

```

1 // SPDX-License-Identifier: MIT
2 // OpenZeppelin Contracts (last updated v4.8.0) (utils/structs/EnumerableSet.sol)
3 // This file was procedurally generated from
  scripts/generate/templates/EnumerableSet.js.
4
5 pragma solidity ^0.8.0;
6
7 /**
8  * @dev Library for managing
9  * https://en.wikipedia.org/wiki/Set_(abstract_data_type) [sets] of primitive
10  * types.
11  *
12  * Sets have the following properties:
13  *
14  * - Elements are added, removed, and checked for existence in constant time
15  *   (O(1)).
16  * - Elements are enumerated in O(n). No guarantees are made on the ordering.
17  *
18  * ``
19  * contract Example {
20  *     // Add the library methods
21  *     using EnumerableSet for EnumerableSet.AddressSet;
22  *
23  *     // Declare a set state variable
24  *     EnumerableSet.AddressSet private mySet;
25  * }
26  * ``
27  *
28  * As of v3.3.0, sets of type `bytes32` (`Bytes32Set`), `address` (`AddressSet`)
29  * and `uint256` (`UintSet`) are supported.
30  *
31  * [WARNING]
32  * ====
33  * Trying to delete such a structure from storage will likely result in data
  corruption, rendering the structure
34  * unusable.
35  * See https://github.com/ethereum/solidity/pull/11843 [ethereum/solidity#11843] for
  more info.
36  *
37  * In order to clean an EnumerableSet, you can either remove all elements one by one
  or create a fresh instance using an
38  * array of EnumerableSet.
39  * ====
40  */
41 library EnumerableSet {
42     // To implement this library for multiple types with as little code
43     // repetition as possible, we write it in terms of a generic Set type with
44     // bytes32 values.
45     // The Set implementation uses private functions, and user-facing
46     // implementations (such as AddressSet) are just wrappers around the
47     // underlying Set.
48     // This means that we can only create new EnumerableSets for types that fit
49     // in bytes32.
50
51     struct Set {
52         // Storage of set values
53         bytes32[] _values;
54         // Position of the value in the `values` array, plus 1 because index 0
55         // means a value is not in the set.
56         mapping(bytes32 => uint256) _indexes;
57     }
58
59     /**
60      * @dev Add a value to a set. O(1).
61      *
62      * Returns true if the value was added to the set, that is if it was not
63      * already present.
64      */
65     function _add(Set storage set, bytes32 value) private returns (bool) {
66         if (!_contains(set, value)) {
67             set._values.push(value);
68             // The value is stored at length-1, but we add 1 to all indexes
69             // and use 0 as a sentinel value

```

```

70         set._indexes[value] := set._values.length;
71         return true;
72     } else {
73         return false;
74     }
75 }
76
77 /**
78  * @dev Removes a value from a set. O(1).
79  *
80  * Returns true if the value was removed from the set, that is if it was
81  * present.
82  */
83 function _remove(Set storage set, bytes32 value) private returns (bool) {
84     // We read and store the value's index to prevent multiple reads from the
85     // same storage slot
86     uint256 valueIndex = set._indexes[value];
87     if (valueIndex != 0) {
88         // Equivalent to contains(set, value)
89         // To delete an element from the _values array in O(1), we swap the
90         // element to delete with the last one in
91         // the array, and then remove the last element (sometimes called as 'swap
92         // and pop').
93         // This modifies the order of the array, as noted in {at}.
94         uint256 toDeleteIndex = valueIndex - 1;
95         uint256 lastIndex = set._values.length - 1;
96         if (lastIndex != toDeleteIndex) {
97             bytes32 lastValue = set._values[lastIndex];
98             // Move the last value to the index where the value to delete is
99             set._values[toDeleteIndex] = lastValue;
100             // Update the index for the moved value
101             set._indexes[lastValue] = valueIndex; // Replace lastValue's index to
102             // valueIndex
103         }
104         // Delete the slot where the moved value was stored
105         set._values.pop();
106         // Delete the index for the deleted slot
107         delete set._indexes[value];
108     }
109     return true;
110 } else {
111     return false;
112 }
113 }
114
115 /**
116  * @dev Returns true if the value is in the set. O(1).
117  *
118  */
119 function _contains(Set storage set, bytes32 value) private view returns (bool) {
120     return set._indexes[value] != 0;
121 }
122
123 /**
124  * @dev Returns the number of values on the set. O(1).
125  *
126  */
127 function _length(Set storage set) private view returns (uint256) {
128     return set._values.length;
129 }
130
131 /**
132  * @dev Returns the value stored at position `index` in the set. O(1).
133  *
134  * Note that there are no guarantees on the ordering of values inside the
135  * array, and it may change when more values are added or removed.
136  *
137  * Requirements:
138  *

