Persistent Data Structure

Generated by Doxygen 1.12.0

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Namespace Documentation

4.1 pds Namespace Reference

Classes

- class fpSet

  Fully persistent set container for sorted, unique objects.
- class pSet

  Partially persistent set container for sorted unique objects.

Variables

- const pds::version_t default_version = std::numeric_limits<pds::version_t>::max()

  Default version indicating the 'last_version' for insert & remove operations.

4.1.1 Variable Documentation

4.1.1.1 default_version

```
const pds::version_t pds::default_version = std::numeric_limits<pds::version_t>::max()
```

Default version indicating the 'last_version' for insert & remove operations.

Examples

D:/Fully-Persistent-DS/include/fpSet.hpp.

Chapter 5

Class Documentation

5.1 pds::fpSet< OBJ > Class Template Reference

Fully persistent set container for sorted, unique objects.

```
#include <fpSet.hpp>
```

Public Member Functions

- fpSet ()

  Construct a new object.
- pds::version_t insert (const OBJ &obj, pds::version_t version=default_version)

  Inserts an object into the set at a specific version.
- pds::version_t insert (OBJ &&obj, pds::version_t version=default_version)

  Inserts an object into the set at a specific version.
- pds::version_t remove (const OBJ &obj, pds::version_t version=default_version)

  Removes an object from the set at a specific version.
- pds::version_t remove (OBJ &&obj, pds::version_t version=default_version)

  Removes an object from the set at a specific version.
- bool contains (const OBJ &obj, pds::version_t version=MasterVersion)

  Check if an object is in the set in a specific version.
- std::vector< OBJ > to_vector (const pds::version_t version=MasterVersion)

  return a sorted set as std::vector<OBJ>.
- pds::version_t size (pds::version_t version=MasterVersion) const noexcept

  size of the set for 'version'.
- pds::version_t curr_version () const noexcept

  current version
- void print (pds::version_t version=MasterVersion)

  print the set sorted.

5.1.1 Detailed Description

template<class OBJ>
class pds::fpSet< OBJ >

Fully persistent set container for sorted, unique objects.

The fpSet class allows storing objects in a way that maintains historical versions of the set after each modification. Each version is preserved, enabling access to any past state of the set.

Template Parameters

OBJ	The object type, which must support `operator<` for sorting and provide either copy or move constructors.

Note

Space Complexity: O(N log(N))

- N represents the number of versions maintained (i.e., `last_version`).
- Every object is saved only once, regardless of how many versions it exists in.

Examples

D:/Fully-Persistent-DS/include/fpSet.hpp, and D:/Fully-Persistent-DS/include/pSet.hpp.

5.1.2 Constructor & Destructor Documentation

5.1.2.1 fpSet()

```
template<class OBJ >
pds::fpSet< OBJ >::fpSet ()
```

Construct a new object.

Also insert two init Versions:

- Version 0: see MasterVersion.
- Version 1: will save the init state version. (will always be empty).

Examples

D:/Fully-Persistent-DS/include/fpSet.hpp.

5.1.3 Member Function Documentation

5.1.3.1 contains()

```
template<class OBJ >
bool pds::fpSet< OBJ >::contains (
            const OBJ & obj,
            pds::version_t version = MasterVersion)
```

Check if an object is in the set in a specific version.

Parameters

obj	the object to query for.
version	version to check for. If the version is not specified then check for MasterVersion which mean: any version.

Exceptions

pds::VersionNotExist thrown if: version is bigger than what returned with 'curr_version()'

Returns

true if the object exists in the specified version; otherwise, false.

Note

Time complexity: O(log(N) ∗ log∗(N)) while N is the number of versions, i.e. N is last_version.

Examples

D:/Fully-Persistent-DS/include/fpSet.hpp, and D:/Fully-Persistent-DS/include/pSet.hpp.

5.1.3.2 curr_version()

```
template<class OBJ >
pds::version_t pds::fpSet< OBJ >::curr_version () const  [noexcept]
```

current version

Exceptions

No	exceptions.

Returns

pds::version_t 'last_version'.

Examples

D:/Fully-Persistent-DS/include/fpSet.hpp.

5.1.3.3 insert() [1/2]

```
template<class OBJ >
pds::version_t pds::fpSet< OBJ >::insert (
             const OBJ & obj,
             pds::version_t version = default_version)
```

Inserts an object into the set at a specific version.

This method adds the given object to the set and ensures that it appears in the specified version and subsequent versions. If the operation succeeds, a new version will be created.

Parameters

obj	object to insert.

Attention

Since fpSet saves a copy of 'obj', the move option is recommended.

Parameters

version	The version to insert. if 'version'=default_version insert to last version.

Exceptions

pds::ObjectAlreadyExist thrown if: 'version'=default_version and contains('obj', 'curr_version()') return true, or 'version' specify and contains('obj', 'version') return true.

- pds::VersionZeroIllegal thrown if: version is 0
- pds::VersionNotExist thrown if: version is bigger than what returned with 'curr_version()'

Returns

pds::version_t of the new version.

Note

Time complexity: O(log(N) $*$ log$*$(N)) while N is the number of versions, i.e. N is last_version.

Examples

D:/Fully-Persistent-DS/include/fpSet.hpp, and D:/Fully-Persistent-DS/include/pSet.hpp.

5.1.3.4 insert() [2/2]

```
template<class OBJ >
pds::version_t pds::fpSet< OBJ >::insert (
            OBJ && obj,
            pds::version_t version = default_version)
```

Inserts an object into the set at a specific version.

This method adds the given object to the set and ensures that it appears in the specified version and subsequent versions. If the operation succeeds, a new version will be created.

This is especially recommended for complex OBJ types, as it can significantly improve performance by avoiding deep copies of data.

Parameters

obj	object to insert.
version	The version to insert. if 'version'=default_version insert to last version.

Exceptions

pds::ObjectAlreadyExist thrown if: 'version'=default_version and contains('obj', 'curr_version()') return true, or 'version' specify and contains('obj', 'version') return true.

- pds::VersionZeroIllegal thrown if: version is 0
- pds::VersionNotExist thrown if: version is bigger than what returned with 'curr_version()'

Returns

pds::version_t of the new version.

Note

Time complexity: O(log(N) $*$ log$*$(N)) while N is the number of versions, i.e. N is last_version.

5.1.3.5 print()

```
template<class OBJ >
void pds::fpSet< OBJ >::print (
             pds::version_t version = MasterVersion)
```

print the set sorted.

The print style is: "Version X: {obj1, obj2, ...}"

Parameters

version	to print. If no version is specify then print the all versions sorted.

