

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:

pds	??
---------------------	-------	----

Chapter 2

Class Index

2.1 Class List

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

pds::fpSet< OBJ >	Fully persistent set container for sorted, unique objects	??
pds::pSet< OBJ >	Partially persistent set container for sorted unique objects	??

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

include/ fpSet.hpp	Fully persistent set container	??
include/ pSet.hpp	Partially persistent set container	??

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

<i>OBJ</i>	The object type, which must support <code>operator<</code> 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

<i>obj</i>	the object to query for.
<i>version</i>	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

<i>obj</i>	object to insert.
------------	-------------------

Attention

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

Parameters

<i>version</i>	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

<i>obj</i>	object to insert.
<i>version</i>	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

<i>version</i>	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

<i>obj</i>	object to remove.
<i>version</i>	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

<i>obj</i>	object to remove.
<i>version</i>	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

<i>version</i>	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

<i>version</i>	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

<i>OBJ</i>	The object type, which must support <code>operator<</code> 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

<i>OBJ</i>	The object type, which must support <code>operator<</code> 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

<i>obj</i>	the object to query for.
<i>version</i>	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

<i>obj</i>	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

<i>obj</i>	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

<i>version</i>	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

<i>obj</i>	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

<i>obj</i>	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

<i>version</i>	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

<i>version</i>	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.](#)

```

00001
00010 #ifndef FULLY_PERSISTENT_SET_HPP
00011 #define FULLY_PERSISTENT_SET_HPP
00012
00013 #include "internal/fpSetTracker.hpp"
00014
00015 namespace pds{
00016
00017     const pds::version_t default_version = std::numeric_limits<pds::version_t>::max();
00018
00019
00020
00045     template <class OBJ>
00046     class fpSet{
00047
00048         pds::fpFatNodePtr<OBJ> root;
00049         pds::version_t last_version;
00050         std::vector<pds::version_t> sizes;
00051
00052     public:
00060         fpSet();
00061
00091         pds::version_t insert(const OBJ& obj, pds::version_t version = default_version);
00092
00093
00125         pds::version_t insert(OBJ&& obj, pds::version_t version = default_version);
00126
00127
00156         pds::version_t remove(const OBJ& obj, pds::version_t version = default_version);
00157
00158
00189         pds::version_t remove(OBJ&& obj, pds::version_t version = default_version);
00190
00191
00209         bool contains(const OBJ& obj, pds::version_t version = MasterVersion);
00210
00211
00228         std::vector<OBJ> to_vector(const pds::version_t version = MasterVersion);
00229
00230
00242         pds::version_t size(pds::version_t version = MasterVersion) const noexcept;
00243
00244
00252         pds::version_t curr_version() const noexcept;
00253
00254
00270         void print(pds::version_t version = MasterVersion);
00271
00272     private:
00278         template <typename T>
00279         pds::version_t insert_impl(T&& obj, pds::version_t version);
00280     };
00281 };
00282
00283
00284 template <class OBJ>
00285 pds::fpSet<OBJ>::fpSet() : root(1), last_version(1), sizes{0, 0} {
00286 }
00287
00288
00289 template <class OBJ>
00290 pds::version_t pds::fpSet<OBJ>::insert(const OBJ& obj, pds::version_t version){
00291
00292     return insert_impl(obj, version);
00293 }
00294
00295
00296 template <class OBJ>
00297 pds::version_t pds::fpSet<OBJ>::insert(OBJ&& obj, pds::version_t version){
00298
00299     return insert_impl(std::move(obj), version);
00300 }
00301
00302
00303 template <class OBJ>
00304 template <typename T>
00305 pds::version_t pds::fpSet<OBJ>::insert_impl(T&& obj, pds::version_t version){
00306
00307     if(version == default_version)
00308         version = last_version;
00309
00310     if(version == MasterVersion)
00311         throw pds::VersionZeroIllegal("Version 0 is not valid for insert");

