

Table of Contents

1	Sorted Map
1.1	Overview
1.2	Performance
1.3	Usage
1.3.1	Create
1.3.2	Container Operations
1.3.3	Data Extraction
1.3.4	Range Operations
1.3.5	Iterators
1.3.6	Equality
1.3.7	Error Handling Best Practices
1.4	Implementation Notes
1.5	Comparison with Other Collections

Sorted Map

A mutable map backed by an AVL tree that maintains keys in sorted order.

Overview

SortedMap is an ordered map implementation that keeps entries sorted by keys. It provides efficient lookup, insertion, and deletion operations, with stable traversal order based on key comparison.

Performance

- **add/set**: $O(\log n)$
- **remove**: $O(\log n)$
- **get/contains**: $O(\log n)$
- **iterate**: $O(n)$
- **range**: $O(\log n + k)$ where k is number of elements in range
- **space complexity**: $O(n)$

Usage

Create

You can create an empty SortedMap or a SortedMap from other containers.

```
1
2  test {
3      let _map1 : @sorted_map.SortedMap[Int, String] = @sorted_map.new()
4      let _map2 = @sorted_map.from_array([(1, "one"), (2, "two"), (3, "three")])
5
6  }
```

Container Operations

Add a key-value pair to the SortedMap in place.

```
1
2  test {
3      let map = @sorted_map.from_array([(1, "one"), (2, "two")])
4      map.set(3, "three")
5      assert_eq(map.size(), 3)
6  }
```

You can also use the convenient subscript syntax to add or update values:

```

1
2  test {
3      let map = @sorted_map.new()
4      map[1] = "one"
5      map[2] = "two"
6      assert_eq(map.size(), 2)
7  }

```

Remove a key-value pair from the SortedMap in place.

```

1
2  test {
3      let map = @sorted_map.from_array([(1, "one"), (2, "two"), (3, "three")])
4      map.remove(2)
5      assert_eq(map.size(), 2)
6      assert_eq(map.contains(2), false)
7  }

```

Get a value by its key. The return type is Option[V].

```

1
2  test {
3      let map = @sorted_map.from_array([(1, "one"), (2, "two"), (3, "three")])
4      assert_eq(map.get(2), Some("two"))
5      assert_eq(map.get(4), None)
6  }

```

Safe access with error handling:

```

1
2  test {
3      let map = @sorted_map.from_array([(1, "one"), (2, "two")])
4      let key = 3
5      inspect(map.get(key), content="None")
6  }

```

Check if a key exists in the map.

```

1
2  test {
3      let map = @sorted_map.from_array([(1, "one"), (2, "two"), (3, "three")])
4      assert_eq(map.contains(2), true)
5      assert_eq(map.contains(4), false)
6  }

```

Iterate over all key-value pairs in the map in sorted key order.

```

1
2  test {
3    let map = @sorted_map.from_array([(3, "three"), (1, "one"), (2, "two")])
4    let keys = []
5    let values = []
6    map.each((k, v) => {
7      keys.push(k)
8      values.push(v)
9    })
10   assert_eq(keys, [1, 2, 3])
11   assert_eq(values, ["one", "two", "three"])
12 }

```

Iterate with index:

```

1
2  test {
3    let map = @sorted_map.from_array([(3, "three"), (1, "one"), (2, "two")])
4    let result = []
5    map.eachi((i, k, v) => result.push((i, k, v)))
6    assert_eq(result, [(0, 1, "one"), (1, 2, "two"), (2, 3, "three")])
7  }

```

Get the size of the map.

```

1
2  test {
3    let map = @sorted_map.from_array([(1, "one"), (2, "two"), (3, "three")])
4    assert_eq(map.size(), 3)
5  }

```

Check if the map is empty.

```

1
2  test {
3    let map : @sorted_map.SortedMap[Int, String] = @sorted_map.new()
4    assert_eq(map.is_empty(), true)
5  }

```

Clear the map.

```

1
2  test {
3    let map = @sorted_map.from_array([(1, "one"), (2, "two"), (3, "three")])
4    map.clear()
5    assert_eq(map.is_empty(), true)
6  }

```

Data Extraction

Get all keys or values from the map.

```

1
2  test {
3    let map = @sorted_map.from_array([(3, "three"), (1, "one"), (2, "two")])
4    assert_eq(map.keys_as_iter().collect(), [1, 2, 3])
5    assert_eq(map.values_as_iter().collect(), ["one", "two", "three"])
6  }