Exceptions

pds::VersionNotExist thrown if: version is bigger than what returned with 'curr_version()'

Note

Time complexity: O((K log$*$(N)) + log(N)) while K is the number of objects in 'version' and N is the number of versions, i.e. N is last_version.

Examples

D:/Fully-Persistent-DS/include/fpSet.hpp.

5.1.3.6 remove() [1/2]

```
template<class OBJ >
pds::version_t pds::fpSet< OBJ >::remove (
            const OBJ & obj,
            pds::version_t version = default_version)
```

Removes an object from the set at a specific version.

This method removes the object from the set, making it unavailable in the specified version and all subsequent versions. If the operation succeeds, a new version will be created.

Parameters

obj	object to remove.
version	the version to remove from. If 'version'=default_version remove from last_version.

Attention

if contains('obj', 'version') == false: exception will thrown.

Exceptions

pds::ObjectNotExist thrown if: 'obj' not exists in 'version'

- pds::VersionZeroIllegal thrown if: version is 0
- pds::VersionNotExist thrown if: version is bigger than what returned with 'curr_version()'

Returns

pds::version_t of the new version.

Note

Time complexity: O(log(N) ∗ log∗(N)) while N is the number of versions, i.e. N is last_version.

Examples

D:/Fully-Persistent-DS/include/fpSet.hpp, and D:/Fully-Persistent-DS/include/pSet.hpp.

5.1.3.7 remove() [2/2]

```
template<class OBJ >
pds::version_t pds::fpSet< OBJ >::remove (
            OBJ && obj,
            pds::version_t version = default_version)
```

Removes an object from the set at a specific version.

This method removes the object from the set, making it unavailable in the specified version and all subsequent versions. If the operation succeeds, a new version will be created.

Recommended for complex OBJ types, for avoiding deep copy of 'obj'.

Parameters

obj	object to remove.
version	the version to remove from. If 'version'=default_version remove from last_version.

Attention

if contains('obj', 'version') == false: exception will thrown.

Exceptions

pds::ObjectNotExist thrown if: 'obj' not exists in 'version'

- pds::VersionZeroIllegal thrown if: version is 0
- pds::VersionNotExist thrown if: version is bigger than what returned with 'curr_version()'

Returns

pds::version_t of the new version.

Note

Time complexity: O(log(N) ∗ log∗(N)) while N is the number of versions, i.e. N is last_version.

5.1.3.8 size()

```
template<class OBJ >
pds::version_t pds::fpSet< OBJ >::size (
             pds::version_t version = MasterVersion) const  [noexcept]
```

size of the set for 'version'.

Parameters

version	the version to return for.

Exceptions

No	exceptions.

Returns

pds::version_t a std::size_t. see pds::version_t. If the version is not specified: the size of all unique objects in all versions. If version not exists: 0

Examples

D:/Fully-Persistent-DS/include/fpSet.hpp.

5.1.3.9 to_vector()

```
template<class OBJ >
std::vector< OBJ > pds::fpSet< OBJ >::to_vector (
             const pds::version_t version = MasterVersion)
```

return a sorted set as std::vector<OBJ>.

Parameters

version	the version to return for. If the version is not specified then return the all objects in all versions as a sorted set.

Exceptions

pds::VersionNotExist thrown if: version is bigger than what returned with 'curr_version()'

Returns

std::vector<OBJ> a sorted set.

Note

Time complexity: O((K log∗(N)) + log(N)) while K is the number of objects in 'version' and N is the number of versions, i.e. N is last_version.

Examples

D:/Fully-Persistent-DS/include/fpSet.hpp.

The documentation for this class was generated from the following file:

- include/fpSet.hpp

5.2 pds::pSet< OBJ > Class Template Reference

Partially persistent set container for sorted unique objects.

```
#include <pSet.hpp>
```

Public Member Functions

- pSet ()

  Construct a new object.
- pds::version_t insert (const OBJ &obj)

  Insert an object to the set. If the operation succeeds, a new version will be created.
- pds::version_t insert (OBJ &&obj)

  Insert an rvalue object to the set. If the operation succeeds, a new version will be created.
- pds::version_t remove (const OBJ &obj)

  remove obj from the set. If the operation succeeds, a new version will be created.
- pds::version_t remove (OBJ &&obj)

  remove an rvalue object from the set. If the operation succeeds, a new version will be created.
- bool contains (const OBJ &obj, pds::version_t version=MasterVersion)

  check if obj is in the set.
- std::vector< OBJ > to_vector (const pds::version_t version=MasterVersion)

  return a sorted set as std::vector<OBJ>.
- std::size_t size (pds::version_t version=MasterVersion) const

  size of the set for 'version'.
- pds::version_t curr_version () const

  current version
- void print (pds::version_t version=MasterVersion)

  print the set sorted.

5.2.1 Detailed Description

template<class OBJ>
class pds::pSet< OBJ >

Partially persistent set container for sorted unique objects.

Partially persistent set container for sorted, unique objects.

Template Parameters

OBJ	The object type, which must support `operator<` for sorting and provide either copy or move constructors.

Note

Space Complexity: O(N) while N is the number of versions, i.e. N is last_version. Every 'obj' will save exactly once regardless of the number of versions it was inserted into.

The pSet class allows storing objects in a way that maintains historical versions of the set after each modification. Each version is preserved, enabling search and get methods for any past state of the set.

Template Parameters

OBJ	The object type, which must support `operator<` for sorting and provide either copy or move constructors.

Note

Space Complexity: O(N)

- N represents the number of versions maintained (i.e., `last_version`).
- Every object is saved only once, regardless of how many versions it exists in.

Examples

D:/Fully-Persistent-DS/include/pSet.hpp.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 pSet()

```
template<class OBJ >
pds::pSet< OBJ >::pSet ()
```

Construct a new object.

Also insert two init Versions:

- Version 0: see MasterVersion.
- Version 1: will save the init state version. (will always be empty).

Examples

D:/Fully-Persistent-DS/include/pSet.hpp.

5.2.3 Member Function Documentation

5.2.3.1 contains()

```
template<class OBJ >
bool pds::pSet< OBJ >::contains (
             const OBJ & obj,
             pds::version_t version = MasterVersion)
```

check if obj is in the set.

Parameters

obj	the object to query for.
version	version to check for. If the version is not specified then check for MasterVersion which mean: any version.

Exceptions

pds::VersionNotExist thrown if: version is bigger than what returned with 'curr_version()'

Returns

true if 'obj' exists in 'version', false otherwise.

Note

Time complexity: O((log(N))^2) while N is the number of versions, i.e. N is last_version.