```

```

00312
00313     PDS_THROW_IF_VERSION_NOT_EXIST("fpSet::insert_impl", version, last_version);
00314
00315     if(contains(obj, version))
00316         throw pds::ObjectAlreadyExist(
00317             "fpSet::insert: Version " + std::to_string(version) + " already contains this object"
00318         );
00319
00320     pds::version_t new_version = last_version + 1;
00321
00322     pds::fpSetTracker<OBJ> tracker(root, version);
00323
00324     // Inserting a new version to the tree:
00325     while(tracker.not_null()){
00326
00327         tracker[new_version] = *tracker;
00328
00329         if(obj < tracker.obj()){
00330
00331             tracker.add_right_map(new_version);
00332             tracker = tracker.left();
00333         }
00334         else{
00335             tracker.add_left_map(new_version);
00336             tracker = tracker.right();
00337         }
00338     }
00339
00340     pds::fpSetTracker<OBJ> track_master(root, MasterVersion);
00341
00342     while(track_master.not_null()){
00343
00344         if(obj < track_master.obj()){
00345
00346             track_master = track_master.left();
00347         }
00348         else if(track_master.obj() < obj){
00349
00350             track_master = track_master.right();
00351         }
00352         else{
00353             tracker[new_version] = *track_master;
00354             tracker.set_left_at(new_version) = nullptr;
00355             tracker.set_right_at(new_version) = nullptr;
00356             break;
00357         }
00358     }
00359
00360     if(track_master.null()){
00361
00362         track_master[MasterVersion] = std::make_shared<pds::fpFatNode<OBJ>(std::forward<T>(obj),
new_version);
00363
00364         ++sizes[MasterVersion];
00365         tracker[new_version] = *track_master;
00366     }
00367
00368     // push the size of the new version
00369     sizes.push_back(sizes[version] + 1);
00370
00371     return (last_version = new_version);
00372 }
00373
00374
00375 template <class OBJ>
00376 pds::version_t pds::fpSet<OBJ>::remove(const OBJ& obj, pds::version_t version){
00377
00378     if(version == default_version)
00379         version = last_version;
00380
00381     if(version == MasterVersion)
00382         throw pds::VersionZeroIllegal("Version 0 is not valid for remove");
00383
00384     PDS_THROW_IF_VERSION_NOT_EXIST("fpSet::remove", version, last_version);
00385
00386     if(!contains(obj, version))
00387         throw pds::ObjectNotExist(
00388             "pds::fpSet::remove: Attempting to remove an object from Version "
00389             + std::to_string(version) + ". But the object is not exists for this Version"
00390         );
00391
00392
00393     pds::version_t new_version = last_version + 1;
00394     pds::fpSetTracker<OBJ> tracker(root, version);
00395
00396     // Inserting a new version to the tree:
00397     while(tracker.not_null()){

```

```

00398
00399     if(obj < tracker.obj()){
00400
00401         tracker[new_version] = *tracker;
00402         tracker.add_right_map(new_version);
00403         tracker = tracker.left();
00404     }
00405     else if(tracker.obj() < obj){
00406
00407         tracker[new_version] = *tracker;
00408         tracker.add_left_map(new_version);
00409         tracker = tracker.right();
00410     }
00411     else break;
00412 }
00413
00414 pds::fpSetTracker<OBJ> to_remove = tracker;
00415
00416 if(to_remove.left_null() && to_remove.right_null()){
00417
00418     to_remove[new_version] = nullptr;
00419 }
00420 else if(to_remove.left_null() || to_remove.right_null()){
00421
00422     if(to_remove.left_null())
00423         tracker = tracker.right();
00424     else
00425         tracker = tracker.left();
00426
00427     to_remove[new_version] = *tracker;
00428     to_remove.add_left_map(new_version);
00429     to_remove.add_right_map(new_version);
00430 }
00431 else{
00432     pds::fpSetTracker<OBJ> track_to_leaf = tracker;
00433     track_to_leaf = track_to_leaf.right();
00434
00435     if(track_to_leaf.left_null()){
00436
00437         to_remove[new_version] = *track_to_leaf;
00438         track_to_leaf.add_left_map(new_version);
00439         track_to_leaf.add_right_map(new_version);
00440
00441         tracker = tracker.left();
00442         to_remove.set_left_at(new_version) = *tracker;
00443         tracker.add_left_map(new_version);
00444         tracker.add_right_map(new_version);
00445     }
00446     else{
00447
00448 /***** ILLUSTRATION *****/
00449
00450         +++ version +++      |      +++ new_version +++
00451         |                    |      |
00452 to_remove:[6]              |      | [11]
00453         / \                  |      | / \
00454      [4]  \ [16]             |      | [4]  \ [16]
00455         / \                  |      | / \
00456      [11] \ [13]             |      | [13]
00457         |                    |      |
00458         |                    |      |
00459
00460 *****/
00461
00462         while(track_to_leaf.left_not_null()){
00463
00464             track_to_leaf[new_version] = *track_to_leaf;
00465             track_to_leaf.add_right_map(new_version);
00466             track_to_leaf = track_to_leaf.left();
00467         }
00468         to_remove[new_version] = *track_to_leaf;
00469
00470         if(track_to_leaf.right_not_null()){
00471
00472             track_to_leaf[new_version] = track_to_leaf.get_right();
00473             track_to_leaf.add_left_map(new_version);
00474
00475             track_to_leaf = track_to_leaf.right();
00476             track_to_leaf.add_left_map(new_version);
00477             track_to_leaf.add_right_map(new_version);
00478         }
00479         else{
00480             track_to_leaf[new_version] = nullptr;
00481         }
00482
00483         to_remove.set_right_at(new_version) = tracker.get_right();
00484         to_remove.set_left_at(new_version) = tracker.get_left();

