B-Tree Framework Programmer Guide

by Andreas Steffens

Version: 0.08

Introduction

This document is intended for programmers who want to use abstract container types as they are provided by the b-tree framework described in this documentation. The reader will find information on how to interface with said framework or more specifically with the classes therein and how to extend those in terms of aiding an application to solve a problem. Furthermore, parts of this document are dedicated to depict why certain design decisions have been made and what layers with which intended purpose exist. Also, sections and paragraphs with information what this framework is incapable of and what potential show stoppers a programmer want to look out for are present. Finally, a full API description of all data class' constructors, methods and operators is appended for the programmer's benefit.

The documentation assumes that the reader already has an understanding of what b-trees are and how they work in terms of data processing. Also what b-tree's are useful for won't be addressed here. This means that, the examples, which can be found in this document, are used as a display what parts of this framework might be employed by a programmer.

Table of Contents

History Revision

Version	Remarks	Date
0.01	initial revision	24/07/2015
0.02	data layer and data layer API sections updated after design change	24/08/2015
0.03	data layer API sections updated – get_pooledData () updated – get_node_offset () created	04/09/2015
0.04a	framework model updated – Class Hierarchy and all interface sections updated – iterator model updated	06/04/2016
0.04b	template parameters updated – STL compatible type definitions (size_type, value_type, etc.) - data layer dimension types are now provided by the data layer properties	04/07/2016
0.05	system that governs size requests to the data layer is explained in more detail	12/10/2016
0.06	for associative containers methods emplace (), emplace_hint () and insert (hint), an API description has been added for linear container methods emplace () and emplace_back (), an API description has been added	04/11/2016
0.07	description for scalability method find_nextKey () has been added for associative containers method equal_range () API description has been added data layer hierarchy abstraction diagram and more detailed explanations added	18/11/2016
0.08	move constructor and move assignment operator descriptions for any container and iterator type have been added	/12/2016

Quick Start Guide

The quick start section gives a brief overview what the Specific Data Classes are, which is the most likely starting point for a new application that wants to involve the b-tree framework. Also, it displays what they are capable of. This is aimed at programmers who want to use the b-tree framework, rather than modify existing behaviour or extend functionality via derivation. If a more detailed documentation is required, then please refer to the Programmer Guide section.

Overview

The b-tree framework consists of a number of layers, which allow for several levels of abstraction. This section however, will only cover the Specific Data Classes, which are the least abstract types. Those classes are the most likely types to be used by an application and are separated into associative and linear types. The associative types always have a key value associated with every data entry that dictates where a data entry is to be stored relative to other data entries, by employing a sorting criteria. The linear type acts like an array, while at its back end is a b-tree used with linear offsets to govern the location of distinct data entries, instead of a key.

This documentation assumes that the reader is familiar with the associative and linear data container types as they are provided by the Standard Template Libraries (STL).

Associative Classes Overview

The A. are sub-divided into STL equivalent containers and the CBTreeKeySort class. The STL Equivalent Containers are, for the most part, compatible with their STL counterparts in terms of their API. Whereas the template parameters between the two type sets are very different (see: Template Parameter Overview). Furthermore, the CBTreeKeySort class is similar to ::std::multimap, but doesn't use ::std::pair<key type, map type> as its value type, with the key and map type being defined via template parameters. Instead its value and key type are defined via template parameters, while the key type is extracted from the value type using a method called extract_key ().

Overview of STL Equivalent Containers

The associative containers that are equivalent to the STL containers are listed below.

b-tree framework	**STL**
CBTreeMultiMap	::std::multimap
CBTreeMap	::std::map
CBTreeMultiSet	::std::multiset
CBTreeSet	::std::set

As stated above, the b-tree framework versions are not template parameter compatible (see Template Parameter Overview) and not fully API compatible. Some interface methods cannot be supported by design, such as get_allocator (), since the data layers between STL and this framework are incompatible. Methods like emplace () or emplace_hint () are not supported, because they haven't been implemented yet. Other than that, the b-tree framework containers allow for abstraction, which the STL containers are not designed for. This is not limited to containers, which are equivalent to an STL container, but involves any container type described in this documentation.

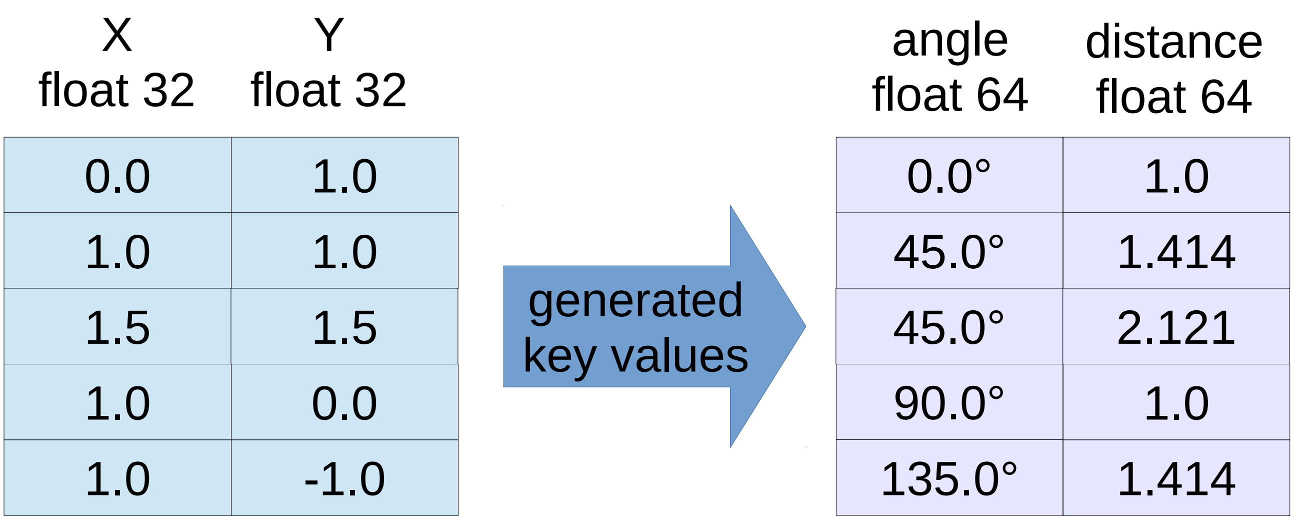
CBTreeKeySort Overview

The CBTreeKeySort class is an ordered container type that sorts data entries by employing a sorting criteria and key values as its operands. The key value is either present in or can be determined from each data set. By default CBTreeKeySort treats each data entry also as its associated key value. The class uses a binary comparison as its default sorting criteria, to determine which data entry is put where.

At a first glance, the range of functionality seems to be already covered by the STL equivalent classes. However, CBTreeKeySort is a bit more versatile, since unlike ::std::[multi]map, the key and the map value are not separated and as a result data entries are being seen as a whole in terms of their key and associated payload data. This means that, in order to sort data entries the key values must be extracted from the respective data entries prior to sorting. This allows for a higher degree of abstraction as oppose to the existing STL containers, as the extraction is done by a virtual method called extract_key (). For set and multiset an extraction is not needed, because the value type is the same as the key type and for map and multimap the extraction would only involve the first element of the value type. That higher degree of abstraction allows for other options, too. For instance, it offers the ability to use any arbitrary part of a data set as its key value, by having the key extraction return a reference to the entire data set, whereas the sorting criteria itself then selects the effective parts in order to create a result. Please see the example below, where key (part 1) has priority over key (part 2):

key (part 1)	data (part 1)	key (part 2)	data (part 2)
A	123456	X	654321
A	987654	Y	456789
C	012345	Z	000000
D	387609	A	100000

Also, it allows for memory-time trade-offs, in cases where the key type is larger than the value type and the key value can be calculated from the data entry value. The example below displays the data entries on the left and their generated key values on the right.



X float 32	Y float 32
0.0	1.0
1.0	1.0
1.5	1.5
1.0	0.0
1.0	-1.0

angle float 64	distance float 64
0.0°	1.0
45.0°	1.414
45.0°	2.121
90.0°	1.0
135.0°	1.414