Examples

D:/Fully-Persistent-DS/include/pSet.hpp.

5.2.3.2 curr_version()

```
template<class OBJ >
pds::version_t pds::pSet< OBJ >::curr_version () const
```

current version

Exceptions

No	exceptions.

Returns

pds::version_t 'last_version'.

Examples

D:/Fully-Persistent-DS/include/pSet.hpp.

5.2.3.3 insert() [1/2]

```
template<class OBJ >
pds::version_t pds::pSet< OBJ >::insert (
             const OBJ & obj)
```

Insert an object to the set. If the operation succeeds, a new version will be created.

Parameters

obj	object to insert.

Attention

Since pSet saves a copy of 'obj', the move option is recommended.

Exceptions

pds::ObjectAlreadyExist thrown if: 'obj' exists in last version, i.e. contains('obj', curr_version()) return true.

Returns

pds::version_t of the new version. see pds::version_t.

Note

Time complexity: O((log(N))^2) while N is the number of versions, i.e. N is last_version.

Examples

D:/Fully-Persistent-DS/include/pSet.hpp.

5.2.3.4 insert() [2/2]

```
template<class OBJ >
pds::version_t pds::pSet< OBJ >::insert (
             OBJ && obj)
```

Insert an rvalue object to the set. If the operation succeeds, a new version will be created.

This is especially recommended for complex OBJ types, as it can significantly improve performance by avoiding deep copies of data.

Parameters

obj	object to insert.

Attention

Since pSet saves a copy of 'obj', the move option is more recommended.

Exceptions

pds::ObjectAlreadyExist thrown if: 'obj' exists in last version, i.e. contains('obj', curr_version()) return true.

Returns

pds::version_t of the new version. see pds::version_t.

Note

Time complexity: O((log(N))^2) while N is the number of versions, i.e. N is last_version.

5.2.3.5 print()

```
template<class OBJ >
void pds::pSet< OBJ >::print (
            pds::version_t version = MasterVersion)
```

print the set sorted.

The print style is: "Version X: {obj1, obj2, ...}"

Parameters

version	to print. If no version is specify then print the all versions sorted.

Exceptions

pds::VersionNotExist thrown if: version is bigger than what returned with 'curr_version()'

Note

Time complexity: O(K log(N)) while K is the number of objects in 'version' and N is the number of versions, i.e. N is last_version.

Examples

D:/Fully-Persistent-DS/include/pSet.hpp.

5.2.3.6 remove() [1/2]

```
template<class OBJ >
pds::version_t pds::pSet< OBJ >::remove (
            const OBJ & obj)
```

remove obj from the set. If the operation succeeds, a new version will be created.

Parameters

obj	object to remove.

Exceptions

pds::ObjectNotExist thrown if: 'obj' does not exist in the last version, i.e. contains('obj', curr_version()) return false.

Returns

pds::version_t of the new version.

Note

Time complexity: O((log(N))^2) while N is the number of versions, i.e. N is last_version.

Examples

D:/Fully-Persistent-DS/include/pSet.hpp.

5.2.3.7 remove() [2/2]

```
template<class OBJ >
pds::version_t pds::pSet< OBJ >::remove (
            OBJ && obj)
```

remove an rvalue object from the set. If the operation succeeds, a new version will be created.

Parameters

obj	object to remove.

Exceptions

pds::ObjectNotExist thrown if: 'obj' not exists in last version, i.e. contains('obj', curr_version()) return false.

Returns

pds::version_t of the new version.

Note

Time complexity: O((log(N))^2) while N is the number of versions, i.e. N is last_version.

5.2.3.8 size()

```
template<class OBJ >
std::size_t pds::pSet< OBJ >::size (
            pds::version_t version = MasterVersion) const
```

size of the set for 'version'.

Parameters

version	the version to return for.

Exceptions

No	exceptions.

Returns

pds::version_t a std::size_t. see pds::version_t. If version is not specify: the size of all unique objects in all versions. If version not exists: 0

Examples

D:/Fully-Persistent-DS/include/pSet.hpp.

5.2.3.9 to_vector()

```
template<class OBJ >
std::vector< OBJ > pds::pSet< OBJ >::to_vector (
            const pds::version_t version = MasterVersion)
```

return a sorted set as std::vector<OBJ>.

Parameters

version	the version to return for. If version is not specify then return the all objects in all versions as a sorted set.

Exceptions

pds::VersionNotExist thrown if: version is bigger than what returned with 'curr_version()'

Returns

std::vector<OBJ> a sorted set.

Note

Time complexity: O(K log(N)) while K is the number of objects in 'version' and N is the number of versions, i.e. N is last_version.

Examples

D:/Fully-Persistent-DS/include/pSet.hpp.

The documentation for this class was generated from the following file:

- include/pSet.hpp

Chapter 6

File Documentation

6.1 include/fpSet.hpp File Reference

fully persistent set container.

```
#include "internal/fpSetTracker.hpp"
```

Classes

- class pds::fpSet< OBJ >

  Fully persistent set container for sorted, unique objects.

Namespaces

- namespace pds

Variables

- const pds::version_t pds::default_version = std::numeric_limits<pds::version_t>::max()

  Default version indicating the 'last_version' for insert & remove operations.

6.1.1 Detailed Description

fully persistent set container.

Author

Assaf Bardugo (`https://github.com/AssafBardugo`)

