

Table of Contents

1	Set Package Documentation
1.1	Creating Sets
1.2	Basic Operations
1.3	Set Operations
1.4	Set Relationships
1.5	Iteration and Conversion
1.6	Modifying Sets
1.7	JSON Serialization
1.8	Working with Different Types
1.9	Performance Examples
1.10	Use Cases
1.11	Performance Characteristics
1.12	Best Practices

Set Package Documentation

This package provides a hash-based set data structure that maintains insertion order. The `Set[K]` type stores unique elements and provides efficient membership testing, insertion, and deletion operations.

Creating Sets

There are several ways to create sets:

```
1
2  test "creating sets" {
3
4      let empty_set : @set.Set[Int] = @set.Set::new()
5      inspect(empty_set.size(), content="0")
6      inspect(empty_set.is_empty(), content="true")
7
8
9      let set_with_capacity : @set.Set[Int] = @set.Set::new(capacity=16)
10     inspect(set_with_capacity.capacity(), content="16")
11
12
13     let from_array = @set.Set::from_array([1, 2, 3, 2, 1])
14     inspect(from_array.size(), content="3")
15
16
17     let from_fixed = @set.Set::of([10, 20, 30])
18     inspect(from_fixed.size(), content="3")
19
20
21     let from_iter = @set.Set::from_iter([1, 2, 3, 4, 5].iter())
22     inspect(from_iter.size(), content="5")
23 }
```

Basic Operations

Add, remove, and check membership:

```

1
2  test "basic operations" {
3      let set = @set.Set::new()
4
5
6      set.add("apple")
7      set.add("banana")
8      set.add("cherry")
9      inspect(set.size(), content="3")
10
11
12     set.add("apple")
13     inspect(set.size(), content="3")
14
15
16     inspect(set.contains("apple"), content="true")
17     inspect(set.contains("orange"), content="false")
18
19
20     set.remove("banana")
21     inspect(set.contains("banana"), content="false")
22     inspect(set.size(), content="2")
23
24
25     let was_added = set.add_and_check("date")
26     inspect(was_added, content="true")
27     let was_added_again = set.add_and_check("date")
28     inspect(was_added_again, content="false")
29     let was_removed = set.remove_and_check("cherry")
30     inspect(was_removed, content="true")
31     let was_removed_again = set.remove_and_check("cherry")
32     inspect(was_removed_again, content="false")
33 }

```

Set Operations

Perform mathematical set operations:

```

1
2  test "set operations" {
3      let set1 = @set.Set::from_array([1, 2, 3, 4])
4      let set2 = @set.Set::from_array([3, 4, 5, 6])
5
6
7      let union_set = set1.union(set2)
8      let union_array = union_set.to_array()
9      inspect(union_array.length(), content="6")
10
11
12     let union_alt = set1 | set2
13     inspect(union_alt.size(), content="6")
14
15
16     let intersection_set = set1.intersection(set2)
17     let intersection_array = intersection_set.to_array()
18     inspect(intersection_array.length(), content="2")
19
20
21     let intersection_alt = set1 & set2
22     inspect(intersection_alt.size(), content="2")
23
24
25     let difference_set = set1.difference(set2)
26     let difference_array = difference_set.to_array()
27     inspect(difference_array.length(), content="2")
28
29
30     let difference_alt = set1 - set2
31     inspect(difference_alt.size(), content="2")
32
33
34     let sym_diff_set = set1.symmetric_difference(set2)
35     let sym_diff_array = sym_diff_set.to_array()
36     inspect(sym_diff_array.length(), content="4")
37
38
39     let sym_diff_alt = set1 ^ set2
40     inspect(sym_diff_alt.size(), content="4")
41 }

```

Set Relationships

Test relationships between sets:

```

1
2  test "set relationships" {
3      let small_set = @set.Set::from_array([1, 2])
4      let large_set = @set.Set::from_array([1, 2, 3, 4])
5      let disjoint_set = @set.Set::from_array([5, 6, 7])
6
7
8      inspect(small_set.is_subset(large_set), content="true")
9      inspect(large_set.is_subset(small_set), content="false")
10
11
12     inspect(large_set.is_superset(small_set), content="true")
13     inspect(small_set.is_superset(large_set), content="false")
14
15
16     inspect(small_set.is_disjoint(disjoint_set), content="true")
17     inspect(small_set.is_disjoint(large_set), content="false")
18
19
20     let set1 = @set.Set::from_array([1, 2, 3])
21     let set2 = @set.Set::from_array([3, 2, 1])
22     inspect(set1 == set2, content="true")
23 }