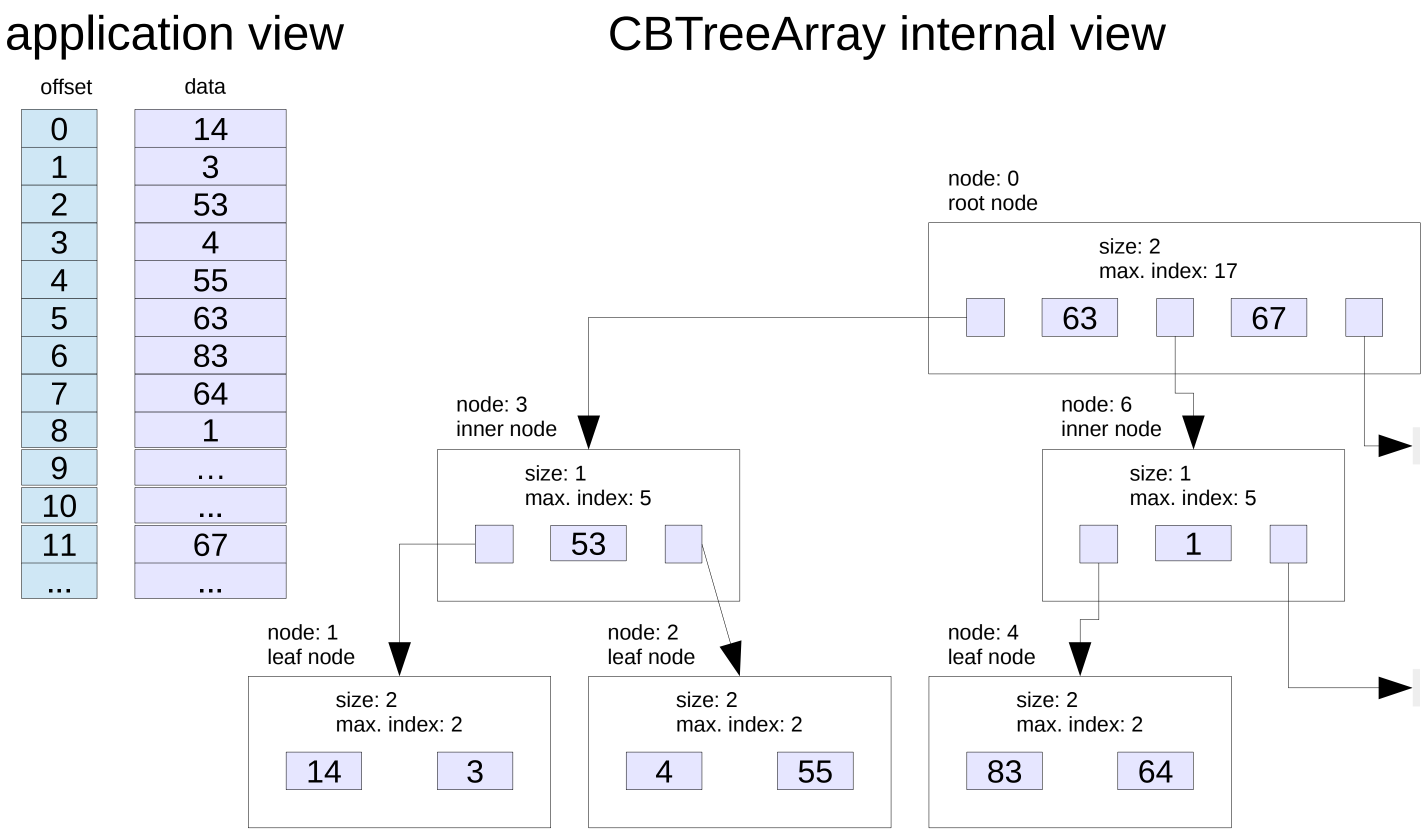
The generated key values are dictating the sorting of the data entries by the angle value in ascending order, as the top priority, and then by the distance value in ascending order. As it can be seen, data entries would be three times in size, if the key values wouldn't be generated on the fly. However, to generate the key values, extra computational time is required. This is interesting to investigate in case a high latency data layer, such as a file, is in use, since computing the key values would have an insignificant impact if data layer accesses are already taking a lot of time and smaller data sets would result in less data layer accesses to begin with.

The CBTreeKeySort also supports data entries sharing the same key (key set), but with that two problems need to be kept in mind. Firstly, it is not guaranteed in what order the data sets will be presented to the application, once they have been assigned from one instance to another by using the assignment operator (operator=) or its copy-constructor (CBTreeKeySort Constructor). Secondly, if any key set gets too large (key set size > maximum node size), then the result will be performance issues in terms of access time, since the key sort data class is not designed to handle large key sets.

CBTreeArray Overview

The CBTreeArray class acts like a dynamic linear list, such as std::vector or std::list. Both of those classes have their strengths and weaknesses. For instance: If a vector gets very large it takes a very long time to insert data at the beginning. It is the same with very long lists, when the application wants the seek for an entry in the middle. Large instances of CBTreeArray are coming without the drawback of long processing periods in the afore mentioned scenarios, since internal mechanisms are those of a b-tree, except that instead of a key type linear offsets are used to control where data is inserted, removed or obtained from.

The example below shows parts of a b-tree state on the right and how the b-tree's data is presented to the application on the left. As it can be seen, the data stored by the b-tree doesn't contain any key values and instead uses the sub-tree sizes displayed by “max. index” per node to decide what data entry access or which sub-tree to recursively to trace into, when dealing with linear addresses.



offset	data
0	14
1	3
2	53
3	4
4	55
5	63
6	83
7	64
8	1
9	…
10	...
11	67
...	...

For each node, size indicates how many data entries are stored within it and also allows to determine the number of sub-tree's each node is referring to, which is size plus one. Furthermore, the maximum index displays the number of data entries plus all sub-node's maximum indexes combined. This means that, if size is equal to maximum index, then it is a leaf node, which cannot have sub-nodes. Also, the root node's maximum index displays the total number of data entries addressable by the entire b-tree.

When an access takes place, then sub-node references and data entries are searched in an alternate order from left to right, beginning with the sub-node, which contains all lower order data entries. Key values aren't present and walking through every sub-tree, when accessing data, would be sub-optimal in terms of performance. To work around that problem, the maximum index property of sub-nodes potentially accessed is used to tell if a sub-tree can be skipped or needs to be traced into. In the example above, if data entries 0 to 4 are being accessed, then by using the maximum index of node 3 the calling method can immediately tell that node 3 needs to be accessed and has to go from their, otherwise node 3 is skipped entirely and it is determined whether the first data entry in the root is addressed or the next sub-node needs to be traced into and so on and so forth. In other words, the offset value of an access is used as a key value in order to navigate within the tree structure, while the procedures on how to insert, access and remove individual data entries still apply even with an offset as a key.

Template Parameter Overview

This section explains what template parameters have to be set in order to instantiate any of the Specific Data Classes and also provides a brief overview of what the parameter is intended for.

Parameter	Default	Class	Description
_t_data	**has to be set!**	CBTreeArray CBTreeKeySort	application data type
_t_key	_t_data	CBTreeKeySort	sort and search information type
_t_keytype	**has to be set!**	CBTreeSet CBTreeMultiSet CBTreeMap CBTreeMultiMap	
_t_maptype	**has to be set!**	CBTreeMap CBTreeMultiMap	
_t_datalayerproperties	CBTreeIOpropertiesRAM	all	data layer set up type

Data Type (_t_data): This parameter defines the type of an entire data entry used by the classes CBTreeArray and CBTreeKeySort, which also the type inserted or returned when using the appropriate interface methods.

Key Type (_t_key): This parameter defines the type, which, when instantiated, is acting as the key value for its associated data entry and is the return type of extract_key (), that is then used as an operand for the sorting criteria defined by comp ().

_t_keytype: This parameter defines the key type used by the Associative Containers (CBTree[Multi]Map|Set), which is the initial element of the pair type (::std::pair<_t_keytype, _t_maptype>) used by CBTreeMap and CBTreeMultiMap as its value type. Also, it defines the value type of CBTreeSet and CBTreeMultiSet, which is the same as the key type for those two container types.

_t_maptype: This parameter defines the associated map type for the container types CBTreeMap and CBTreeMultiMap and is the final element of the pair type (::std::pair<_t_keytype, _t_maptype>) used by those two containers.

Data Layer Properties (_t_datalayerproperties): This parameter defines the data layer properties type, which allows to set the data layer dimension types and provides the data layer type as type definitions (SFINEA - substitution failure is not an error). see: Data Layer Dimension Types

Data Layer Dimension Types

This section provides a brief overview what template parameters can be set when defining a Data Layer Property type (Data Layer Properties (_t_datalayerproperties)). See the list below:

Template Parameter	Type Definition	Default	Description
_t_sizetype	size_type	uint64_t	application address type
_t_nodeiter	node_iter_type	uint64_t	node address type
_t_subnodeiter	sub_node_iter_type	uint32_t	sub-position address type
_t_address	address_type	uint64_t	absolute address type
_t_offset	offset_type	uint32_t	relative address type

Size Type (_t_sizetype): This parameter defines what type is to be used by an application address individual data entries stored within the data container.

Node Iterator Type (_t_nodeiter): This parameter defines what type is used by the container instance to address individual nodes within the b-tree structure.

Sub-Node Iterator Type (_t_subnodeiter): This parameter defines what type is used by the container instance to address individual data entries with a node.

Absolute Address Type (_t_addresstype): This parameter defines what type is used by the data layer to calculate an absolute byte location.

Relative Address Type (_t_offsettype): This parameter defines what type is used by the data layer to calculate a relative byte location.

Differences Between The STL Container and B-Tree Framework Template Parameters

This section briefly describes what the differences between the STL container's and the Specific Data Classes' (Specific Data Classes) template parameters are. Since different containers use a different sets of template parameter, this section is divided into two parts. The initial part covers parameters that apply to all container types, while the second part covers parameters, which are specific to a sub-set of container types.

Differences that apply to all container types:

- The data containers of the b-tree framework and the STL types have to store data at some point. This is what the Data Layer Type Definition (data_layer_type) for this framework and the allocators for the STL containers are employed for. While the framework employs the Data Layer Properties (_t_datalayerproperties) to determine the Data Layer Type Definition (data_layer_type) using `typename _t_datalayerproperties::data_layer_type`, the STL types use allocators directly set via their template parameters. These two data storage types are incompatible, since the framework Data Layer Type Definition (data_layer_type) isn't aware of the type that is to be stored and sees every pool (Pools) as a byte array, whereas STL allocators are type aware. As a result, anything that involves allocators on the API level cannot be supported by this framework.
- The b-tree framework containers need to have their size_type set via one of the Data Layer Dimension Types when defining Specific Data Classes or has to be set directly when defining Abstract Container Interface (CBTreeIf) type, whereas none of the STL containers has that option. Also, the size_type parameter needs to be compatible if Specific Data Classes are abstracted to an Abstract Container Interface (CBTreeIf) type.