Version

0.1

Date

2024-10-09

6.2 fpSet.hpp

Go to the documentation of this file.

```

#ifndef FULLY_PERSISTENT_SET_HPP
#define FULLY_PERSISTENT_SET_HPP

#include "internal/fpSetTracker.hpp"

namespace pds{

    const pds::version_t default_version = std::numeric_limits<pds::version_t>::max();


    template <class OBJ>
    class fpSet{

        pds::fpFatNodePtr<OBJ> root;
        pds::version_t last_version;
        std::vector<pds::version_t> sizes;

    public:
        fpSet();

        pds::version_t insert(const OBJ& obj, pds::version_t version = default_version);


        pds::version_t insert(OBJ&& obj, pds::version_t version = default_version);


        pds::version_t remove(const OBJ& obj, pds::version_t version = default_version);


        pds::version_t remove(OBJ&& obj, pds::version_t version = default_version);


        bool contains(const OBJ& obj, pds::version_t version = MasterVersion);


        std::vector<OBJ> to_vector(const pds::version_t version = MasterVersion);


        pds::version_t size(pds::version_t version = MasterVersion) const noexcept;


        pds::version_t curr_version() const noexcept;


        void print(pds::version_t version = MasterVersion);

    private:
        template <typename T>
        pds::version_t insert_impl(T&& obj, pds::version_t version);
    };
};


template <class OBJ>
pds::fpSet<OBJ>::fpSet() : root(1), last_version(1), sizes{0, 0} {
}


template <class OBJ>
pds::version_t pds::fpSet<OBJ>::insert(const OBJ& obj, pds::version_t version){

    return insert_impl(obj, version);
}


template <class OBJ>
pds::version_t pds::fpSet<OBJ>::insert(OBJ&& obj, pds::version_t version){

    return insert_impl(std::move(obj), version);
}


template <class OBJ>
template <typename T>
pds::version_t pds::fpSet<OBJ>::insert_impl(T&& obj, pds::version_t version){

    if(version == default_version)
        version = last_version;

    if(version == MasterVersion)
        throw pds::VersionZeroIllegal("Version 0 is not valid for insert");

    PDS_THROW_IF_VERSION_NOT_EXIST("fpSet::insert_impl", version, last_version);

    if(contains(obj, version))
        throw pds::ObjectAlreadyExist(
                "fpSet::insert: Version " + std::to_string(version) + " already contains this object"
            );

    pds::version_t new_version = last_version + 1;

    pds::fpSetTracker<OBJ> tracker(root, version);

    // Inserting a new version to the tree:
    while(tracker.not_null()){

        tracker[new_version] = *tracker;

        if(obj < tracker.obj()){

            tracker.add_right_map(new_version);
            tracker = tracker.left();
        }
        else{
            tracker.add_left_map(new_version);
            tracker = tracker.right();
        }
    }

    pds::fpSetTracker<OBJ> track_master(root, MasterVersion);

    while(track_master.not_null()){

        if(obj < track_master.obj()){

            track_master = track_master.left();
        }
        else if(track_master.obj() < obj){

            track_master = track_master.right();
        }
        else{
            tracker[new_version] = *track_master;
            tracker.set_left_at(new_version) = nullptr;
            tracker.set_right_at(new_version) = nullptr;
            break;
        }
    }

    if(track_master.null()){

        track_master[MasterVersion] = std::make_shared<pds::fpFatNode<OBJ»(std::forward<T>(obj),
new_version);

        ++sizes[MasterVersion];
        tracker[new_version] = *track_master;
    }

    // push the size of the new version
    sizes.push_back(sizes[version] + 1);

    return (last_version = new_version);
}


template <class OBJ>
pds::version_t pds::fpSet<OBJ>::remove(const OBJ& obj, pds::version_t version){

    if(version == default_version)
        version = last_version;

    if(version == MasterVersion)
        throw pds::VersionZeroIllegal("Version 0 is not valid for remove");

    PDS_THROW_IF_VERSION_NOT_EXIST("fpSet::remove", version, last_version);

    if(!contains(obj, version))
        throw pds::ObjectNotExist(
            "pds::fpSet::remove: Attempting to remove an object from Version "
            + std::to_string(version) + ". But the object is not exists for this Version"
        );


    pds::version_t new_version = last_version + 1;
    pds::fpSetTracker<OBJ> tracker(root, version);

    // Inserting a new version to the tree:
    while(tracker.not_null()){

        if(obj < tracker.obj()){

            tracker[new_version] = *tracker;
            tracker.add_right_map(new_version);
            tracker = tracker.left();
        }
        else if(tracker.obj() < obj){

            tracker[new_version] = *tracker;
            tracker.add_left_map(new_version);
            tracker = tracker.right();
        }
        else break;
    }

    pds::fpSetTracker<OBJ> to_remove = tracker;

    if(to_remove.left_null() && to_remove.right_null()){

        to_remove[new_version] = nullptr;
    }
    else if(to_remove.left_null() || to_remove.right_null()){

        if(to_remove.left_null())
            tracker = tracker.right();
        else
            tracker = tracker.left();

        to_remove[new_version] = *tracker;
        to_remove.add_left_map(new_version);
        to_remove.add_right_map(new_version);
    }
    else{
        pds::fpSetTracker<OBJ> track_to_leaf = tracker;
        track_to_leaf = track_to_leaf.right();

        if(track_to_leaf.left_null()){

            to_remove[new_version] = *track_to_leaf;
            track_to_leaf.add_left_map(new_version);
            track_to_leaf.add_right_map(new_version);

            tracker = tracker.left();
            to_remove.set_left_at(new_version) = *tracker;
            tracker.add_left_map(new_version);
            tracker.add_right_map(new_version);
        }
        else{

/***************  ILLUSTRATION  **************************

          +++ version +++    |   +++ new_version +++
                             |
      to_remove:[6]          |           [11]
               /   \        ==»        /   \
            [4]     [16]     |       [4]     [16]
                   /         |              /
               [11]          |          [13]
                   \         |
                    [13]     |

***********************************************************/

            while(track_to_leaf.left_not_null()){

                track_to_leaf[new_version] = *track_to_leaf;
                track_to_leaf.add_right_map(new_version);
                track_to_leaf = track_to_leaf.left();
            }
            to_remove[new_version] = *track_to_leaf;

            if(track_to_leaf.right_not_null()){

                track_to_leaf[new_version] = track_to_leaf.get_right();
                track_to_leaf.add_left_map(new_version);

                track_to_leaf = track_to_leaf.right();
                track_to_leaf.add_left_map(new_version);
                track_to_leaf.add_right_map(new_version);
            }
            else{
                track_to_leaf[new_version] = nullptr;
            }

            to_remove.set_right_at(new_version) = tracker.get_right();
            to_remove.set_left_at(new_version) = tracker.get_left();
            tracker = tracker.left();
            tracker.add_left_map(new_version);
            tracker.add_right_map(new_version);
        }
    }

    // push the size of the new version
    sizes.push_back(sizes[version] - 1);

    return (last_version = new_version);
}


template <class OBJ>
pds::version_t pds::fpSet<OBJ>::remove(OBJ&& obj, pds::version_t version){

    return remove(std::as_const(obj), version);
}


template <class OBJ>
bool pds::fpSet<OBJ>::contains(const OBJ& obj, pds::version_t version){

    PDS_THROW_IF_VERSION_NOT_EXIST("fpSet::contains", version, last_version);

    pds::fpSetTracker<OBJ> tracker(root, version);

    while(tracker.not_null()){

        if(obj < tracker.obj()){

            tracker = tracker.left();
        }
        else if(tracker.obj() < obj){

            tracker = tracker.right();
        }
        else return true;
    }
    return false;
}


template <class OBJ>
std::vector<OBJ> pds::fpSet<OBJ>::to_vector(const pds::version_t version) {

    PDS_THROW_IF_VERSION_NOT_EXIST("fpSet::to_vector", version, last_version);

    std::vector<OBJ> obj_vec;

    /*** Inorder Stack Traversal ***/
    std::stack<pds::fpSetTracker<OBJ>> trav_stack;
    pds::fpSetTracker<OBJ> tracker(root, version);

    while(tracker.not_null() || !trav_stack.empty()){

        while(tracker.not_null()){

            trav_stack.push(tracker);
            tracker = tracker.left();
        }
        tracker = trav_stack.top();
        trav_stack.pop();

        obj_vec.push_back(tracker.obj());
        tracker = tracker.right();
    }
    return obj_vec;
}


template <class OBJ>
pds::version_t pds::fpSet<OBJ>::size(pds::version_t version) const noexcept {

    try{
        return sizes.at(version);
    }
    catch(const std::out_of_range&){

        return 0;
    }
}


template <class OBJ>
pds::version_t pds::fpSet<OBJ>::curr_version() const noexcept {

    return last_version;
}


template <class OBJ>
void pds::fpSet<OBJ>::print(pds::version_t version){

    PDS_THROW_IF_VERSION_NOT_EXIST("pset::print", version, last_version);

    std::vector<OBJ> vec = to_vector(version);

    std::cout « "Version " « version « ": {";
    if(!vec.empty()){
        std::copy(vec.begin(), vec.end() - 1, std::ostream_iterator<OBJ>(std::cout, ", "));
        std::cout « vec.back();
    }
    std::cout « "}" « std::endl;
}


#endif /* FULLY_PERSISTENT_SET_HPP */
```

