```solidity
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.8.0) (utils/structs/EnumerableSet.sol)
// This file was procedurally generated from scripts/generate/templates/EnumerableSet.js.

pragma solidity ^0.8.0;

/**
 * @dev Library for managing
 * https://en.wikipedia.org/wiki/Set_(abstract_data_type)[sets] of primitive
 * types.
 *
 * Sets have the following properties:
 *
 * - Elements are added, removed, and checked for existence in constant time
 * (O(1)).
 * - Elements are enumerated in O(n). No guarantees are made on the ordering.
 *
 * ```
 * contract Example {
 *     // Add the library methods
 *     using EnumerableSet for EnumerableSet.AddressSet;
 *
 *     // Declare a set state variable
 *     EnumerableSet.AddressSet private mySet;
 * }
 * ```
 *
 * As of v3.3.0, sets of type `bytes32` (`Bytes32Set`), `address` (`AddressSet`)
 * and `uint256` (`UintSet`) are supported.
 *
 * [WARNING]
 * ====
 * Trying to delete such a structure from storage will likely result in data corruption, rendering the structure
 * unusable.
 * See https://github.com/ethereum/solidity/pull/11843[ethereum/solidity#11843] for more info.
 *
 * In order to clean an EnumerableSet, you can either remove all elements one by one or create a fresh instance using an
 * array of EnumerableSet.
 * ====
 */
library EnumerableSet {
    // To implement this library for multiple types with as little code
    // repetition as possible, we write it in terms of a generic Set type with
    // bytes32 values.
    // The Set implementation uses private functions, and user-facing
    // implementations (such as AddressSet) are just wrappers around the
    // underlying Set.
    // This means that we can only create new EnumerableSets for types that fit
    // in bytes32.

    struct Set {
        // Storage of set values
        bytes32[] _values;
        // Position of the value in the `values` array, plus 1 because index 0
        // means a value is not in the set.
        mapping(bytes32 => uint256) _indexes;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function _add(Set storage set, bytes32 value) private returns (bool) {
        if (!_contains(set, value)) {
            set._values.push(value);
            // The value is stored at length-1, but we add 1 to all indexes
            // and use 0 as a sentinel value
```

```solidity
            set._indexes[value] = set._values.length;
            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function _remove(Set storage set, bytes32 value) private returns (bool) {
        // We read and store the value's index to prevent multiple reads from the same storage slot
        uint256 valueIndex = set._indexes[value];

        if (valueIndex != 0) {
            // Equivalent to contains(set, value)
            // To delete an element from the _values array in O(1), we swap the element to delete with the last one in
            // the array, and then remove the last element (sometimes called as 'swap and pop').
            // This modifies the order of the array, as noted in {at}.

            uint256 toDeleteIndex = valueIndex - 1;
            uint256 lastIndex = set._values.length - 1;

            if (lastIndex != toDeleteIndex) {
                bytes32 lastValue = set._values[lastIndex];

                // Move the last value to the index where the value to delete is
                set._values[toDeleteIndex] = lastValue;
                // Update the index for the moved value
                set._indexes[lastValue] = valueIndex; // Replace lastValue's index to valueIndex
            }

            // Delete the slot where the moved value was stored
            set._values.pop();

            // Delete the index for the deleted slot
            delete set._indexes[value];

            return true;
        } else {
            return false;
        }
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function _contains(Set storage set, bytes32 value) private view returns (bool) {
        return set._indexes[value] != 0;
    }

    /**
     * @dev Returns the number of values on the set. O(1).
     */
    function _length(Set storage set) private view returns (uint256) {
        return set._values.length;
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function _at(Set storage set, uint256 index) private view returns (bytes32) {
        return set._values[index];
    }

    /**
     * @dev Return the entire set in an array
     *
     * WARNING: This operation will copy the entire storage to memory, which can be quite expensive. This is designed
     * to mostly be used by view accessors that are queried without any gas fees. Developers should keep in mind that
     * this function has an unbounded cost, and using it as part of a state-changing function may render the function
     * uncallable if the set grows to a point where copying to memory consumes too much gas to fit in a block.