Differences that apply to all associative container types:

- All associative containers employ an order operator of some sort to determine in what order individual data entries are stored. This is handled very differently between the STL containers and the b-tree framework containers. While the STL containers have a template parameter, defining a type that has to provide a comparison method, the b-tree framework containers use a virtual method called comp () in order to allow for those container types to be abstracted.

Quick Programmer Guide

This section will discuss what is to be done in order to compile and run a program involving parts of this b-tree framework. The initial step is to create one or more Application Classes by deriving each of those from one of the Specific Data Classes as needed. It is very likely that these classes will end up almost empty at first, but creating Application Classes is strongly recommended for two reasons. Firstly, it allows for quicker additions and modifications in the development process, since it is unlikely that any of the given Specific Data Classes will be sufficient as they are. Secondly, even if the given Specific Data Classes are sufficient, the newly created Application Classes must be named, which makes it more obvious what the class' use is intended for. Once the Application Classes have been defined, they may act as base classes for more specialised application container classes. Also, Application Classes that have been derived from CBTreeKeySort may want to provide their own versions of comp () and extract_key (), in case their default behaviour is not sufficient.

This framework also provides abstract types such as the Abstract Interface Classes and the Abstract Container Interface (CBTreeIf). These enable programmers to abstract container classes, while using instances that may employ different template parameters or use varied sorting criteria. However, using varied template parameters by an abstract type has limitations though. The template parameters Data Type (_t_data), Size Type (_t_sizetype) and for associative container types Key Type (_t_key) have to be the same, since those parameters have to be set when any of the abstract types is declared.

Includes Overview

The I. section briefly explains what files need to be included in order to use different parts of the b-tree framework. First of all, most compilers cannot process template base code without having the definition and the declaration in the same place. This means that, either any template base declaration needs to be in the respective header file where also its definition can be found or any source file, which contains template declarations needs to be included once the associated definitions have been included. To avoid having the programmers to worry about when to include what source file along side with an include, any header file that contains template definitions also includes its respective source file, so that b-tree framework source file includes don't need to appear in the application's source code.

Which data class header file needs to be included, depends on what the application classes need to be derived from. Please see the list below:

- btreearray.h
  - data class: CBTreeArray

    - interface class: CBTreeArrayIf

- btreekeysort.h
    - data class: CBTreeKeySort
    - interface class: CBTreeAssociativeIf

- ./associative/btreemultimap.h
    - data class: CBTreeMultiMap
    - interface class: CBTreeAssociativeIf

- ./associative/btreemap.h
    - data class: CBTreeMap
    - interface class: CBTreeAssociativeIf

- ./associative/btreemultiset.h
    - data class: CBTreeMultiSet
    - interface class: CBTreeAssociativeIf

- ./associative/btreeset.h
    - data class: CBTreeSet
    - interface class: CBTreeAssociativeIf

- btreeif.h
    - interface class: CBTreeIf

Furthermore, at least one data layer class needs to be included. Please see the list below:

- btreeioram.h
    - data layer type: RAM
    - includes: btreeramioprop.h containing RAM data layer property class

- btreeiofile.h
  - data layer type: file
  - includes: btreefileioprop.h containing file data layer property class

A more detailed list of all target data layers can be found here: Target Data Layer Types and Their Property Types

Source File List

The b-tree framework consists of a number of files listed below. That list also has brief explanations what each files contains.

file name	file content
btree.cpp btree.h	CBTreeBase declaration CBTreeBase definition These files contain the b-tree base class required by any b-tree data class.
btreeaux.cpp btreeaux.h	CBTreeSuper declaration CBTreeSuper definition These files contain the b-tree super and interface classes required by the b-tree base data class.
btreebasedefaults.cpp btreebasedefaults.h	CBTreeBaseDefaults declaration CBTreeBaseDefaults definition These files contain the default code for the some of the scalability and iterator methods, required by the Abstract or Specific Data Classes.
btreebaseif.cpp btreebaseif.h	CBTreeBaseIf declaration CBTreeBaseIf definition These files contain the abstract class' code for the scalability methods, which is part of the base stack.
btreedefaults.cpp btreedefaults.h	CBTreeDefaults declaration CBTreeDefaults definition These files contain the class providing the

file name	file content
	default code for the abstract container interface.
btreeif.cpp btreeif.h	CBTreeIf declaration CBTreeIf definition These files contain the abstract container interface class required by the abstract interface classes and the base stack.
btreeassociative.cpp btreeassociative.h	CBTreeAssociative declaration CBTreeAssociative definition These files contain the generic code applicable for any associative container type within this framework.
btreeassociativebase.cpp btreeassociativebase.h	CBTreeAssociativeBase declaration CBTreeAssociativeBase definition These files contain the generic code applicable for any STL equivalent container type within this framework.
btreeiter.cpp btreeiter.h	CBTreeIterator, CBTreeConstIterator, CBTreeReverseIterator and CBTreeConstReverseIterator declarations and definitions These files contain the b-tree framework's iterator code.
btreecommon.h	This file contains any commonly used definition.
btreearray.cpp btreearray.h	CBTreeArray declaration CBTreeArray definition These files contain the array data class, which also is a data class. If an application needs to define an application specific array class, then btreearray.h has to be included and said class needs to inherit from CBTreeArray.
btreekeysort.cpp btreekeysort.h	CBTreeKeySort declaration CBTreeKeySort definition These files contain the key sort class, which is a b-tree data class. If an application needs to define an application specific key sort class, then btreekeysort.h has to be included and said class needs to inherit from CBTreeKeySort.
btreemultimap.cpp btreemultimap.h	CBTreeMultiMap declaration CBTreeMultiMap definition

file name	file content
	These file contain the code for the STL multimap container equivalent class.
btreemap.cpp btreemap.h	CBTreeMap declaration CBTreeMap definition These file contain the code for the STL map container equivalent class.
btreemultiset.cpp btreemultiset.h	CBTreeMultiSet declaration CBTreeMultiSet definition These file contain the code for the STL multiset container equivalent class.
btreeset.cpp btreeset.h	CBTreeSet declaration CBTreeSet definition These file contain the code for the STL set conatiner equivalent class.
btreeio.cpp btreeio.h	CBTreeIO declaration CBTreeIO definition These files contain the base data layer class required by any data layer class.
btreeiolinear.cpp btreeiolinear.h	CBTreeLinearIO declaration CBTreeLinearIO definition These files contain the base linear data layer class required by any linear data layer class.
btreeioblock.cpp btreeioblock.h	CBTreeBlockIO declaration CBTreeBlockIO definition These files contain the base block data layer class required by any block data layer class.
btreeioram.cpp btreeioram.h	CBTreeRAMIO declaration CBTreeRAMIO definition These files contain the RAM data layer class, which is a linear data layer class. The file btreeioram.h needs to included if an application wants a b-tree data class or a class derived from that, to make use of the RAM data layer.
btreeiofile.cpp btreeiofile.h	CBTreeFileIO declaration CBTreeFileIO definition These files contain the file data layer class, which is a block data layer class. The file btreeiofile.h needs to included if an application wants a b-tree data class or a class derived from

file name	file content
	that, to make use of the file data layer.
btreeioprop.cpp btreeioprop.h	CBTreeIOproperties declaration CBTreeIOproperties definition These files contain the base data layer properties class, required by any data layer properties class.
btreeramioprop.cpp btreeramioprop.h	CBTreeIOpropertiesRAM declaration CBTreeIOpropertiesRAM definition These files contain the RAM data layer properties class, required by the RAM data layer.
btreefileioprop.cpp btreefileioprop.h	CBTreeIOpropertiesFile declaration CBTreeIOpropertiesFile definition These files contain the file data layer properties class, required by the file data layer.