6.3 include/pSet.hpp File Reference

Partially persistent set container.

```
#include "internal/pSetTracker.hpp"
```

Classes

- class pds::pSet< OBJ >

    Partially persistent set container for sorted unique objects.

Namespaces

- namespace pds

6.3.1 Detailed Description

Partially persistent set container.

Author

Assaf Bardugo (https://github.com/AssafBardugo)

Version

0.1

Date

2024-10-28

6.4 pSet.hpp

Go to the documentation of this file.

```
#ifndef PARTIALLY_PERSISTENT_SET_HPP
#define PARTIALLY_PERSISTENT_SET_HPP

#include "internal/pSetTracker.hpp"

namespace pds{

    template <class OBJ>
    class pSet{

        pds::pFatNodePtr<OBJ> root;
        pds::version_t last_version;
        std::vector<pds::version_t> sizes;

    public:
        pSet();


        pds::version_t insert(const OBJ& obj);


        pds::version_t insert(OBJ&& obj);


        pds::version_t remove(const OBJ& obj);


        pds::version_t remove(OBJ&& obj);


        bool contains(const OBJ& obj, pds::version_t version = MasterVersion);


        std::vector<OBJ> to_vector(const pds::version_t version = MasterVersion);


        std::size_t size(pds::version_t version = MasterVersion) const;


        pds::version_t curr_version() const;


        void print(pds::version_t version = MasterVersion);

    private:
        template <typename T>
        pds::version_t insert_impl(T&& obj);
    };
};


template <class OBJ>
pds::pSet<OBJ>::pSet() : root(1), last_version(1), sizes{0, 0} {
}

template <class OBJ>
pds::version_t pds::pSet<OBJ>::insert(const OBJ& obj){

    return insert_impl(obj);
}

template <class OBJ>
pds::version_t pds::pSet<OBJ>::insert(OBJ&& obj){

    return insert_impl(std::move(obj));
}

template <class OBJ>
template <typename T>
pds::version_t pds::pSet<OBJ>::insert_impl(T&& obj){

    pds::pSetTracker<OBJ> tracker(root);

    while(tracker.not_null()){

        if(obj < tracker.obj()){

            tracker = tracker.left();
        }
        else if(tracker.obj() < obj){

            tracker = tracker.right();
        }
        else{
            throw pds::ObjectAlreadyExist(
                "pSet::insert: Version " + std::to_string(last_version) + " already contains this object"
            );
        }
    }

    pds::version_t new_version = last_version + 1;

    pds::pSetTracker<OBJ> track_master(root);

    while(track_master.not_null_at(MasterVersion)){

        if(obj < track_master.obj_at(MasterVersion)){

            track_master = track_master.left_at(MasterVersion);
        }
        else if(track_master.obj_at(MasterVersion) < obj){

            track_master = track_master.right_at(MasterVersion);
        }
        else{
            tracker[new_version] = track_master.at(MasterVersion);
            break;
        }
    }

    if(track_master.null_at(MasterVersion)){

        track_master[MasterVersion] = std::make_shared<pds::pFatNode<OBJ»(std::forward<T>(obj), new_version);

        ++sizes[MasterVersion];
        tracker[new_version] = track_master.at(MasterVersion);
    }
    else{
        tracker.set_track_version(new_version);

        if(tracker.left_null() == false){

            tracker.set_left(new_version) = nullptr;
        }
        if(tracker.right_null() == false){

            tracker.set_right(new_version) = nullptr;
        }
    }

    sizes.push_back(sizes[last_version] + 1);

    return (last_version = new_version);
}

template <class OBJ>
pds::version_t pds::pSet<OBJ>::remove(OBJ&& obj){

    return remove(std::as_const(obj));
}

template <class OBJ>
pds::version_t pds::pSet<OBJ>::remove(const OBJ& obj){

    pds::pSetTracker<OBJ> tracker(root);

    while(tracker.not_null()){

        if(obj < tracker.obj()){

            tracker = tracker.left();
        }
        else if(tracker.obj() < obj){

            tracker = tracker.right();
        }
        else break;
    }

    if(tracker.null())
        throw pds::ObjectNotExist(
            "pds::pset::remove: Attempting to remove an object but the object is not exists"
        );

    pds::version_t new_version = last_version + 1;

    pds::pSetTracker<OBJ> to_remove = tracker;
    pds::pSetTracker<OBJ> track_to_leaf = tracker;

    if(to_remove.left_null()){

        to_remove[new_version] = tracker.get_right();
    }
    else if(to_remove.right_null()){

        to_remove[new_version] = tracker.get_left();
    }
    else{
        track_to_leaf = track_to_leaf.right();

        if(track_to_leaf.left_null()){

            to_remove[new_version] = *track_to_leaf;
            to_remove.set_track_version(new_version);
            to_remove.set_left(new_version) = tracker.get_left();
        }
        else{
            while(!track_to_leaf.left_null()){

                track_to_leaf = track_to_leaf.left();
            }
            track_to_leaf[new_version] = track_to_leaf.get_right();

            to_remove[new_version] = *track_to_leaf;
            to_remove.set_track_version(new_version);
            to_remove.set_left(new_version) = tracker.get_left();
            to_remove.set_right(new_version) = tracker.get_right();
        }
    }
    // push the size of the new version
    sizes.push_back(sizes[last_version] - 1);

    return (last_version = new_version);
}

template <class OBJ>
bool pds::pSet<OBJ>::contains(const OBJ& obj, pds::version_t version) {

    PDS_THROW_IF_VERSION_NOT_EXIST("pSet::contains", version, last_version);

    pds::pSetTracker<OBJ> tracker(root);

    while(tracker.not_null_at(version)){

        if(obj < tracker.obj_at(version)){

            tracker = tracker.left_at(version);
        }
        else if(tracker.obj_at(version) < obj){

            tracker = tracker.right_at(version);
        }
        else return true;
    }
    return false;
}

template <class OBJ>
std::vector<OBJ> pds::pSet<OBJ>::to_vector(const pds::version_t version) {

    PDS_THROW_IF_VERSION_NOT_EXIST("pSet::to_vector", version, last_version);

    std::vector<OBJ> obj_vec;

    /*** Inorder Stack Traversal ***/
    std::stack<pds::pSetTracker<OBJ>> trav_stack;
    pds::pSetTracker<OBJ> tracker(root);

    while(tracker.not_null_at(version) || !trav_stack.empty()){

        while(tracker.not_null_at(version)){

            trav_stack.push(tracker);
            tracker = tracker.left_at(version);
        }
        tracker = trav_stack.top();
        trav_stack.pop();

        obj_vec.push_back(tracker.obj_at(version));
        tracker = tracker.right_at(version);
    }
    return obj_vec;
}
```

```

template <class OBJ>
std::size_t pds::pSet<OBJ>::size(pds::version_t version) const {

    try{
        return sizes.at(version);
    }
    catch(const std::out_of_range&){

        return 0;
    }
}

template <class OBJ>
pds::version_t pds::pSet<OBJ>::curr_version() const {

    return last_version;
}

template <class OBJ>
void pds::pSet<OBJ>::print(pds::version_t version){

    PDS_THROW_IF_VERSION_NOT_EXIST("pSet::print", version, last_version);

    std::vector<OBJ> vec = to_vector(version);

    std::cout « "Version " « version « ": {";
    if(!vec.empty()){
        std::copy(vec.begin(), vec.end() - 1, std::ostream_iterator<OBJ>(std::cout, ", "));
        std::cout « vec.back();
    }
    std::cout « "}" « std::endl;
}


#endif /* PARTIALLY_PERSISTENT_SET_HPP */
```