```

```

00485         tracker = tracker.left();
00486         tracker.add_left_map(new_version);
00487         tracker.add_right_map(new_version);
00488     }
00489 }
00490
00491 // push the size of the new version
00492 sizes.push_back(sizes[version] - 1);
00493
00494 return (last_version = new_version);
00495 }
00496
00497
00498 template <class OBJ>
00499 pds::version_t pds::fpSet<OBJ>::remove(OBJ&& obj, pds::version_t version){
00500     return remove(std::as_const(obj), version);
00501 }
00502
00503
00504
00505 template <class OBJ>
00506 bool pds::fpSet<OBJ>::contains(const OBJ& obj, pds::version_t version){
00507     PDS_THROW_IF_VERSION_NOT_EXIST("fpSet::contains", version, last_version);
00508     pds::fpSetTracker<OBJ> tracker(root, version);
00509
00510     while(tracker.not_null()){
00511         if(obj < tracker.obj()){
00512             tracker = tracker.left();
00513         }
00514         else if(tracker.obj() < obj){
00515             tracker = tracker.right();
00516         }
00517         else return true;
00518     }
00519     return false;
00520 }
00521
00522
00523 template <class OBJ>
00524 std::vector<OBJ> pds::fpSet<OBJ>::to_vector(const pds::version_t version) {
00525     PDS_THROW_IF_VERSION_NOT_EXIST("fpSet::to_vector", version, last_version);
00526     std::vector<OBJ> obj_vec;
00527
00528     /** Inorder Stack Traversal */
00529     std::stack<pds::fpSetTracker<OBJ> trav_stack;
00530     pds::fpSetTracker<OBJ> tracker(root, version);
00531
00532     while(tracker.not_null() || !trav_stack.empty()){
00533         while(tracker.not_null()){
00534             trav_stack.push(tracker);
00535             tracker = tracker.left();
00536         }
00537         tracker = trav_stack.top();
00538         trav_stack.pop();
00539         obj_vec.push_back(tracker.obj());
00540         tracker = tracker.right();
00541     }
00542     return obj_vec;
00543 }
00544
00545
00546 template <class OBJ>
00547 pds::version_t pds::fpSet<OBJ>::size(pds::version_t version) const noexcept {
00548     try{
00549         return sizes.at(version);
00550     }
00551     catch(const std::out_of_range&){
00552         return 0;
00553     }
00554 }
00555
00556
00557 template <class OBJ>
00558 pds::version_t pds::fpSet<OBJ>::curr_version() const noexcept {
00559

```

```

00572     return last_version;
00573 }
00574
00575
00576 template <class OBJ>
00577 void pds::fpSet<OBJ>::print(pds::version_t version){
00578
00579     PDS_THROW_IF_VERSION_NOT_EXIST("pset::print", version, last_version);
00580
00581     std::vector<OBJ> vec = to_vector(version);
00582
00583     std::cout << "Version " << version << ": {";
00584     if(!vec.empty()){
00585         std::copy(vec.begin(), vec.end() - 1, std::ostream_iterator<OBJ>(std::cout, ", "));
00586         std::cout << vec.back();
00587     }
00588     std::cout << "}" << std::endl;
00589 }
00590
00591
00592 #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.](#)