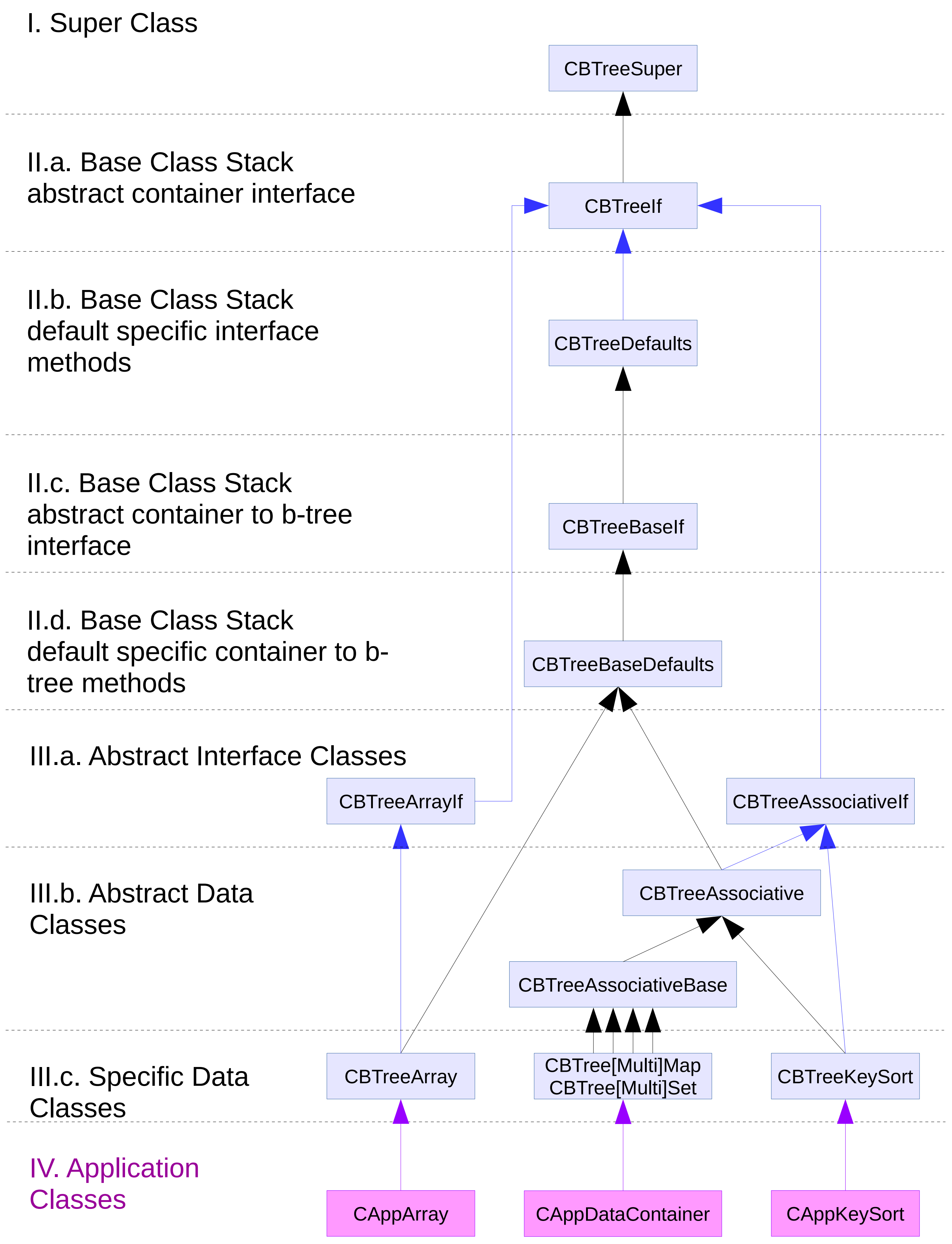
Programmer Guide

The P. section contains everything to know from simply using to fully exploiting the existing b-tree framework. This means, there will be a remark at the beginning of every sub-section, telling the reader whom this part is intended for.

Class Hierarchy

This section will explain the layers within b-tree framework in more detail than the Quick Programmer Guide section. For programmers that want to create an application class by inheriting from one of the existing data classes, only section Specific Data Classes is relevant. If the creation of a new data class type is required, then this section must be read and understood in its entirety.

The diagram below shows an overview of the framework layers and the classes therein. Also it displays, which class inherits from where and if the inheritance is virtual.


I. Super Class
CBTreeSuper
II.a. Base Class Stack abstract container interface
CBTreeIf
II.b. Base Class Stack default specific interface methods
CBTreeDefaults
II.c. Base Class Stack abstract container to b-tree interface
CBTreeBaseIf
II.d. Base Class Stack default specific container to b-tree methods
CBTreeBaseDefaults
III.a. Abstract Interface Classes
CBTreeArrayIf
CBTreeAssociativeIf
III.b. Abstract Data Classes
CBTreeAssociative
CBTreeAssociativeBase
III.c. Specific Data Classes
CBTreeArray
CBTree[Multi]Map
CBTree[Multi]Set
CBTreeKeySort
IV. Application Classes
CAppArray
CAppDataContainer
CAppKeySort

The above diagram shows everything that is part of the framework as black and blue. The purple segments at the bottom, however, are not part of the b-tree framework and their purpose is to display what an application class has to be derived from in order to use the framework. Furthermore, all arrows in the diagram are always connecting two classes going from one class, that is higher in the hierarchy, pointing to a class or classes it is being derived from. While the derivations shown in black (and purple) are regular, those displayed in blue are virtual. Virtual inheritance has been employed to avoid presenting the compiler with ambiguous versions of virtual methods, when abstract interface classes are used by the application.

Since the Specific Data Classes have to be used in any event and every remaining class is either very specific or has a lesser likelihood to be employed by an application, the following sections will explain the above diagram's layers in reverse order.

Specific Data Classes

The S. are used by an application as the most basic layer to derive application classes from, which then form the actual container types. What the S. provide is an API (see: API Reference Guide) allowing the application to interact with the underlying container. If the set or a sub-set of the existing S. are good enough for an application to implement its solution, then only the respective sub-sections CBTreeArray, CBTreeKeySort and Associative Containers (CBTree[Multi]Map|Set) are required to be understood. This is the case, if the default Scalability Methods, Internal Position Template Parameter (_ti_pos) and for CBTreeKeySort and the Associative Containers (CBTree[Multi]Map|Set) the existing sorting criteria (see: comp ()) are meeting all requirements. If that is not the case, then the next paragraph and all following sub-sections say what needs to be done in order to create a new S. Furthermore, the sub-section Abstract Interface Classes is interesting for applications that are created with the intention to abstract data classes, which have partly different template parameters per instance, which in turn also allows for application classes to be abstracted. Every remaining sub-section within the Class Hierarchy section has to be understood if a new data class type (Specific Data Classes) has to be created. This is also advantageous to know if a new or existing application class' behaviour needs to modify behaviour defined within the Base Class Stack.

The following part of this section explains what needs to be done to create a new S. First of all the class needs to be derived from CBTreeBaseDefaults (Specific Default Container to b-tree Methods (CBTreeBaseDefaults)):

- either the new S. has to be derived from CBTreeBaseDefaults (Specific Default Container to b-tree Methods (CBTreeBaseDefaults)) directly
- or the new S. is derived from an existing Abstract Data Class
- or new Abstract Data Class(es) need(s) to be implemented, which is derived from CBTreeBaseDefaults (Specific Default Container to b-tree Methods (CBTreeBaseDefaults)), while the new S. is derived from one of the Abstract Data Classes

The last point is only required if a number of new S. will be created and they share functionality.

The next step is to decide if a new Internal Position Template Parameter (_ti_pos) type needs to be created to be used by CBTreeBaseDefaults (Specific Default Container to b-tree Methods (CBTreeBaseDefaults)). The currently existing internal position types are:

- CBTreeArray Position Template Parameter (CBTreeArrayPos <_t_sizetype>), which is capable to display an absolute or relative position within the a container
- CBTreeKeySort Position Template Parameter (CBTreeKeySortPos <_t_sizetype, _t_key>), which presents an absolute position by referring to a key set plus a linear offset (also called instance) within said key set

If none of the above position types is sufficient, then a new position type needs to be created. Also, as a result either one of the newly developed S. or if present one of the Abstract Data Classes needs to provide its version of the Virtual Base Class Methods. These methods are required to translate between the b-tree structure and the data class' linear addressing model of data items. Also, S. may register additional pools, which has to happen during the construction. Pools describe what data has to be stored in which quantities per node. To determine if additional pools are required and how to register those, please see section Pools (Technical Reference).

The final step is optional. A new abstract interface class (Abstract Interface Classes) may be created, if that is desired to aid the application's solution. That new class has to be virtual derived from CBTreeIf (Abstract Container Interface (CBTreeIf)). This can be omitted if one of the existing Abstract Interface Classes is already sufficient. In any event, Specific Data Classes and Abstract Data Classes have to be virtual derived from any of the Abstract Interface Classes, otherwise the compiler would face a so called diamond inheritance, which would be ambiguous and the code would not compile.

CBTreeArray

The C. class is a type that is presenting it self as a linear array to a calling application and has an interface that is, for the most part, equal to the STL classes ::std::vector and ::std::list. Usually linear arrays and link lists have drawbacks in terms of performance, when it comes to container instances with an ample number of data sets. For instance, inserting data at the beginning of an array over and over again will get progressively slower. An application would face the same slow down, if a link list would be repeatedly and randomly accessed. The array container class discussed here doesn't have these disadvantages, since internally a b-tree is employed to executed the actual operations. As oppose to a regular b-tree, where the location of a data set is governed by a sorting criteria, here locations are determined by linear positions set by the calling application. For this to work with the b-tree framework (namely Specific Default Container to b-tree Methods (CBTreeBaseDefaults)), a number of the CBTreeBaseDefaults class' properties need to be addressed. Firstly, when the C. type inherits from CBTreeBaseDefaults, the Internal Position Template Parameter (_ti_pos) type is set to CBTreeArray Position Template Parameter (CBTreeArrayPos <_t_sizetype>). This type is then used by the Scalability Methods as a parameter type. The C. class has to provide its own version of said methods, which then allow

CBTreeBaseDefaults to operate inside a b-tree structure based on linear addresses. Secondly, this means that, any linear address coming from the application needs to be converted to the array position type (CBTreeArrayPos) displayed by the CBTreeArray Position Template Parameter (CBTreeArrayPos <_t_sizetype>).

As it was initially said, the C. class is aiming to be interface compatible with the STL ::std::vector and :std::list classes, as it supports iterators, which is reflected by its API (see: CBTreeArray API). However, some methods, such as reserve () and shrink_to_fit (), are missing, since the base class is doing allocations and deallocations automatically. The methods data () and get_allocator () are missing by design. The method get_allocator () needs an allocator type template parameter, which this b-tree framework doesn't support. The data back end works different as oppose to ::std::vector and is explained in the section Data Layer. The method data () cannot be supported, because the data is not present as a linear array in memory. If an access like that is required, then one option is to use Iterators and another option is to use the method serialize (). The latter allows to retrieve data in chunks and puts copies in a linear array. The methods emplace () and emplace_back () aren't supported yet, since their implementation is outstanding.

