

---

# Hash\_v1.0.0 — Documentation

---

## Contents

<b>Library Description</b>	<b>2</b>
CONTENTS OF THE LIBRARY . . . . .	2
INHERITARY STRUCTURE . . . . .	2
ITERATOR IMPLEMENTATION . . . . .	2
<b>Classes Documentation</b>	<b>3</b>
IF. Hash . . . . .	4
1. HashSet . . . . .	6
1.1 HashSet -> Open . . . . .	8
1.2 HashSet -> Close . . . . .	10

## Library Description

This library implements some usefull Hash-Table based Data-Structures, such as Hash-Sets.

In particular, every **Hash-Structure** has been implemented following two different rules: **Open hashing** and **Close hashing**.

## CONTENTS OF THE LIBRARY

Since the Classes definitions have been made using templates, this is a *header-only* library.

However, in order to be optimized for the most common uses, have also been build some shared-library files *.so*, containing the **explicit template instantiations** of some primitive types.

The only precaution to take when compiling the library headers, is to set the flag **-std=c++20**, since it is needed in order to work properly.

The library is divided in multiple headers:

- **"IF\_Hash.hpp"**: here is defined the Interface `LDN::Hash`, with the derived Abstract Class `LDN::HashSet`, which is the Polimorphic Abstract Class for the **Hash-Structure**.
- **"Hash.hpp"**: the generic header, which includes all the **Hash-Structure** (it has been designed in order to be included alone).
- **"HashSet.hpp"**: the header file whic implements the Specifications for the `LDN::HashSet` Abstract Class.
- **"Specification.hpp"**: template implementation for the Specification Classes (such as `LDN::hash_set::Open`).

## INHERITARY STRUCTURE

The library starts with the definition of an Interface, represented by a simple abstract class: *"Hash"*.

Starting from this class, all the derivated ones create a Tree Structure, which allows to manage in a simple way all the Abstract **Hash-Structure**, and makes easy including new ones or derivating more Specifications for the already existing ones.

## ITERATOR IMPLEMENTATION

In order to be used in a simple and secure way, the `LDN::HashSet` Class has been implemented as an *Iterator*, both via Polimorphism and via Specification.

## Classes Documentation

In the "*Hash*" library have been implemented a total of four classes:

IF. **LDN::Hash:** the Root Interface, from which all the other **Hash-Structure** derive from.

1. **LDN::HashSet:** Abstract Class which defines an Hash-Set Data-Structure.

- (a) **LDN::hash\_set::Open:** Hash-Set with Open hashing rule.
- (b) **LDN::hash\_set::Close:** Hash-Set with Close hashing rule.

```
template <typename TYPE>
class Hash {

    protected:

        // attributes
        size_t size;
        size_t elements;

        //Method -- Hash Function
        virtual const size_t hash(const TYPE& key) const noexcept = 0;

    public:

        // constructor / destructor
        explicit Hash(const size_t& sz) : size(sz), elements(0) {}
        virtual ~Hash() noexcept = default;

        // getters
        inline const size_t& getSize() const noexcept { return this->size; }
        inline const size_t& getElements() const noexcept { return this->elements; }

        // Methods -- clear / contains / toString
        virtual void clear() noexcept = 0;
        virtual bool contains(const TYPE& key) const noexcept = 0;
        virtual std::string toString() const noexcept = 0;

};

// output stream operator «
template <typename TYPE>
inline std::ostream& operator<<(std::ostream& out_stream, const LDN::Hash<TYPE>& hash)
{
    out_stream << hash.toString();
    return out_stream;
}
```

```
size_t size;
```

The **Hash-Structure** size, it is a *protected* member, since the user is not able to modify it.

```
size_t elements;
```

The number of elements currently stored in the **Hash-Structure**, it is a *protected* member, since the user is not able to modify it.

```
explicit Hash(const size_t& sz);
```

This is the *constructor*, which assigns a `size_t` `sz` to the *protected* member `Hash.size`, then sets to 0 the *protected* member `Hash.elements`.

```
virtual ~Hash() noexcept = default;
```

This is the *destructor*, which is automatically invocated when deleting the object. It has been left as a *virtual method*, in order to ensure the Class to be abstract.

```
virtual void clear() noexcept = 0;
```

This *virtual method* imposes that each **Hash-Structure** has to provide a method to clean itself.

```
virtual bool contains(const TYPE& key) const noexcept = 0;
```

This *virtual method* imposes that each **Hash-Structure** has to provide a method to check if an element is stored.

```
virtual std::string toString() const noexcept = 0;
```

This *virtual method* imposes that each **Hash-Structure** has to provide a method to be printed correctly.

```
inline std::ostream& operator<<(std::ostream& out_stream, const LDN::Hash<TYPE>& hash);
```

Operator related to every **Hash-Structure**, which allows to print it via `std::ostream` using its method `Hash.toString()`.

