Table of Contents

1 Deque
2 Usage
2.1 Create
2.2 Length & Capacity
2.3 Front & Back & Get
2.4 Push & Set
2.5 Pop
2.6 Clear
2.7 Equal
2.8 Iter & Map
2.9 Search & Contains

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 acc
ess.

Usage

Create

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

```

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

}
```

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

```

test {
  let _dv : @deque Deque[Int] = @deque new(capacity=10)

}
```

Length & Capacity

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

```

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

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

```

test {
  let dv : @deque Deque[Int] = @deque new()
  assert_eq(dv is_empty(), true)
}
```

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

```
test {
  let dv = @deque of([1])
  dv reserve_capacity(10)
  assert_eq(dv capacity(), 10)
}
```

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

```
test {
  let dv = @deque new(capacity=10)
  dv push_back(1)
  dv push_back(2)
  dv push_back(3)
  assert_eq(dv capacity(), 10)
  dv shrink_to_fit()
  assert_eq(dv capacity(), 3)
}
```

Front & Back & Get

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

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

You can also use get to access elements of the queue directly, but be careful no
t to cross the boundaries!

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

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.

```

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

```

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

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.

```

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

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.

```

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

Clear

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

```

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

Equal

deque supports comparing them directly using equal.

```

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

Iter & Map

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

```

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

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.

```

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