CBTreeKeySort

The C. class is an ordered data container type, allowing for a customisable sorting criteria. As oppose to ::std::multimap, which divides each data set into a key type and a map type, this class sees the every data set as a whole and employs an extraction method to convert to or extract from a data set its respective key value. This translation is done by a method called extract_key (), which is an arbiter selecting between a cast from data type to key type and an application class' version of extract_key (). How said selection is working exactly, is explained in section extract_key () addendum. In any event, the resulting key values are then used by a sorting criteria method called comp (). By default, this method is also an arbiter and it decides if the input key values are compared using an arithmetic operation or a binary comparison. The compare selection is explained in section Default comp () addendum. In case an application class needs to use a different sorting criteria, then it needs to provide its own version of comp (), since that method is virtual.

The C. class has an API similar to the STL ::std::multimap interface, but some differences are present. While multimap uses ::std::pair<const key type, map type> as its value type, to associate a key value to a map value, C. extracts a key value from each data entry, as explained above. This has repercussions in terms of accessing a C. container instance using read / write iterators. If, for instance, multimap is being looked at, only the map part of its pair type can be modified, since the key part has a const qualifier and is therefore read-only. Hence, changing a data entry via an iterator access would be harmless, because there is no potential to corrupt the entry order by reading, modifying and the feeding back a data entry. As for the C. container type though, the situation is very different. This container type cannot guarantee that a data entry being read, modified and then written back would still be in order, since its key value is not forced to be read-only. As a result read / write iterator types obtained from this container type work differently. If a data set is read and modified, then once it gets written back, the input data entry's key value is extracted as well as the key value of the existing data entry as it is stored in the container. If both key values are determined to be the same, the input data entry replaces the existing data entry, otherwise the existing data entry is removed, whereas subsequently the modified data entry gets inserted, while the sorting criteria is employed. This creates a lack of performance, as every time a data entry is written via a read / write iterator at least additional checks have to be executed. Also, if the modified data entry's key value is different, and as a result the above described remove-insert-process has to take place, the iterator ends up referring to one of the adjacent data entries, since iterators operate based on linear addresses. This means that, once the iterator's evaluation has completed, if the modified entry was inserted past the location the iterator is referring to, then the iterator refers to the next data entry without being moved. Otherwise, it is referring to the previous data entry without being moved. It is therefore strongly recommended to use read-only iterators with the C. container type, as they are provided by the methods cbegin (), cend (), crbegin () and crend (). A description of these methods can be found in section CBTreeKeySort API.

Associative Containers (CBTree[Multi]Map|Set)

The A. of this framework are equivalent to a sub-set of STL associative containers as displayed below:

1. CBTreeSet is an ordered container type allowing for not more than one instance per unique key value and is equivalent to STL class ::std::set.
2. CBTreeMultiSet is an ordered container type allowing for more than one instance per unique key value and is equivalent to ::std::multiset.
3. CBTreeMap is an ordered associative container type allowing for not more than one instance per unique key value and is equivalent to ::std::map.
4. CBTreeMultiMap is an ordered associative container type allowing for more than one instance per unique key value and is equivalent to ::std::multimap.

This means that, the interface methods of each associative container are, for the most part, equivalent to the respective STL container they are reflecting. However, there are some differences. Most noticeably the methods get_allocator (), emplace () and emplace_hint () are not present. While get_allocator () is missing by design, since the data layer model is different from the STL classes, the methods emplace () and emplace_hint () have not been implemented yet. Also, constructors using allocator types and initialiser lists as input parameters are not supported. The reason for the absence of those is the same as mentioned above, constructors using involving allocator types cannot be supported and the development of initialiser list support is outstanding. Another noteworthy difference between the STL and the framework containers is, that the insert (const value_type &) method of both ::std::map and ::std::set return a pair type, while the CBTreeMap and

CBTreeSet insert (const value_type &) method return an iterator. Using a different return type for CBTreeMap::insert (const value_type &) and CBTreeSet::insert (const value_type &) was necessary, in order to have the abstract data class CBTreeAssocaitaiveBase (Abstract Data Classes) as an abstract type. Please find section CBTreeMap, CBTreeMultiMap, CBTreeSet and CBTreeMultiSet API to see the full interface description.

Abstract Data Classes

The A. are an additional layer between the Specific Data Classes and the Base Class Stack and unify shared parts of the Specific Data Classes in terms of methods and member variables. Those shared methods may involve API methods exposed by the Specific Data Classes or internal methods to be employed by classes being derived from new or expanded A.

This section is interesting for developers that want to create a set of new Specific Data Classes or expand an existing set. That is the case if:

1. an existing Abstract Data Class needs to be expanded or modified
2. a new Abstract Data Class needs to be created
3. or a set of new A. need to be created

1) To expand an existing abstract data class, it is best to derive a new class from an existing A, that is to be expanded. That new class can be used as a foundation to expand functionality by implementing new methods or to modify the behaviour of the inherited class chain by replacing methods. Said replacing may also involve the Scalability Methods.

2) If a new Abstract Data Class is created, then it needs to be derived from CBTreeBaseDefaults (Specific Default Container to b-tree Methods (CBTreeBaseDefaults)), which as a template parameter called _ti_pos that needs to be populated. If none of the existing Internal Position Template Parameter (_ti_pos) can be applied, then a new internal position type needs to be developed and therewith a new set of Scalability Methods, which are employing the new type.

3) Instead of a new abstract data class, a hierarchy of new A. can be implemented if need be. This means, the A. are not restricted to only one layer and a new set of A. can involve any inheritance topology that additional Specific Data Classes desire. The top of said new hierarchy needs to inherit from CBTreeBaseDefaults (Specific Default Container to b-tree Methods (CBTreeBaseDefaults)), as it is explained above at point 2).

Any expanded or new abstract data class ought to be in a separate space, in terms of source file location, and not part of the b-tree framework common area, while those new source files include btreebasedefaults.h. This means that, new or expanded A. are in separate source files, which are part of the application source code or at least in a separate common area, since these classes are not part of the b-tree framework as it is explained in this documentation.

Abstract Interface Classes

The A. provide an abstract interface for a sub-set of Specific Data Classes, which is intended to be used by a calling application to access containers in a target agnostic manner. A. are derived from the Abstract Container Interface (CBTreeIf), which means methods to create iterators and basic methods such as clear (), size () or empty () are present by default. Unfortunately, it is not possible to support member functions that involve template parameters, due to restrictions of the capabilities of C++. This means that, A. are limited to non-template methods only.

The remaining part of this section is only interesting for programmers who want to create a new Abstract Interface Class. The new interface class has to be virtual derived from CBTreeIf (Abstract Container Interface (CBTreeIf)), so that instances, abstracted to that type, can access methods offered by that type. Also, any Specific Data Classes must be virtual derived from the new type directly or via the Abstract Data Classes, to ensure all virtual methods are populated with code on the Specific Data Classes level and therefore are abstracted.

Base Class Stack

The B. consists of two parts: the interface and the base part. The interface part is formed of the classes Abstract Container Interface (CBTreeIf) and Default Specific Interface Methods (CBTreeDefaults), while the base part is formed of Abstract Container to b-tree Interface (CBTreeBaseIf) and Specific Default Container to b-tree Methods (CBTreeBaseDefaults). The major difference between the two interface and the two base classes in terms definition, is that the interface classes are lacking the Internal Position Template Parameter (_ti_pos), the Node Iterator Type (_t_nodeiter) and the Sub-Node Iterator Type (_t_subnodeiter) template parameters. Both the interface and the base part consist of an abstract class (CBTreeIf and CBTreeBaseIf) and a regular class (CBTreeDefaults and CBTreeBaseDefaults). While the abstract classes only contain pure virtual methods and no member variables, in order to be abstract, the regular classes contain the default specific code for the said abstract classes. (As of this release “0.04a”, this is not entirely true and since implementation is incomplete!) The reason why Abstract Container Interface (CBTreeIf) is abstract so that the Abstract Interface ClassesAbstract Interface Classes are enabled to be derived from it. Abstract Container to b-tree Interface (CBTreeBaseIf) on the other hand has been introduced, as it wasn't clear during the development, what level of abstraction is needed within the B. and therefore it has been decided to give the development more freedom by abstracting the base part.

This section explains the role of every class within the B. from top to bottom, which is in reverse order as oppose to the rest of Class Hierarchy. It is more likely for a developer to be interested in how to interface with the b-tree framework via the Abstract Container Interface (CBTreeIf) methods, then how the internals of this framework function in order to created or expand existing classes, as it is explained above (see: Abstract Data Classes).

Abstract Container Interface (CBTreeIf)

The A. class is the lowest common denominator in terms of an interface for an application that is container agnostic. This means that, any of the Specific Data Classes, and therewith any Application Class, can be dynamic cast to this type. This class only contains pure virtual methods, also member variables are not present, so that this class can act as the interface as described in the section above. The default code for the declared methods is provided by the Default Specific Interface Methods (CBTreeDefaults) class.