---

## 1. HashSet

---

```
template <typename TYPE>
class HashSet : public Hash<TYPE> {

    public:

        // constructor / destructor
        explicit HashSet(const size_t& sz) : Hash<TYPE>(sz) {}
        virtual ~HashSet() noexcept = default;

        // Methods -- insert / remove / resize /loadFactor
        virtual void insert(const TYPE& key) = 0;
        virtual void insert(const TYPE& key) noexcept = 0;
        virtual void resize() noexcept = 0;
        inline const double loadFactor() const {
            return static_cast<double>(this->elements) / static_cast<double>(this->size)
        }

        // ----- Iterator Interface -----
        ...

        // Iterator Methods -- begin / end
        virtual Iterator begin() noexcept = 0;
        virtual Iterator end() noexcept = 0;

};
```

```
explicit HashSet(const size_t& sz);
```

This is the *constructor*, which calls back the constructor of the Interface `LDN::Hash<TYPE>`.

```
virtual ~HashSet() noexcept = default;
```

This is the *destructor*, which is automatically invoked when deleting the object. It has been left as a *virtual method*, in order to ensure the Class to be abstract.

```
virtual void insert(const TYPE& key) = 0;
```

This *virtual method* imposes that each **Hash-Set** has to provide a method to insert and store correctly a key (if possible).

```
virtual void remove(const TYPE& key) noexcept = 0;
```

This *virtual method* imposes that each **Hash-Set** has to provide a method to remove a key (if already stored).

```
virtual void resize() noexcept = 0;
```

This *virtual method* imposes that each **Hash-Set** has to provide a method to resize itself.

```
inline const double loadFactor() const
```

Method which allows to calculate the current *load factor* of the **Hash-Set**.

```
// ----- Iterator Interface -----
```

Technical implementation of the **Hash-Set** as an *Iterator*, if you want to see the technicalities, check the header *"IF\_Hash.hpp"*.

```
virtual Iterator begin() noexcept = 0;  
virtual Iterator end() noexcept = 0;
```

Method which allows the **Hash-Set** to be used as an *Iterator*, the user does not need to call them directly. They have been left as *virtual methods* since the implementation differs depending on the Specification.

---

## 1.1 HashSet -> Open

---

```
template <typename TYPE>
class Open : public LDN::HashSet<TYPE> {

    private:

        // helper Bucket definition
        struct Bucket{

            TYPE value;
            struct Bucket* next;

            explicit Bucket(const TYPE& key): value(key), next(nullptr) {}
            ~Bucket() noexcept = default;

        };

        // attributes
        std::unique_ptr<Bucket*> table;

        // Method -- Hash Function
        const size_t hash(const TYPE& key) const noexcept override;

    public:

        // constructor / destructor
        explicit Open(const size_t& sz);
        ~Open() noexcept override;

        // Hash Methods -- implementation
        ...

        // HashSet Methods -- implementation
        ...

        // ----- Iterator Implementation -----
        ...

        // Iterator Methods -- begin / end
        LDN::HashSet<TYPE>::Iterator begin() noexcept override {
            SpecificationIterator it(this->table.get(), this->size, 0);
            return typename LDN::HashSet<TYPE>::Iterator(it.clone());
        }
        LDN::HashSet<TYPE>::Iterator end() noexcept override {
            SpecificationIterator it(this->table.get(), this->size, this->size);
            return typename LDN::HashSet<TYPE>::Iterator(it.clone());
        }

};
```



```
struct Bucket{...};
```

Definition of an helper struct `LDN::hash_set::Open::Bucket`, private to the **Hash-Set** Specification, which allows to use the Hash-Structure with chaining (following the **Open hashing** rule).

```
std::unique_ptr<Bucket*> table;
```

Implementation of the Hash-Table used for the **Hash-Set**, it has been used a `std::unique_ptr<>` in order to keep the user from accessing the table directly (due to the concept of **Hash-Set**).

```
const size_t hash(const TYPE& key) const noexcept override;
```