```

```

139     * - `index` must be strictly less than {length}.
140     */
141     function _at(Set storage set, uint256 index) private view returns (bytes32) {
142         return set._values[index];
143     }
144
145     /**
146     * @dev Return the entire set in an array
147     *
148     * WARNING: This operation will copy the entire storage to memory, which can be
149     * quite expensive. This is designed
150     * to mostly be used by view accessors that are queried without any gas fees.
151     * Developers should keep in mind that
152     * this function has an unbounded cost, and using it as part of a state-changing
153     * function may render the function
154     * uncallable if the set grows to a point where copying to memory consumes too
155     * much gas to fit in a block.
156     */
157     function _values(Set storage set) private view returns (bytes32[] memory) {
158         return set._values;
159     }
160
161     // Bytes32Set
162
163     struct Bytes32Set {
164         Set _inner;
165     }
166
167     /**
168     * @dev Add a value to a set. O(1).
169     *
170     * Returns true if the value was added to the set, that is if it was not
171     * already present.
172     */
173     function add(Bytes32Set storage set, bytes32 value) internal returns (bool) {
174         return _add(set._inner, value);
175     }
176
177     /**
178     * @dev Removes a value from a set. O(1).
179     *
180     * Returns true if the value was removed from the set, that is if it was
181     * present.
182     */
183     function remove(Bytes32Set storage set, bytes32 value) internal returns (bool) {
184         return _remove(set._inner, value);
185     }
186
187     /**
188     * @dev Returns true if the value is in the set. O(1).
189     */
190     function contains(Bytes32Set storage set, bytes32 value) internal view returns (
191         bool) {
192         return _contains(set._inner, value);
193     }
194
195     /**
196     * @dev Returns the number of values in the set. O(1).
197     */
198     function length(Bytes32Set storage set) internal view returns (uint256) {
199         return _length(set._inner);
200     }
201
202     /**
203     * @dev Returns the value stored at position `index` in the set. O(1).
204     *
205     * Note that there are no guarantees on the ordering of values inside the
206     * array, and it may change when more values are added or removed.
207     *
208     * Requirements:
209     * - `index` must be strictly less than {length}.
210     */

```

```

207     function at(Bytes32Set storage set, uint256 index) internal view returns (bytes32)
208     {
209         return _at(set._inner, index);
210     }
211     /**
212     * @dev Return the entire set in an array
213     *
214     * WARNING: This operation will copy the entire storage to memory, which can be
215     * quite expensive. This is designed
216     * to mostly be used by view accessors that are queried without any gas fees.
217     * Developers should keep in mind that
218     * this function has an unbounded cost, and using it as part of a state-changing
219     * function may render the function
220     * uncalled if the set grows to a point where copying to memory consumes too
221     * much gas to fit in a block.
222     */
223     function values(Bytes32Set storage set) internal view returns (bytes32[] memory) {
224         bytes32[] memory store = _values(set._inner);
225         bytes32[] memory result;
226         /// @solidity memory-safe-assembly
227         assembly {
228             result := store
229         }
230         return result;
231     }
232     // AddressSet
233     struct AddressSet {
234         Set _inner;
235     }
236     /**
237     * @dev Add a value to a set. O(1).
238     *
239     * Returns true if the value was added to the set, that is if it was not
240     * already present.
241     */
242     function add(AddressSet storage set, address value) internal returns (bool) {
243         return _add(set._inner, bytes32(uint256(uint160(value))));
244     }
245     /**
246     * @dev Removes a value from a set. O(1).
247     *
248     * Returns true if the value was removed from the set, that is if it was
249     * present.
250     */
251     function remove(AddressSet storage set, address value) internal returns (bool) {
252         return _remove(set._inner, bytes32(uint256(uint160(value))));
253     }
254     /**
255     * @dev Returns true if the value is in the set. O(1).
256     */
257     function contains(AddressSet storage set, address value) internal view returns (
258         bool) {
259         return _contains(set._inner, bytes32(uint256(uint160(value))));
260     }
261     /**
262     * @dev Returns the number of values in the set. O(1).
263     */
264     function length(AddressSet storage set) internal view returns (uint256) {
265         return _length(set._inner);
266     }
267     /**
268     * @dev Returns the value stored at position `index` in the set. O(1).
269     */