Iterator Functionality Has Been Moved Into Containers

Iterators are employed to refer to a specific data set in a specific data container, which in terms of how data is ordered in a container can be very different between Specific Data Classes. This would mean any specific data class would require its own set of iterator types* being defined, which would contradict the abstract nature of the Abstract Container Interface (CBTreeIf) class, as it can call container agnostic iterators into existence. To achieve that, iterators need to use the Scalability Methods, which they can do via the Iterator Container Access Methods. However, both sets of methods are protected. Hence, for that to work, all b-tree framework iterator types are declared as a friend type by any class within this framework, so that protected methods can be accessed by any iterator type. This allows for the Iterator Container Access Methods to be used as a means to manage a so called external state, which reflects what the respective container sees as a location, but is agnostic to the iterator itself. Outside the external state, an iterator also has a pointer to a container, a linear offset and a time stamp. The external state itself, contains a node ID plus a sub-node offset as well as a linear offset associated to the node ID / sub-node offset pair, which is only visible to the container. Therefore, to an iterator instance, its external state is just a void pointer. For a container, the external state contains enough information to refer to a specific data entry or to tell an iterator that an update in terms of the external state among other things is required. This means that, if an iterator has to perform an access, then the iterator's and the container's time stamp are tested to be the same. If they are the same, then the iterator's and the external state's linear offsets are tested to be the same. If the offsets are the same, then an access can be safely performed. Otherwise, if the time stamps or the linear offsets are mismatching, then the external state must be evaluated using the iterator's linear offset. Evaluating an external state, means, that the iterator's linear offset is copied to the external state's linear offset and then the new external state's linear offset is employed to derive the node ID / sub-node offset pair, using the methods evaluate_iter () or evaluate_iter_by_seek ().

* set of iterator types: this refers to the four iterator types which can be found in association with every STL: iterator, const_iterator, reverse_iterator and const_reverse_iterator

Default Specific Interface Methods (CBTreeDefaults)

This class provides the default code for the Abstract Container Interface (CBTreeIf) in terms of the CBTree API. This means that, all virtual public and some protected methods have their default code set here, while all Iterator Container Access Methods remain pure virtual methods at this point. The Iterator Container Access Methods need the Scalability Methods to work as actual code, which don't exist at this point in the Base Class Stack.

Abstract Container to b-tree Interface (CBTreeBaseIf)

This class contains the pure virtual definitions of all Scalability Methods and has been made abstract, since it wasn't clear what level of abstraction is needed within the framework. Therefore, it has been implemented this way to have more freedom in terms of development.

Specific Default Container to b-tree Methods (CBTreeBaseDefaults)

This section is relevant for readers, who intend to create one or more new Abstract Data Classes.

The class CBTreeBaseDefaults provides the code for all skeleton methods to function as a b-tree. This means that, the class is capable to create, destroy, merge and split nodes as well as the ability to insert or remove data entries on a very basic level. However, this class has no concept of what a data set position is (see: Internal Position Template Parameter (_ti_pos)) and thus provides the definition of pure virtual methods, which form the Virtual Base Class Methods, allowing Specific Data Classes and Abstract Data Classes to access data entries, by providing code for said methods. Also, this class provides the default code for the Iterator Container Access Methods, as at this point in the Base Class Stack all Scalability Methods are at least declared as pure virtual methods and therefore can be referred to by the afore mentioned iterator methods.

The base class also provides a virtual method called rebuild_node (), in order to rebuild a node's integrity after data was inserted or removed. By default it updates the node's maximum index based on its sub-nodes maximum indexes, while assuming that said indexes have been updated before hand or have not been altered. Newly created Specific Data Classes or Abstract Data Classes need to provide their own version of this method, if that new class type employs additional pools that are in any way integral to the b-tree structure.

Also a virtual method called convert_pos_to_container_pos () is provided, which allows for linear read / write accesses to any data entry by recursively tracing into the tree structure. Creating one or more new Specific Data Classes or Abstract Data Classes providing their own version of this method is optional, but recommended if data pools can be exploited to recursively trace into a sub-tree and can be beneficial in terms of access speed.

Node Descriptor

Every node has a N. telling its parent node (nParent), the number of data items stored in this node (nNumData) and the amount of data items contained by this node and this node's sub-trees combined (nMaxIdx). The N. is defined as follows:

```
typedef struct node_s
{
	_t_nodeiter			nParent;
	_t_sizetype			nMaxIdx;
	_t_subnodeiter	nNumData;
} node_t;
```

The member variable nParent contains the this node's parent node id. If nParent is set to the current node's id, then this node is the root node. The type of this member is defined by the template parameter _t_nodeiter and the description of that template parameter can be found here: Node Iterator Type (_t_nodeiter)

The member variable nNumData (number of data items) contains the fill state of a node in terms of data items. The number of sub-trees a node refers to is nNumData plus 1. If the top bit of that member variable is set or that member is set to zero, then this node is a leaf node and nNumData must be interpreted as its 2's complement value (ie. ~(nNumData – 1)) to calculate this node's data item fill state. In case this node is a leaf, then the number of sub-trees for this node is zero. The type of this member is defined by the template parameter _t_subnodeiter and the description of it can be found here: Sub-Node Iterator Type (_t_subnodeiter)

The member variable nMaxIdx (maximum index) contains this node's fill state combined with the number of data entries stored within all of this node's sub-trees. If this node is a leaf node, then nMaxIdx contains this node's fill state again. If this node is the root node, then nMaxIdx contains the entire data container's fill state, which can be obtained by calling the method size (). The type of this member is defined by the template parameter _t_sizetype and the description of it can be found here: Size Type (_t_sizetype)

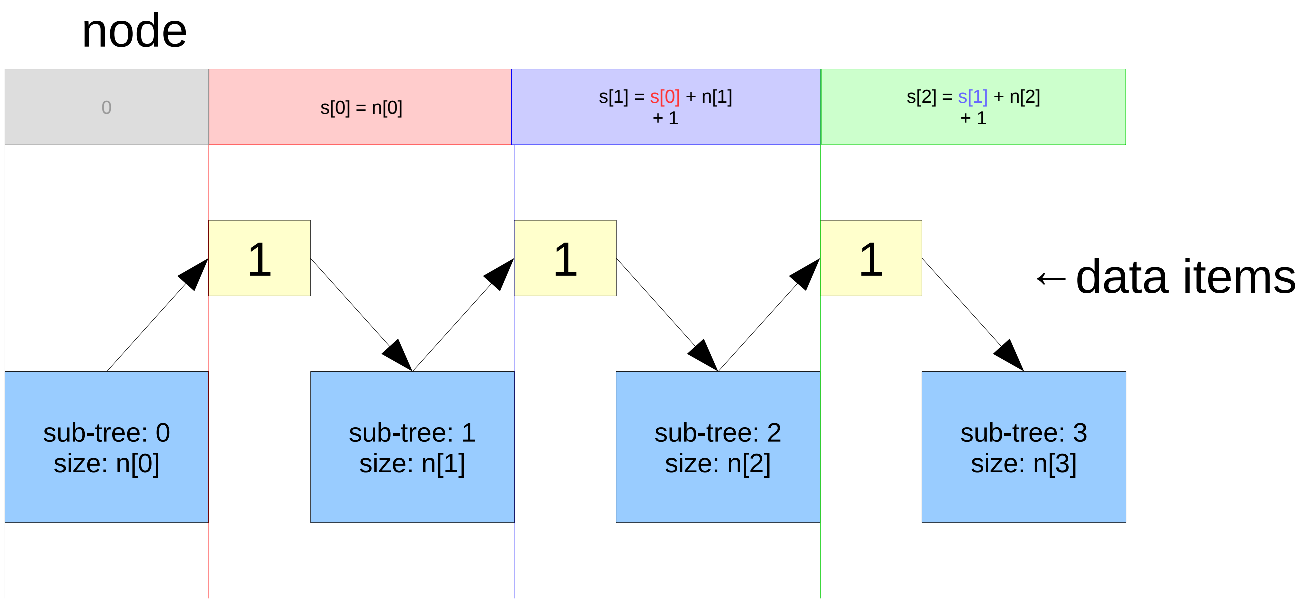
Maintenance Vector

The M. vector is part of every node and actually a bit vector, which contains the status of its associated node. Currently only the least significant bit is in use. If that bit is asserted, then this node is valid, has been previously reserved and therefore in use, otherwise this node is invalid and open to be reserved.

Programmers who intend to make additions in terms of per node statuses ought not to use the M., since the remaining part of this vector is reserved for future expansions. If additional custom maintenance information needs to be stored, then an additional pool must be created. See section: Pools (Technical Reference)

Serial Vector Pool

When a method traces into the tree structure of a container to search for a specific data item only known by its linear offset, then walking through a series of sub-node descriptors to find out what sub-node to track into is a lengthy process. To workaround that problem the so called serial vector has been created, for every inner node to have one. The s. is formed of up to nNodeSize times 2 minus 1 entries, which are of type _t_sizetype and contains, in ascending order, for every sub-tree, the accumulated number of data items prior to their start, except for the initial sub-tree. The initial sub-tree's serial vector entry doesn't exist, since that would always be zero. See the figure below:



If the data items of one node as well as all the data items of its sub-nodes were seen as an ascending series of data items, then the serial vector, displayed in red, blue and green, contains the number of data items prior to every data item. While searching a linear offset, instead of walking through the sub-node descriptors, employing the already prepared s. only requires a binary search, which reduces the average number of operations from O(n) = n / 2 down to O(n) = log2 (n), while n is the current data item fill state of the node in question. Sub-tree sizes are not guaranteed to be the same, therefore finding the next node to trace into cannot be achieved by calculating an offset.

Internal Position Types

The sub-sections of this section briefly describe the existing types suitable for the template parameter Internal Position Template Parameter (_ti_pos) and what each of their intended purpose is.

CBTreeArray Position Template Parameter (CBTreeArrayPos <_t_sizetype>)

The class CBTreeArray sets the template parameter _ti_pos to CBTreeArrayPos <_t_sizetype>, when inheriting from CBTreeBase. This array position class (CBTreeArrayPos) has its own template parameter (_t_sizetype), which is set to the same type as the class' CBTreeArray _t_sizetype template parameter is set to.