```

00001
00009 #ifndef PARTIALLY_PERSISTENT_SET_HPP
00010 #define PARTIALLY_PERSISTENT_SET_HPP
00011
00012 #include "internal/pSetTracker.hpp"
00013
00014 namespace pds{
00015
00051     template <class OBJ>
00052     class pSet{
00053
00054         pds::pFatNodePtr<OBJ> root;
00055         pds::version_t last_version;
00056         std::vector<pds::version_t> sizes;
00057
00058     public:
00066         pSet();
00067
00068
00085         pds::version_t insert(const OBJ& obj);
00086
00087
00107         pds::version_t insert(OBJ&& obj);
00108
00109
00125         pds::version_t remove(const OBJ& obj);
00126
00127
00144         pds::version_t remove(OBJ&& obj);
00145
00146
00164         bool contains(const OBJ& obj, pds::version_t version = MasterVersion);
00165
00166
00183         std::vector<OBJ> to_vector(const pds::version_t version = MasterVersion);
00184
00185
00197         std::size_t size(pds::version_t version = MasterVersion) const;
00198
00199
00207         pds::version_t curr_version() const;
00208
00209
00225         void print(pds::version_t version = MasterVersion);
00226
00227     private:
00233         template <typename T>
00234         pds::version_t insert_impl(T&& obj);
00235     };
00236 };
00237
00238
00239 template <class OBJ>
00240 pds::pSet<OBJ>::pSet() : root(1), last_version(1), sizes{0, 0} {
00241 }
00242
00243 template <class OBJ>
00244 pds::version_t pds::pSet<OBJ>::insert(const OBJ& obj){
00245
00246     return insert_impl(obj);
00247 }
00248
00249 template <class OBJ>
00250 pds::version_t pds::pSet<OBJ>::insert(OBJ&& obj){
00251
00252     return insert_impl(std::move(obj));
00253 }
00254
00255 template <class OBJ>
00256 template <typename T>
00257 pds::version_t pds::pSet<OBJ>::insert_impl(T&& obj){
00258
00259     pds::pSetTracker<OBJ> tracker(root);
00260
00261     while(tracker.not_null()){
00262
00263         if(obj < tracker.obj()){
00264
00265             tracker = tracker.left();
00266         }
00267         else if(tracker.obj() < obj){
00268

```

```

00269         tracker = tracker.right();
00270     }
00271     else{
00272         throw pds::ObjectAlreadyExist(
00273             "pSet::insert: Version " + std::to_string(last_version) + " already contains this
object"
00274         );
00275     }
00276 }
00277
00278 pds::version_t new_version = last_version + 1;
00279
00280 pds::pSetTracker<OBJ> track_master(root);
00281
00282 while(track_master.not_null_at(MasterVersion)){
00283     if(obj < track_master.obj_at(MasterVersion)){
00284         track_master = track_master.left_at(MasterVersion);
00285     }
00286     else if(track_master.obj_at(MasterVersion) < obj){
00287         track_master = track_master.right_at(MasterVersion);
00288     }
00289     else{
00290         tracker[new_version] = track_master.at(MasterVersion);
00291         break;
00292     }
00293 }
00294
00295 if(track_master.null_at(MasterVersion)){
00296     track_master[MasterVersion] = std::make_shared<pds::pFatNode<OBJ>(std::forward<T>(obj),
new_version);
00297
00298     ++sizes[MasterVersion];
00299     tracker[new_version] = track_master.at(MasterVersion);
00300 }
00301 else{
00302     tracker.set_track_version(new_version);
00303     if(tracker.left_null() == false){
00304         tracker.set_left(new_version) = nullptr;
00305     }
00306     if(tracker.right_null() == false){
00307         tracker.set_right(new_version) = nullptr;
00308     }
00309 }
00310 sizes.push_back(sizes[last_version] + 1);
00311 return (last_version = new_version);
00312 }
00313
00314 template <class OBJ>
00315 pds::version_t pds::pSet<OBJ>::remove(OBJ&& obj){
00316     return remove(std::as_const(obj));
00317 }
00318
00319 template <class OBJ>
00320 pds::version_t pds::pSet<OBJ>::remove(const OBJ& obj){
00321     pds::pSetTracker<OBJ> tracker(root);
00322     while(tracker.not_null()){
00323         if(obj < tracker.obj()){
00324             tracker = tracker.left();
00325         }
00326         else if(tracker.obj() < obj){
00327             tracker = tracker.right();
00328         }
00329         else break;
00330     }
00331     if(tracker.null())
00332         throw pds::ObjectNotExist(
00333             "pds::pset::remove: Attempting to remove an object but the object is not exists"
00334         );
00335     pds::version_t new_version = last_version + 1;
00336 }

```