Chapter 7

Examples

7.1 D:/Fully-Persistent-DS/include/pSet.hpp

```
pds::fpSet<int> vrsSet;
vrsSet.insert(5);
vrsSet.insert(10);
vrsSet.remove(5);
// Access previous version states
assert(vrsSet.contains(5, 2) == true);

#ifndef PARTIALLY_PERSISTENT_SET_HPP
#define PARTIALLY_PERSISTENT_SET_HPP

#include "internal/pSetTracker.hpp"

namespace pds{

    template <class OBJ>
    class pSet{

        pds::pFatNodePtr<OBJ> root;
        pds::version_t last_version;
        std::vector<pds::version_t> sizes;

    public:
        pSet();


        pds::version_t insert(const OBJ& obj);


        pds::version_t insert(OBJ&& obj);


        pds::version_t remove(const OBJ& obj);


        pds::version_t remove(OBJ&& obj);


        bool contains(const OBJ& obj, pds::version_t version = MasterVersion);


        std::vector<OBJ> to_vector(const pds::version_t version = MasterVersion);


        std::size_t size(pds::version_t version = MasterVersion) const;


        pds::version_t curr_version() const;


        void print(pds::version_t version = MasterVersion);

    private:
        template <typename T>
        pds::version_t insert_impl(T&& obj);
    };
};


template <class OBJ>
```

```
pds::pSet<OBJ>::pSet() : root(1), last_version(1), sizes{0, 0} {
}

template <class OBJ>
pds::version_t pds::pSet<OBJ>::insert(const OBJ& obj){

    return insert_impl(obj);
}

template <class OBJ>
pds::version_t pds::pSet<OBJ>::insert(OBJ&& obj){

    return insert_impl(std::move(obj));
}

template <class OBJ>
template <typename T>
pds::version_t pds::pSet<OBJ>::insert_impl(T&& obj){

    pds::pSetTracker<OBJ> tracker(root);

    while(tracker.not_null()){

        if(obj < tracker.obj()){

            tracker = tracker.left();
        }
        else if(tracker.obj() < obj){

            tracker = tracker.right();
        }
        else{
            throw pds::ObjectAlreadyExist(
                "pSet::insert: Version " + std::to_string(last_version) + " already contains this object"
            );
        }
    }

    pds::version_t new_version = last_version + 1;

    pds::pSetTracker<OBJ> track_master(root);

    while(track_master.not_null_at(MasterVersion)){

        if(obj < track_master.obj_at(MasterVersion)){

            track_master = track_master.left_at(MasterVersion);
        }
        else if(track_master.obj_at(MasterVersion) < obj){

            track_master = track_master.right_at(MasterVersion);
        }
        else{
            tracker[new_version] = track_master.at(MasterVersion);
            break;
        }
    }

    if(track_master.null_at(MasterVersion)){

        track_master[MasterVersion] = std::make_shared<pds::pFatNode<OBJ»(std::forward<T>(obj),
      new_version);

        ++sizes[MasterVersion];
        tracker[new_version] = track_master.at(MasterVersion);
    }
    else{
        tracker.set_track_version(new_version);

        if(tracker.left_null() == false){

            tracker.set_left(new_version) = nullptr;
        }
        if(tracker.right_null() == false){

            tracker.set_right(new_version) = nullptr;
        }
    }

    sizes.push_back(sizes[last_version] + 1);

    return (last_version = new_version);
}

template <class OBJ>
```

```
pds::version_t pds::pSet<OBJ>::remove(OBJ&& obj){

    return remove(std::as_const(obj));
}

template <class OBJ>
pds::version_t pds::pSet<OBJ>::remove(const OBJ& obj){

    pds::pSetTracker<OBJ> tracker(root);

    while(tracker.not_null()){

        if(obj < tracker.obj()){

            tracker = tracker.left();
        }
        else if(tracker.obj() < obj){

            tracker = tracker.right();
        }
        else break;
    }

    if(tracker.null())
        throw pds::ObjectNotExist(
            "pds::pset::remove: Attempting to remove an object but the object is not exists"
        );

    pds::version_t new_version = last_version + 1;

    pds::pSetTracker<OBJ> to_remove = tracker;
    pds::pSetTracker<OBJ> track_to_leaf = tracker;

    if(to_remove.left_null()){

        to_remove[new_version] = tracker.get_right();
    }
    else if(to_remove.right_null()){

        to_remove[new_version] = tracker.get_left();
    }
    else{
        track_to_leaf = track_to_leaf.right();

        if(track_to_leaf.left_null()){

            to_remove[new_version] = *track_to_leaf;
            to_remove.set_track_version(new_version);
            to_remove.set_left(new_version) = tracker.get_left();
        }
        else{
            while(!track_to_leaf.left_null()){

                track_to_leaf = track_to_leaf.left();
            }
            track_to_leaf[new_version] = track_to_leaf.get_right();

            to_remove[new_version] = *track_to_leaf;
            to_remove.set_track_version(new_version);
            to_remove.set_left(new_version) = tracker.get_left();
            to_remove.set_right(new_version) = tracker.get_right();
        }
    }
    // push the size of the new version
    sizes.push_back(sizes[last_version] - 1);

    return (last_version = new_version);
}

template <class OBJ>
bool pds::pSet<OBJ>::contains(const OBJ& obj, pds::version_t version) {

    PDS_THROW_IF_VERSION_NOT_EXIST("pSet::contains", version, last_version);

    pds::pSetTracker<OBJ> tracker(root);

    while(tracker.not_null_at(version)){

        if(obj < tracker.obj_at(version)){

            tracker = tracker.left_at(version);
        }
        else if(tracker.obj_at(version) < obj){

            tracker = tracker.right_at(version);
        }
        else return true;
    }
    return false;
}

template <class OBJ>
std::vector<OBJ> pds::pSet<OBJ>::to_vector(const pds::version_t version) {

    PDS_THROW_IF_VERSION_NOT_EXIST("pSet::to_vector", version, last_version);

    std::vector<OBJ> obj_vec;

    /*** Inorder Stack Traversal ***/
    std::stack<pds::pSetTracker<OBJ» trav_stack;
    pds::pSetTracker<OBJ> tracker(root);

    while(tracker.not_null_at(version) || !trav_stack.empty()){

        while(tracker.not_null_at(version)){

            trav_stack.push(tracker);
            tracker = tracker.left_at(version);
        }
        tracker = trav_stack.top();
        trav_stack.pop();

        obj_vec.push_back(tracker.obj_at(version));
        tracker = tracker.right_at(version);
    }
    return obj_vec;
}

template <class OBJ>
std::size_t pds::pSet<OBJ>::size(pds::version_t version) const {

    try{
        return sizes.at(version);
    }
    catch(const std::out_of_range&){

        return 0;
    }
}

template <class OBJ>
pds::version_t pds::pSet<OBJ>::curr_version() const {

    return last_version;
}

template <class OBJ>
void pds::pSet<OBJ>::print(pds::version_t version){

    PDS_THROW_IF_VERSION_NOT_EXIST("pSet::print", version, last_version);

    std::vector<OBJ> vec = to_vector(version);

    std::cout « "Version " « version « ": {";
    if(!vec.empty()){
        std::copy(vec.begin(), vec.end() - 1, std::ostream_iterator<OBJ>(std::cout, ", "));
        std::cout « vec.back();
    }
    std::cout « "}" « std::endl;
}


#endif /* PARTIALLY_PERSISTENT_SET_HPP */
```

