dv.contains(6), false)
6      assert_eq(dv.search(1), Some(0))
7      assert_eq(dv.search(6), None)
8    }
```