```

00354     pds::pSetTracker<OBJ> to_remove = tracker;
00355     pds::pSetTracker<OBJ> track_to_leaf = tracker;
00356
00357     if(to_remove.left_null()){
00358
00359         to_remove[new_version] = tracker.get_right();
00360     }
00361     else if(to_remove.right_null()){
00362
00363         to_remove[new_version] = tracker.get_left();
00364     }
00365     else{
00366         track_to_leaf = track_to_leaf.right();
00367
00368         if(track_to_leaf.left_null()){
00369
00370             to_remove[new_version] = *track_to_leaf;
00371             to_remove.set_track_version(new_version);
00372             to_remove.set_left(new_version) = tracker.get_left();
00373         }
00374         else{
00375             while(!track_to_leaf.left_null()){
00376
00377                 track_to_leaf = track_to_leaf.left();
00378             }
00379             track_to_leaf[new_version] = track_to_leaf.get_right();
00380
00381             to_remove[new_version] = *track_to_leaf;
00382             to_remove.set_track_version(new_version);
00383             to_remove.set_left(new_version) = tracker.get_left();
00384             to_remove.set_right(new_version) = tracker.get_right();
00385         }
00386     }
00387     // push the size of the new version
00388     sizes.push_back(sizes[last_version] - 1);
00389
00390     return (last_version = new_version);
00391 }
00392
00393 template <class OBJ>
00394 bool pds::pSet<OBJ>::contains(const OBJ& obj, pds::version_t version) {
00395
00396     PDS_THROW_IF_VERSION_NOT_EXIST("pSet::contains", version, last_version);
00397
00398     pds::pSetTracker<OBJ> tracker(root);
00399
00400     while(tracker.not_null_at(version)){
00401
00402         if(obj < tracker.obj_at(version)){
00403
00404             tracker = tracker.left_at(version);
00405         }
00406         else if(tracker.obj_at(version) < obj){
00407
00408             tracker = tracker.right_at(version);
00409         }
00410         else return true;
00411     }
00412     return false;
00413 }
00414
00415 template <class OBJ>
00416 std::vector<OBJ> pds::pSet<OBJ>::to_vector(const pds::version_t version) {
00417
00418     PDS_THROW_IF_VERSION_NOT_EXIST("pSet::to_vector", version, last_version);
00419
00420     std::vector<OBJ> obj_vec;
00421
00422     /** Inorder Stack Traversal */
00423     std::stack<pds::pSetTracker<OBJ> > trav_stack;
00424     pds::pSetTracker<OBJ> tracker(root);
00425
00426     while(tracker.not_null_at(version) || !trav_stack.empty()){
00427
00428         while(tracker.not_null_at(version)){
00429
00430             trav_stack.push(tracker);
00431             tracker = tracker.left_at(version);
00432         }
00433         tracker = trav_stack.top();
00434         trav_stack.pop();
00435
00436         obj_vec.push_back(tracker.obj_at(version));
00437         tracker = tracker.right_at(version);
00438     }
00439     return obj_vec;
00440 }

```

```
00441
00442 template <class OBJ>
00443 std::size_t pds::pSet<OBJ>::size(pds::version_t version) const {
00444
00445     try{
00446         return sizes.at(version);
00447     }
00448     catch(const std::out_of_range&){
00449
00450         return 0;
00451     }
00452 }
00453
00454 template <class OBJ>
00455 pds::version_t pds::pSet<OBJ>::curr_version() const {
00456     return last_version;
00457 }
00458
00459 template <class OBJ>
00460 void pds::pSet<OBJ>::print(pds::version_t version){
00461
00462     PDS_THROW_IF_VERSION_NOT_EXIST("pSet::print", version, last_version);
00463
00464     std::vector<OBJ> vec = to_vector(version);
00465
00466     std::cout << "Version " << version << ": {";
00467     if(!vec.empty()){
00468         std::copy(vec.begin(), vec.end() - 1, std::ostream_iterator<OBJ>(std::cout, ", "));
00469         std::cout << vec.back();
00470     }
00471     std::cout << "}" << std::endl;
00472 }
00473
00474
00475
00476 #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 */
```