7.2 D:/Fully-Persistent-DS/include/fpSet.hpp

```
pds::fpSet<int> vrsSet;
vrsSet.insert(5);
vrsSet.insert(10);
vrsSet.remove(5);
// Access previous version states

#ifndef FULLY_PERSISTENT_SET_HPP
```

```
#define FULLY_PERSISTENT_SET_HPP

#include "internal/fpSetTracker.hpp"

namespace pds{

    const pds::version_t default_version = std::numeric_limits<pds::version_t>::max();


    template <class OBJ>
    class fpSet{

        pds::fpFatNodePtr<OBJ> root;
        pds::version_t last_version;
        std::vector<pds::version_t> sizes;

    public:
        fpSet();

        pds::version_t insert(const OBJ& obj, pds::version_t version = default_version);


        pds::version_t insert(OBJ&& obj, pds::version_t version = default_version);


        pds::version_t remove(const OBJ& obj, pds::version_t version = default_version);


        pds::version_t remove(OBJ&& obj, pds::version_t version = default_version);


        bool contains(const OBJ& obj, pds::version_t version = MasterVersion);


        std::vector<OBJ> to_vector(const pds::version_t version = MasterVersion);


        pds::version_t size(pds::version_t version = MasterVersion) const noexcept;


        pds::version_t curr_version() const noexcept;


        void print(pds::version_t version = MasterVersion);

    private:
        template <typename T>
        pds::version_t insert_impl(T&& obj, pds::version_t version);
    };
};


template <class OBJ>
pds::fpSet<OBJ>::fpSet() : root(1), last_version(1), sizes{0, 0} {
}


template <class OBJ>
pds::version_t pds::fpSet<OBJ>::insert(const OBJ& obj, pds::version_t version){

    return insert_impl(obj, version);
}


template <class OBJ>
pds::version_t pds::fpSet<OBJ>::insert(OBJ&& obj, pds::version_t version){

    return insert_impl(std::move(obj), version);
}


template <class OBJ>
template <typename T>
pds::version_t pds::fpSet<OBJ>::insert_impl(T&& obj, pds::version_t version){

    if(version == default_version)
        version = last_version;

    if(version == MasterVersion)
        throw pds::VersionZeroIllegal("Version 0 is not valid for insert");

    PDS_THROW_IF_VERSION_NOT_EXIST("fpSet::insert_impl", version, last_version);

    if(contains(obj, version))
        throw pds::ObjectAlreadyExist(
                "fpSet::insert: Version " + std::to_string(version) + " already contains this object"
```

```
            );

    pds::version_t new_version = last_version + 1;

    pds::fpSetTracker<OBJ> tracker(root, version);

    // Inserting a new version to the tree:
    while(tracker.not_null()){

        tracker[new_version] = *tracker;

        if(obj < tracker.obj()){

            tracker.add_right_map(new_version);
            tracker = tracker.left();
        }
        else{
            tracker.add_left_map(new_version);
            tracker = tracker.right();
        }
    }

    pds::fpSetTracker<OBJ> track_master(root, MasterVersion);

    while(track_master.not_null()){

        if(obj < track_master.obj()){

            track_master = track_master.left();
        }
        else if(track_master.obj() < obj){

            track_master = track_master.right();
        }
        else{
            tracker[new_version] = *track_master;
            tracker.set_left_at(new_version) = nullptr;
            tracker.set_right_at(new_version) = nullptr;
            break;
        }
    }

    if(track_master.null()){

        track_master[MasterVersion] = std::make_shared<pds::fpFatNode<OBJ»(std::forward<T>(obj),
      new_version);

        ++sizes[MasterVersion];
        tracker[new_version] = *track_master;
    }

    // push the size of the new version
    sizes.push_back(sizes[version] + 1);

    return (last_version = new_version);
}


template <class OBJ>
pds::version_t pds::fpSet<OBJ>::remove(const OBJ& obj, pds::version_t version){

    if(version == default_version)
        version = last_version;

    if(version == MasterVersion)
        throw pds::VersionZeroIllegal("Version 0 is not valid for remove");

    PDS_THROW_IF_VERSION_NOT_EXIST("fpSet::remove", version, last_version);

    if(!contains(obj, version))
        throw pds::ObjectNotExist(
            "pds::fpSet::remove: Attempting to remove an object from Version "
            + std::to_string(version) + ". But the object is not exists for this Version"
        );


    pds::version_t new_version = last_version + 1;
    pds::fpSetTracker<OBJ> tracker(root, version);

    // Inserting a new version to the tree:
    while(tracker.not_null()){

        if(obj < tracker.obj()){

            tracker[new_version] = *tracker;
            tracker.add_right_map(new_version);
            tracker = tracker.left();
        }
        else if(tracker.obj() < obj){

            tracker[new_version] = *tracker;
            tracker.add_left_map(new_version);
            tracker = tracker.right();