```

```

274     * Note that there are no guarantees on the ordering of values inside the
275     * array, and it may change when more values are added or removed.
276     *
277     * Requirements:
278     *
279     * - `index` must be strictly less than {length}.
280     */
281     function at(AddressSet storage set, uint256 index) internal view returns (address)
282     {
283         return address(uint160(uint256(_at(set._inner, index))));
284     }
285     /**
286     * @dev Return the entire set in an array
287     *
288     * WARNING: This operation will copy the entire storage to memory, which can be
289     * quite expensive. This is designed
290     * to mostly be used by view accessors that are queried without any gas fees.
291     * Developers should keep in mind that
292     * this function has an unbounded cost, and using it as part of a state-changing
293     * function may render the function
294     * uncallable if the set grows to a point where copying to memory consumes too
295     * much gas to fit in a block.
296     */
297     function values(AddressSet storage set) internal view returns (address[] memory) {
298         bytes32[] memory store = _values(set._inner);
299         address[] memory result;
300
301         /// @solidity memory-safe-assembly
302         assembly {
303             result := store
304         }
305
306         return result;
307     }
308     /** UintSet
309     *
310     * @dev Add a value to a set. O(1).
311     *
312     * Returns true if the value was added to the set, that is if it was not
313     * already present.
314     */
315     function add(UintSet storage set, uint256 value) internal returns (bool) {
316         return _add(set._inner, bytes32(value));
317     }
318     /**
319     * @dev Removes a value from a set. O(1).
320     *
321     * Returns true if the value was removed from the set, that is if it was
322     * present.
323     */
324     function remove(UintSet storage set, uint256 value) internal returns (bool) {
325         return _remove(set._inner, bytes32(value));
326     }
327     /**
328     * @dev Returns true if the value is in the set. O(1).
329     */
330     function contains(UintSet storage set, uint256 value) internal view returns (bool)
331     {
332         return _contains(set._inner, bytes32(value));
333     }
334     /**
335     * @dev Returns the number of values in the set. O(1).
336     */
337
338     /**
339     *
340     */

```

```

341     ....function length(UintSet storage set) internal view returns (uint256) {
342     ....    return _length(set._inner);
343     ....}
344
345     ..../**
346     .... * @dev Returns the value stored at position `index` in the set. O(1).
347     .... *
348     .... * Note that there are no guarantees on the ordering of values inside the
349     .... * array, and it may change when more values are added or removed.
350     .... *
351     .... * Requirements:
352     .... *
353     .... * - `index` must be strictly less than {length}.
354     .... */
355     ....function at(UintSet storage set, uint256 index) internal view returns (uint256) {
356     ....    return uint256(_at(set._inner, index));
357     ....}
358
359     ..../**
360     .... * @dev Return the entire set in an array
361     .... *
362     .... * WARNING: This operation will copy the entire storage to memory, which can be
363     .... * quite expensive. This is designed
364     .... * to mostly be used by view accessors that are queried without any gas fees.
365     .... * Developers should keep in mind that
366     .... * this function has an unbounded cost, and using it as part of a state-changing
367     .... * function may render the function
368     .... * uncallable if the set grows to a point where copying to memory consumes too
369     .... * much gas to fit in a block.
370     .... */
371     ....function values(UintSet storage set) internal view returns (uint256[] memory) {
372     ....    bytes32[] memory store = _values(set._inner);
373     ....    uint256[] memory result;
374     ....    // @solidity memory-safe-assembly
375     ....    assembly {
376     ....        result := store
377     ....    }
378     ....    return result;
379     ....}

```