The array position class has only one member which is of type _t_sizetype and called nIndex, which displays the linear offset of an entry to be accessed. This offset is absolute to the sub-tree, determined by the current node being processed. In other words, whenever, during the tree walk, a sub-node is being entered, the array position is updated to the offset within the sub-tree said sub-node is referring to, until the data item in question has been found in an inner node or eventually in a leaf node.

CBTreeKeySort Position Template Parameter (CBTreeKeySortPos <_t_sizetype, _t_key>)

When CBTreeKeySort is inheriting from CBTreeBase, then the template parameter _ti_pos is set to the key sort position class CBTreeKeySortPos <_t_sizetype, _t_key>, while that position class' template parameters _t_sizetype and _t_key are set to the same types as they have been respectively set to for CBTreeKeySort.

The key sort position class has two members, one which is of type _t_sizetype displaying the instance of a key and another is a pointer of type _t_key referring to a key value. These members are called nInstance and pKey respectively. If nInstance has all bits asserted (ie. nInstance

= ~0x0), then depending on the type of access the entire set of entries sharing the same key is addressed or the method can choose in what order each instance is processed to optimise that process for speed.

What governs the size of the data layer?

In order to store data on the Data Layer the CBTreeBaseDefaults (Specific Default Container to b-tree Methods (CBTreeBaseDefaults)) class requests nodes from the Data Layer. This means that, every time a container instance runs out of allocatable nodes, a number of nodes is requested to be made available by the Data Layer, while the number of nodes per request is governed by the member variable m_nGrowBy. Since requesting additional space has the potential to take a long time, those requests are done in chunks. So, the larger m_nGrowBy is set the smaller is the number of requests for extra space from the Data Layer, but it also means the Data Layer is potentially bloated with unused space.

To determine when to reduce the size of the Data Layer, the CBTreeBaseDefaults class employs three methods called: auto_shrink (), auto_shrink_inc () and auto_shrink_dec (). When auto_shrink () is called, then the method determines the allocated node with the highest identifier and requests the Data Layer to de-allocate any space beyond the node. This has the potential to create a number of problems. Firstly, if m_nGrowBy would be used as a block size parameter to determine when to de-allocate, then the risk of so called border hopping would exist, in case data gets rapidly inserted and removed again. Secondly, if auto_shrink () is always called when a data entry is removed, then this method would spend a lot of time to determine the last node, since that process is iterative. To address those issues the member variable m_nAutoShrinkThreshold is employed in order to prevent too frequent calls to auto_shrink (). When a data entry is removed, then auto_shrink_inc () is called. This method increments an auto shrink counter by one and once said counter exceeds m_nAutoShrinkThreshold the method auto_shrink () is exercised. This mechanism prevents too frequent requests to the Data Layer to reduce size. The address the border hopping issue, the method auto_shrink_dec () is called when a data entry is inserted. This method decrements the auto shrink counter by one unless said counter is already zero. As a result potential requests to reduce size to the Data Layer are delayed.

Super Class (CBTreeSuper)

The S. class provides everything that is not based on any template parameter, such as methods to instantiate pools and time stamp methods for iterator support (Iterators). As shown in the figure at the beginning of section Class Hierarchy the Abstract Container Interface (CBTreeIf) class is derived from the S.

Also, an application may use this class as an abstract data class type (Specific Data Classes), which aren't sharing any of their template parameters, if that is desired. However, this abstract type offers almost no functionality accept for the method get_time_stamp ().

Pools (Technical Reference)

P. displays what data is stored in which quantities per node, for the data layer to know how much space has be allocated. By default the constructor of CBTreeBaseDefaults registers the following pools:

- descriptor (item count:1)
- maintenance vector (item count: 1)
- application data array (item count: node size times 2 minus 1 → n = t * 2 - 1)
- sub-node ID list (item count: node size times 2 → n = t * 2)
- serial vector (item count: node size times 2 minus 1 → n = t * 2 - 1)

If a data or application class requires to register additional p., then that has to happen during the construction of a said class by calling the method: create_dataPool (). This method returns a pool ID, which then has to be used to address the newly created pool later on. As for application classes, if additional pools are created on that abstraction level, then the application class has to provide its own version of the Scalability Methods. For more information about how p. are organised within the data layer, please see section: Pools

create_dataPool ()

```
uint32_t create_dataPool (CBTreeIOperBlockPoolDesc_t *pPoolDesc)
```

Description:

This method registers an additional pool.

Input parameters:

pPoolDesc	- pointer to structure displaying pool properties (see below)

```
typedef struct
{
	uint32_t		nTotalSize;
	uint32_t		nEntrySize;
} CBTreeIOperBlockPoolDesc_t;
```

nTotalSize	- specifies the entire pool size in bytes
nEntrySize	- specifies the size of one entry in bytes

Remarks:

This method registers an additional pool by appending a copy of the description, being pointed at by pPoolDesc, to the list of the already registered pools. To register a pool to be used by a data class, this method has to be called during the construction.

A pool defines a linear array of one or more entries, depending on what the data class requires. For instance, by default a pool for the node descriptor is registered by the base data class. Said pool contains only one entry, since one node descriptor is required per node. In such a case nTotalSize and nEntrySize are the same. If a maximum of n data items is required to be stored by a pool, then nTotalSize is nEntrySize times n. Hence, nTotalSize must be divisible by nEntrySize. Also, nTotalSize must be greater or equal to nEntrySize.

Return value:

The return value is the newly created pool's identifier. This method doesn't have an error return value and in such a case the method throws an exception (std::runtimeerror::runtimeerror).

Note:

Calling this method outside of a data class constructor, is likely to put that container instance in an undefined state.

get_time_stamp ()

```
btree_time_stamp_t get_time_stamp () const
```

Description:

This method returns a container's time stamp.

Remarks:

This method is employed by iterators, allowing them to tell if a container was modified. The time stamp gets updated if a data entry was inserted, removed or modified. An iterator can then use the time stamp information and compare it to a time stamp copied before, when the iterator was evaluated previously and therefore test if that evaluation is still valid.

Return value:

The returned value is a copy of the container's current time stamp, which is a structure declared as follows:

```
typedef struct btree_time_stamp_s
{
	::std::chrono::high_resolution_clock::time_point		sTime;
	uint32_t								nAccCtr;

	bool			operator== (const struct btree_time_stamp_s &rTimeStamp) const
	{
		return ((sTime == rTimeStamp.sTime) && (nAccCtr == rTimeStamp.nAccCtr));
	}

	bool			operator!= (const struct btree_time_stamp_s &rTimeStamp) const
	{
		return ((sTime != rTimeStamp.sTime) || (nAccCtr != rTimeStamp.nAccCtr));
	}
} btree_time_stamp_t;
```

sTime	- displays the time stamp of the last modification
nAccCtr	- displays the modification counter

Scalability Methods

This section describes a number of methods being part of the interface between the Specific Default Container to b-tree Methods (CBTreeBaseDefaults) and a data class. The Virtual Base Class Methods section lists all methods that need to be created if a new data class is required, while the Methods Utilising Pools section contains all methods that need to be provided, if addition pools (see: Pools (Technical Reference)) for data access optimisation have been created. The table below shows the type definitions for the S.

Typename	Template Parameter or Definition	Description
size_type	Size Type (_t_sizetype)	absolute address type employed by the container for accessing individual

Typename	Template Parameter or Definition	Description
		data entries
position_t	Internal Position Template Parameter (_ti_pos)	internal type employed by the Specific Default Container to b-tree Methods (CBTreeBaseDefaults) to abstract accesses from the Abstract Data Classes
data_layer_properties_type	Data Layer Properties Types (data_layer_properties_type)	parameter defines type use by Specific Default Container to b-tree Methods (CBTreeBaseDefaults) to call a data into existence
node_iter_type	typename data_layer_properties_type::node_iter_type or Node Iterator Type (_t_nodeiter)	type defines node address type
sub_node_iter_type	typename data_layer_properties_type::sub_node_iter_type or Sub-Node Iterator Type (_t_subnodeiter)	type defines sub-node address type

Virtual Base Class Methods

The V. methods translate back and forth between the addresses being used by the application accessing the b-tree and the actual b-tree's node / sub-node pairs and as a result act as an interface. That interface basically involves every method, which uses a parameter type of position_t. The base class is incapable to translate between a linear position and a node / sub-position pair, since it is an abstract type. This means that, a data class inheriting from the base class sets the parameter position_t and therefore must provide its own version of the methods listed in this section. These methods ought to be virtual, as this allows classes inheriting from interface classes to replace virtual base class methods again.

determine_position ()

```
position_t determine_position (position_t sPos, node_iter_type nNode,
sub_node_iter_type &nSubPos, sub_node_iter_type &nSubData, bool &bFound) const
```

Description:

This method is used by other methods to trace into a b-tree structure.

Input parameters:

sPos	- specifies data class specific linear position of sub-tree which nNode is referring to
nNode	- specifies node in which the next sub-node or the data entry of interest is to be sought for

Output parameters:

nSubPos	- returns sub-node position to be entered next
nSubData	- returns data position if bFound is true, otherwise it returns the nearest sub-position for rotate or merge operations that may be used once this method has returned
bFound	- returns true if sPos is pointing at data entry of interest in nNode, otherwise false

Remarks:

This method is used by other methods to trace into a b-tree structure, in order to add, remove or access data entries. The recursive walk down is being achieved by returning enough information to the calling method for it to step into the next sub-tree. That process is stopped once bFound is asserted, which may happen if an inner node is entered and will definitely happen if the current node is a leaf node. The parameter nNode and the return value of nSubData are forming the node / sub-position pair, which is pointing at the data item of interest.