```

Iteration and Conversion

Iterate over sets and convert to other types:

```

1
2  test "iteration and conversion" {
3      let set = @set.Set::from_array(["first", "second", "third"])
4
5
6      let array = set.to_array()
7      inspect(array.length(), content="3")
8
9
10     let mut count = 0
11     set.each(fn(_element) { count = count + 1 })
12     inspect(count, content="3")
13
14
15     let mut indices_sum = 0
16     set.eachi(fn(i, _element) { indices_sum = indices_sum + i })
17     inspect(indices_sum, content="3")
18
19
20     let elements = set.iter().collect()
21     inspect(elements.length(), content="3")
22
23
24     let copied_set = set.copy()
25     inspect(copied_set.size(), content="3")
26     inspect(copied_set == set, content="true")
27 }

```

Modifying Sets

Clear and modify existing sets:

```

1
2  test "modifying sets" {
3      let set = @set.Set::from_array([10, 20, 30, 40, 50])
4      inspect(set.size(), content="5")
5
6
7      set.clear()
8      inspect(set.size(), content="0")
9      inspect(set.is_empty(), content="true")
10
11
12     set.add(100)
13     set.add(200)
14     inspect(set.size(), content="2")
15     inspect(set.contains(100), content="true")
16 }

```

JSON Serialization

Sets can be serialized to JSON as arrays:

```

1
2  test "json serialization" {
3      let set = @set.Set::from_array([1, 2, 3])
4      let json = set.to_json()
5
6
7      inspect(json, content="Array([Number(1), Number(2), Number(3)])")
8
9
10     let string_set = @set.Set::from_array(["a", "b", "c"])
11     let string_json = string_set.to_json()
12     inspect(
13         string_json,
14         content="Array([String(\"a\"), String(\"b\"), String(\"c\")])",
15     )
16 }

```

Working with Different Types

Sets work with any type that implements Hash and Eq:

```

1
2  test "different types" {
3
4      let int_set = @set.Set::from_array([1, 2, 3, 4, 5])
5      inspect(int_set.contains(3), content="true")
6
7
8      let string_set = @set.Set::from_array(["hello", "world", "moonbit"])
9      inspect(string_set.contains("world"), content="true")
10
11
12
13     let char_codes = @set.Set::from_array([97, 98, 99])
14     inspect(char_codes.contains(98), content="true")
15
16
17     let bool_codes = @set.Set::from_array([1, 0, 1])
18     inspect(bool_codes.size(), content="2")
19 }

```

Performance Examples

Demonstrate efficient operations:

```

1
2  test "performance examples" {
3
4      let large_set = @set.Set::new(capacity=1000)
5
6
7      for i in 0..<100 {
8          large_set.add(i)
9      }
10     inspect(large_set.size(), content="100")
11
12
13     inspect(large_set.contains(50), content="true")
14     inspect(large_set.contains(150), content="false")
15
16
17     let another_set = @set.Set::new()
18     for i in 50..<150 {
19         another_set.add(i)
20     }
21     let intersection = large_set.intersection(another_set)
22     inspect(intersection.size(), content="50")
23 }

```

Use Cases

Sets are particularly useful for:

- **Removing duplicates:** Convert arrays to sets and back to remove duplicates
- **Membership testing:** Fast $O(1)$ average-case lookups
- **Mathematical operations:** Union, intersection, difference operations
- **Unique collections:** Maintaining collections of unique items
- **Algorithm implementation:** Graph algorithms, caching, etc.

Performance Characteristics

- **Insertion:** $O(1)$ average case, $O(n)$ worst case
- **Removal:** $O(1)$ average case, $O(n)$ worst case
- **Lookup:** $O(1)$ average case, $O(n)$ worst case
- **Space complexity:** $O(n)$ where n is the number of elements
- **Iteration order:** Maintains insertion order (linked hash set)

Best Practices

- **Pre-size when possible:** Use `@set.Set::new(capacity=n)` if you know the approximate size
- **Use appropriate types:** Ensure your key type has good Hash and Eq implementations
- **Prefer set operations:** Use built-in union, intersection, etc. instead of manual loops
- **Check return values:** Use `add_and_check` and `remove_and_check` when you need to know if the operation succeeded
- **Consider memory usage:** Sets have overhead compared to arrays for small collections