     */
    function _values(Set storage set) private view returns (bytes32[] memory) {
        return set._values;
    }

    // Bytes32Set

    struct Bytes32Set {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(Bytes32Set storage set, bytes32 value) internal returns (bool) {
        return _add(set._inner, value);
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(Bytes32Set storage set, bytes32 value) internal returns (bool) {
        return _remove(set._inner, value);
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(Bytes32Set storage set, bytes32 value) internal view returns (bool) {
        return _contains(set._inner, value);
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(Bytes32Set storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
```

```solidity
    function at(Bytes32Set storage set, uint256 index) internal view returns (bytes32)
     {
        return _at(set._inner, index);
    }

    /**
     * @dev Return the entire set in an array
     *
     * WARNING: This operation will copy the entire storage to memory, which can be
     quite expensive. This is designed
     * to mostly be used by view accessors that are queried without any gas fees.
     Developers should keep in mind that
     * this function has an unbounded cost, and using it as part of a state-changing
     function may render the function
     * uncallable if the set grows to a point where copying to memory consumes too
     much gas to fit in a block.
     */
    function values(Bytes32Set storage set) internal view returns (bytes32[] memory) {
        bytes32[] memory store = _values(set._inner);
        bytes32[] memory result;

        /// @solidity memory-safe-assembly
        assembly {
            result := store
        }

        return result;
    }

    // AddressSet

    struct AddressSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(AddressSet storage set, address value) internal returns (bool) {
        return _add(set._inner, bytes32(uint256(uint160(value))));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(AddressSet storage set, address value) internal returns (bool) {
        return _remove(set._inner, bytes32(uint256(uint160(value))));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(AddressSet storage set, address value) internal view returns (
    bool) {
        return _contains(set._inner, bytes32(uint256(uint160(value))));
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(AddressSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(AddressSet storage set, uint256 index) internal view returns (address) {
        return address(uint160(uint256(_at(set._inner, index))));
    }

    /**
     * @dev Return the entire set in an array
     *
     * WARNING: This operation will copy the entire storage to memory, which can be quite expensive. This is designed
     * to mostly be used by view accessors that are queried without any gas fees. Developers should keep in mind that
     * this function has an unbounded cost, and using it as part of a state-changing function may render the function
     * uncallable if the set grows to a point where copying to memory consumes too much gas to fit in a block.
     */
    function values(AddressSet storage set) internal view returns (address[] memory) {
        bytes32[] memory store = _values(set._inner);
        address[] memory result;

        /// @solidity memory-safe-assembly
        assembly {
            result := store
        }

        return result;
    }

    // UintSet

    struct UintSet {
        Set _inner;
    }

    /**
     * @dev Add a value to a set. O(1).
     *
     * Returns true if the value was added to the set, that is if it was not
     * already present.
     */
    function add(UintSet storage set, uint256 value) internal returns (bool) {
        return _add(set._inner, bytes32(value));
    }

    /**
     * @dev Removes a value from a set. O(1).
     *
     * Returns true if the value was removed from the set, that is if it was
     * present.
     */
    function remove(UintSet storage set, uint256 value) internal returns (bool) {
        return _remove(set._inner, bytes32(value));
    }

    /**
     * @dev Returns true if the value is in the set. O(1).
     */
    function contains(UintSet storage set, uint256 value) internal view returns (bool) {
        return _contains(set._inner, bytes32(value));
    }

    /**
     * @dev Returns the number of values in the set. O(1).
     */
    function length(UintSet storage set) internal view returns (uint256) {
        return _length(set._inner);
    }

    /**
     * @dev Returns the value stored at position `index` in the set. O(1).
     *
     * Note that there are no guarantees on the ordering of values inside the
     * array, and it may change when more values are added or removed.
     *
     * Requirements:
     *
     * - `index` must be strictly less than {length}.
     */
    function at(UintSet storage set, uint256 index) internal view returns (uint256) {
        return uint256(_at(set._inner, index));
    }

    /**
     * @dev Return the entire set in an array
     *
     * WARNING: This operation will copy the entire storage to memory, which can be quite expensive. This is designed
     * to mostly be used by view accessors that are queried without any gas fees. Developers should keep in mind that
     * this function has an unbounded cost, and using it as part of a state-changing function may render the function
     * uncallable if the set grows to a point where copying to memory consumes too much gas to fit in a block.
     */
    function values(UintSet storage set) internal view returns (uint256[] memory) {
        bytes32[] memory store = _values(set._inner);
        uint256[] memory result;

        /// @solidity memory-safe-assembly
        assembly {
            result := store
        }

        return result;
    }
}
```