        }
        else break;
    }

    pds::fpSetTracker<OBJ> to_remove = tracker;

    if(to_remove.left_null() && to_remove.right_null()){

        to_remove[new_version] = nullptr;
    }
    else if(to_remove.left_null() || to_remove.right_null()){

        if(to_remove.left_null())
            tracker = tracker.right();
        else
            tracker = tracker.left();

        to_remove[new_version] = *tracker;
        to_remove.add_left_map(new_version);
        to_remove.add_right_map(new_version);
    }
    else{
        pds::fpSetTracker<OBJ> track_to_leaf = tracker;
        track_to_leaf = track_to_leaf.right();

        if(track_to_leaf.left_null()){

            to_remove[new_version] = *track_to_leaf;
            track_to_leaf.add_left_map(new_version);
            track_to_leaf.add_right_map(new_version);

            tracker = tracker.left();
            to_remove.set_left_at(new_version) = *tracker;
            tracker.add_left_map(new_version);
            tracker.add_right_map(new_version);
        }
        else{

/***************  ILLUSTRATION  *************************

          +++ version +++    |   +++ new_version +++
                             |
      to_remove:[6]          |           [11]
               /   \        ==»         /   \
            [4]     [16]     |       [4]     [16]
                   /         |              /
               [11]          |           [13]
                   \         |
                    [13]     |

***********************************************************/

            while(track_to_leaf.left_not_null()){

                track_to_leaf[new_version] = *track_to_leaf;
                track_to_leaf.add_right_map(new_version);
                track_to_leaf = track_to_leaf.left();
            }
            to_remove[new_version] = *track_to_leaf;

            if(track_to_leaf.right_not_null()){

                track_to_leaf[new_version] = track_to_leaf.get_right();
                track_to_leaf.add_left_map(new_version);

                track_to_leaf = track_to_leaf.right();
                track_to_leaf.add_left_map(new_version);
                track_to_leaf.add_right_map(new_version);
            }
            else{
                track_to_leaf[new_version] = nullptr;
            }

            to_remove.set_right_at(new_version) = tracker.get_right();
            to_remove.set_left_at(new_version) = tracker.get_left();
            tracker = tracker.left();
            tracker.add_left_map(new_version);
            tracker.add_right_map(new_version);
        }
    }

    // push the size of the new version
    sizes.push_back(sizes[version] - 1);

    return (last_version = new_version);
}


template <class OBJ>
pds::version_t pds::fpSet<OBJ>::remove(OBJ&& obj, pds::version_t version){

    return remove(std::as_const(obj), version);
}


template <class OBJ>
bool pds::fpSet<OBJ>::contains(const OBJ& obj, pds::version_t version){

    PDS_THROW_IF_VERSION_NOT_EXIST("fpSet::contains", version, last_version);

    pds::fpSetTracker<OBJ> tracker(root, version);

    while(tracker.not_null()){

        if(obj < tracker.obj()){

            tracker = tracker.left();
        }
        else if(tracker.obj() < obj){

            tracker = tracker.right();
        }
        else return true;
    }
    return false;
}


template <class OBJ>
std::vector<OBJ> pds::fpSet<OBJ>::to_vector(const pds::version_t version) {

    PDS_THROW_IF_VERSION_NOT_EXIST("fpSet::to_vector", version, last_version);

    std::vector<OBJ> obj_vec;

    /*** Inorder Stack Traversal ***/
    std::stack<pds::fpSetTracker<OBJ>> trav_stack;
    pds::fpSetTracker<OBJ> tracker(root, version);

    while(tracker.not_null() || !trav_stack.empty()){

        while(tracker.not_null()){

            trav_stack.push(tracker);
            tracker = tracker.left();
        }
        tracker = trav_stack.top();
        trav_stack.pop();

        obj_vec.push_back(tracker.obj());
        tracker = tracker.right();
    }
    return obj_vec;
}


template <class OBJ>
pds::version_t pds::fpSet<OBJ>::size(pds::version_t version) const noexcept {

    try{
        return sizes.at(version);
    }
    catch(const std::out_of_range&){

        return 0;
    }
}


template <class OBJ>
pds::version_t pds::fpSet<OBJ>::curr_version() const noexcept {

    return last_version;
```

```
}


template <class OBJ>
void pds::fpSet<OBJ>::print(pds::version_t version){

    PDS_THROW_IF_VERSION_NOT_EXIST("pset::print", version, last_version);

    std::vector<OBJ> vec = to_vector(version);

    std::cout « "Version " « version « ": {";
    if(!vec.empty()){
        std::copy(vec.begin(), vec.end() - 1, std::ostream_iterator<OBJ>(std::cout, ", "));
        std::cout « vec.back();
    }
    std::cout « "}" « std::endl;
}


#endif /* FULLY_PERSISTENT_SET_HPP */
```