```

Convert the map to an array of key-value pairs.

```

1
2  test {
3    let map = @sorted_map.from_array([(3, "three"), (1, "one"), (2, "two")])
4    assert_eq(map.to_array(), [(1, "one"), (2, "two"), (3, "three")])
5  }

```

Range Operations

Get a subset of the map within a specified range of keys. The range is inclusive for both bounds [low, high].

```

1
2  test {
3    let map = @sorted_map.from_array([
4      (1, "one"),
5      (2, "two"),
6      (3, "three"),
7      (4, "four"),
8      (5, "five"),
9    ])
10   let range_items = []
11   map.range(2, 4).each((k, v) => range_items.push((k, v)))
12   assert_eq(range_items, [(2, "two"), (3, "three"), (4, "four")])
13 }

```

Edge cases for range operations:

- If low > high, returns an empty result
- If low or high are outside the map bounds, returns only pairs within valid bound
- The returned iterator preserves the sorted order of keys

```

1
2
3  test {
4    let map = @sorted_map.from_array([(1, "one"), (2, "two"), (3, "three")])
5    let range_items = []
6    map.range(0, 10).each((k, v) => range_items.push((k, v)))
7    assert_eq(range_items, [(1, "one"), (2, "two"), (3, "three")])
8
9
10   let empty_range : Array[(Int, String)] = []
11   map.range(10, 5).each((k, v) => empty_range.push((k, v)))
12   assert_eq(empty_range, [])
13 }

```

Iterators

The SortedMap supports several iterator patterns. Create a map from an iterator:

```
1
2  test {
3    let pairs = [(1, "one"), (2, "two"), (3, "three")].iter()
4    let map = @sorted_map.from_iter(pairs)
5    assert_eq(map.size(), 3)
6  }
```

Use the iter method to get an iterator over key-value pairs:

```
1
2  test {
3    let map = @sorted_map.from_array([(3, "three"), (1, "one"), (2, "two")])
4    let pairs = map.iter().to_array()
5    assert_eq(pairs, [(1, "one"), (2, "two"), (3, "three")])
6  }
```

Use the iter2 method for a more convenient key-value iteration:

```
1
2  test {
3    let map = @sorted_map.from_array([(3, "three"), (1, "one"), (2, "two")])
4    let transformed = []
5    map.iter2().each((k, v) => transformed.push(k.to_string() + ": " + v))
6    assert_eq(transformed, ["1: one", "2: two", "3: three"])
7  }
```

Equality

Maps with the same key-value pairs are considered equal, regardless of the order in which elements were added.

```
1
2  test {
3    let map1 = @sorted_map.from_array([(1, "one"), (2, "two")])
4    let map2 = @sorted_map.from_array([(2, "two"), (1, "one")])
5    assert_eq(map1 == map2, true)
6  }
```

Error Handling Best Practices

When working with keys that might not exist, prefer using pattern matching for safety:

```

1
2  fn get_score(scores : @sorted_map.SortedMap[Int, Int], student_id : Int)
3      match scores.get(student_id) {
4          Some(score) => score
5          None =>
6
7
8
9
10
11         0
12     }
13 }
14
15
16 test "safe_key_access" {
17
18     let scores = @sorted_map.from_array([(1001, 85), (1002, 92), (1003, 78)
19
20
21     assert_eq(get_score(scores, 1001), 85)
22
23
24     assert_eq(get_score(scores, 9999), 0)
25 }

```

Implementation Notes

The SortedMap is implemented as an AVL tree, a self-balancing binary search tree. After insertions and deletions, the tree automatically rebalances to maintain $O(\log n)$ search, insertion, and deletion times.

Key properties of the AVL tree implementation:

- Each node stores a balance factor (height difference between left and right subtrees)
- The balance factor is maintained between -1 and 1 for all nodes
- Rebalancing is done through tree rotations (single and double rotations)

Comparison with Other Collections

- **@hashmap.T**: Provides $O(1)$ average case lookups but doesn't maintain order; use when order doesn't matter
- **@indexmap.T**: Maintains insertion order but not sorted order; use when insertion order matters
- **@sorted_map.SortedMap**: Maintains keys in sorted order; use when you need keys to be sorted

Choose SortedMap when you need:

- Key-value pairs sorted by key
- Efficient range queries
- Ordered traversal guarantees