Implementation of the *hash function*, it is based on the standard-library struct `std::hash<>`, allowing correct and complete management of the hashing rule following C++ standard.

```
explicit Open(const size_t& sz);
```

This is the *constructor*, which calls back the constructor of the Generalization `LDN::HashSet<TYPE>` and makes an unique Hash-Table.

```
~Open() noexcept override;
```

This is the *destructor*, which is automatically invocated when deleting the object.

```
// Hash Methods -- implementation
```

Implementation of the Methods declared by the Interface `LDN::Hash<TYPE>`.

```
// HashSet Methods -- implementation
```

Implementation of the Methods declared by the Generalization `LDN::HashSet<TYPE>`.

```
// ----- Iterator Implementation -----
```

Technical implementation of the **Hash-Set** as an *Iterator*, if you want to see the technicalities, check the header *"HashSet.hpp"*.

```
LDN::HashSet<TYPE>::Iterator begin() noexcept override;
LDN::HashSet<TYPE>::Iterator end() noexcept override;
```

Method which allows the **Hash-Set** to be used as an *Iterator*, the user does not need to call them directly. They implement the Polimorphic **Hash-Set** ones, so that both Generalization and Specification can be used as *Iterators*.

---

## 1.2 HashSet -> Close

---

```
template <typename TYPE>
class Close : public LDN::HashSet<TYPE> {

    private:

        // helper fake-constants declaration
        TYPE EMPTY;
        TYPE TOMBSTONE;

        // attributes
        std::unique_ptr<TYPE[]> table;
        bool quadratic;

        // Method -- Hash Function
        const size_t hash(const TYPE& key) const noexcept override;

    public:

        // constructor / destructor
        explicit Close(const size_t& sz, bool probing, TYPE empty, TYPE tombstone);
        ~Close() noexcept override = default;

        // Hash Methods -- implementation
        ...

        // HashSet Methods -- implementation
        ...

        // ----- Iterator Implementation -----
        ...

        // Iterator Methods -- begin / end
        LDN::HashSet<TYPE>::Iterator begin() noexcept override {...}
        LDN::HashSet<TYPE>::Iterator end() noexcept override {...}

};
```

```
TYPE EMPTY;
```

Declaration of an helper fake-constant, private to an instance of the **Hash-Set** Specification, which allows to use the Hash-Structure as a single array (following the **Close hashing** rule).

It is set by the *constructor*.

```
TYPE TOMBSTONE;
```

Declaration of an helper fake-constant, private to an instance of the **Hash-Set** Specification, which allows to use the Hash-Structure as a single array (following the **Close hashing** rule).

It is set by the *constructor*.

```
std::unique_ptr<TYPE[]> table;
```

Implementation of the Hash-Table used for the **Hash-Set**, it has been used a `std::unique_ptr<>` in order to keep the user from accessing the table directly (due to the concept of **Hash-Set**).

```
bool quadratic;
```

Flag wich represents the method of probing (linear if set to *false*, quadratic is set to *true*).

```
const size_t hash(const TYPE& key) const noexcept override;
```

Implementation of the *hash function*, it is based on the standard-library struct `std::hash<>`, allowing correct and complete management of the hashing rule following C++ standard.

```
explicit Close(const size_t& sz, bool probing, TYPE empty, TYPE tombstone);
```

This is the *constructor*, which calls back the constructor of the Generalization `LDN::HashSet<TYPE>` and makes an unique Hash-Table.

It also set for the only time the fake-constants `EMPTY` and `TOMBSTONE`.

```
~Close() noexcept override = default;
```

This is the *destructor*, which is automatically invoked when deleting the object.

It is set to default, since the `std::unique_ptr<>` delete itself automatically, and there are no other deletions (like chaining-related ones).

```
// Hash Methods -- implementation
```

Implementation of the Methods declared by the Interface `LDN::Hash<TYPE>`.

```
// HashSet Methods -- implementation
```

Implementation of the Methods declared by the Generalization `LDN::HashSet<TYPE>`.

```
// ----- Iterator Implementation -----
```

Technical implementation of the **Hash-Set** as an *Iterator*, if you want to see the technicalities, check the header *"HashSet.hpp"*.

```
LDN::HashSet<TYPE>::Iterator begin() noexcept override;  
LDN::HashSet<TYPE>::Iterator end() noexcept override;
```

Method which allows the **Hash-Set** to be used as an *Iterator*, the user does not need to call them directly. They implement the Polimorphic **Hash-Set** ones, so that both Generalization and Specification can be used as *Iterators*.