The initial call to this method sets nNode to root node (m_nRootNode) and sPos is set to the data class' specific linear position, that refers to the data entry of interest.

Return value:

In case nNode is referring to an inner node and if bFound is false, then the return value contains the linear position within the next sub-node, otherwise for inner nodes position_t's equivalent of zero is returned. If nNode is referring to a leaf node, then bFound is always true and the return value is equal to sPos.

Note:

The return value is undefined if sPos refers to a position which exceeds nNode's sub-tree size. Also, nSubPos remains uninitialised, if nNode is a leaf node!

generate_prev_position () and generate_next_position ()

```
position_t generate_prev_position (const node_iter_type nNode, const sub_node_iter_type nSub,
position_t sPos) const
position_t generate_next_position (const node_iter_type nNode, const sub_node_iter_type nSub,
position_t sPos) const
```

Description:

Both generate_prev_position and generate_next_position respectively generate the previous or next linear position from the current linear position and the associated node / sub-position pair.

Input parameters:

nNode	- node part of node / sub-position pair associated with sPos
nSub	- sub-position part of node / sub-position pair associated with sPos
sPos	- current linear position associated node / sub-position pair

Remarks:

These methods are used by the method remove_fromNode, in case data has to be copied in order to remove a data entry from an inner node by replacing with an adjacent entry from a leaf node. The return value is then also used to remove the originating data entry of which a copy was taken from.

nNode must not refer to a leaf node, which means, these methods will never bounce at the start or the end of the contained list.

Return value:

Depending on whether generate_prev_position and generate_next_position is called the previous or next linear position is returned.

find_next_sub_pos ()

```
sub_node_iter_type find_next_sub_pos (const node_iter_type nNode, position_t &sPos) const
```

Description:

This method returns the sub-node of an inner node, based on a linear offset.

Input parameters:

nNode	- specifies current node in which a sub-node of interest is to be sought for

Input / Output parameters:

sPos	- references the linear position inside node

Remarks:

This method translates a linear position, displayed by sPos, of a sub-tree, pointed at by nNode, into a sub-position of nNode, which serves as a reference to a sub-node, referring to a sub-tree. Also it returns the linear address inside said sub-tree, which is written back to sPos, once the method has finished.

Return value:

If nNode is referring to an inner node the returned value is either the sub-position referring to the next sub-node or a linear address of data item contained the current node. It is for the calling method to determine whether a search can be aborted early or if a jump to a sub-node is required.

In case nNode refers to a leaf node, then the return value is the linear address to the sought data item and the position displayed by sPos is undefined.

Note:

The behaviour of this method is undefined, in case sPos exceeds the b-tree size nNode is pointing at.

Methods Utilising Pools

Since the CBTreeBase class is agnostic to what dictates the location of a data item within or outside a container instance and also is unaware of potentially added Pools (Technical Reference), it only provides a virtual interface for data classes, being derived from said base class. A data class has to provide some functional methods for it to work properly within this b-tree framework and this section lists and describes what the minimum of those methods is required.

It is strongly suggested that the M. are again virtual, in case additional pools have been created on a higher level of abstraction to optimise data access. By doing so, programmers have the option to derive specialised application classes from existing data classes or to create an abstraction model on application level as they see fit. The method find_next_sub_pos () ought to be part these methods as well, but has already been explained in the Scalability Methods section. If the description of said method is needed, then please find it there.

rebuild_node ()

```
void rebuild_node (const node_iter_type nNode, const int32_t triMod = 0, sub_node_iter_type
nSubStart = 0)
```

Description:

This method is called if a node has been changed in terms of size.

Input parameters:

nNode	- specifies the node that needs to be updated	
triMod	- is a triple state modifier	
	0	- an unknown number of entries has been added or removed
		This means that the descriptor needs to be rebuild.
	-1	- one entry has been removed
	1	- one entry has been inserted
nSubStart	- specifies from what point on the serial vector has to be rebuild	

Remarks:

By default the method updates the node's descriptor to its new size based on the information all sub-nodes provide as well as the Serial Vector Pool. Also, it assumes that all sub-nodes of the node nNode is referring to are up to date.

If triMod is set to zero this method retrieves the size information of every sub-node and combines that with its own size to generate the maximum index for this node's descriptor, while assuming all of the sub-node descriptors are up to date. By touching every sub-node's descriptor, this operation has the potential to create performance issues, if the data layer has a high latency. Furthermore, the situation can exacerbate, since some parts of the cache get overwritten by obtaining said sub-nodes descriptors, which means that frequently used data needs to be reloaded in the future. To avoid cache line oscillations the second parameter can tell the method directly in what way the size of the node has been changed, without a complete rebuild. Also, the parameter nSubStart may remedy the situation. If the calling method knows at which point the node was altered, then nSubStart can tell from what point on the node needs to be rebuild. Specialised data classes may exploit this parameter in order to avoid situations that result in lowered performance.

Note:

Although the parameter triMod is capable to display a size modification by more than one entry, this has not been tested and therefore values other than -1, 0 and 1 ought not to be used for the time being. However, values of larger magnitude are reserved for future expansions. This means that, the parameter triMod is not to be used for anything other than modifying the size of a node!

Optional Pool Utilisation Methods

As oppose to the Methods Utilising Pools, the implementation of these methods in a data class that uses additional pools is optional, but will help greatly exploiting those pools if they have been added to accelerate access in terms of speed.

convert_pos_to_container_pos ()

```
void convert_pos_to_container_pos (const node_iter_type nNode, const size_type nPos,
node_iter_type &rRsltNode, sub_node_iter_type &rRsltSubPos) const
```

Description:

This method recursively walks into the tree structure of a container instance to convert a linear position to its associated node / sub-position pair.

Input parameters:

nNode	- specifies the node in which the data entry is to be sought for
nPos	- specifies the linear position of the data entry in question

Output parameters:

rRsltNode	- refers to where the returned node part is to be stored at
rRsltSubPos	- refers to where the returned sub-position part is to be stored at

Remarks:

This method recursively traces into the tree structure of a data container to convert a linear position to its associated node / sub-position pair. The initial call ought to set nNode to the root node (m_nRootNode) which addresses the entire tree. nPos needs to be set to the entry's absolute linear position.

Note:

The behaviour of the this method is undefined if nPos exceeds the b-tree's size. In other words, the calling method needs to make sure all parameters are correct, since this method hasn't the ability to tell that something went wrong.

Iterator Container Access Methods

This section lists all methods used by this framework's iterators to manipulate an iterator's external state via the methods of its associated container. This means that, since containers have the ability to be abstracted, iterators have to be abstract types too. Otherwise, instantiating iterators on the Abstract Container Interface (CBTreeIf) layer would not be possible. Therefore said layer provides a number of pure virtual methods to be exploited by iterator types, using a so called external state, which is managed and manipulated by the iterator's associated container, while that external state remains to be agnostic for the respective iterator. In other words, an iterator only acts as a front end for a calling application and performs the actual iterator operations by invoking its associated container. That invocation happens via a set of primitive I., which are described below.

Type Definitions

All iterators only have one template parameter, which is the type of container the iterator is going to be used for. This template parameter is called _ti_container and has to provide a number of definitions as shown in the table below:

Type	Definition	Description
value_type	_t_data	This type defines what a data entry is and what is displayed to the application by the container when accessing it.
size_type	_t_sizetype	This defines what type is employed to

Type	Definition	Description
		address individual data entries when accessing a container as a linear array container type.

get_iter_state_size ()

```
uint32_t get_iter_state_size () const
```

Description:

This method returns the size of an external state.

Remarks:

To obtain the size in bytes of an external state as it is being used by the associated container type, this method can be called. The return value can be different from container type to container type, as the external state structure depends on the size of the template parameters: Size Type (_t_sizetype), Node Iterator Type (_t_nodeiter) and Sub-Node Iterator Type (_t_subnodeiter).

Return value:

This method returns the size of the associated container type's external state structure in bytes.

reset_iter_state ()

```
void reset_iter_state (void *pState) const
```

Description:

This method resets an external state.

Input parameters:

pState	- specifies the pointer to an external state structure

Remarks:

To bring an external state to its reset state, this method can be called. Once this call returns, the state will end up having all of its data set to zero.

evaluate_iter ()

```
void evaluate_iter (void *pState, const size_type nOffsetPos) const
```

Description:

This method evaluates an external state to assume a new linear offset.

Input parameters:

pState	- specifies pointer to external state
nOffsetPos	- specifies new linear offset

Remarks:

This method evaluates the external state specified by pState to a new linear position specified by nOffsetPos. The evaluation is done from the root node and therefore the external state doesn't need to be in a valid state before this method is called. If nOffsetPos is greater or equal to the size of the associated container, then call of this method is ineffective and it returns immediately. Otherwise, once this method returns, the external state has been updated and refers to the new linear offset.

evaluate_iter_by_seek ()

```
void evaluate_iter_by_seek (void *pState, const size_type nNewPos) const
```

Description:

This method evaluates an external state to assume a new linear offset.

Input parameters:

pState	- specifies pointer to external state
nNewPos	- specifies new linear offset

Remarks:

This method evaluates the external state specified by pState to a new linear position specified by nNewPos. The evaluation is done from the current location as it is referred to by the external state and therefore it needs to be in a valid state before this method is called. If nNewPos is greater or equal to the size of the associated container or nNewPos is equal to the location as it is displayed by the external state, then call of this method is ineffective and it returns immediately. Otherwise, once this method returns, the external state has been updated and refers to the new linear offset.

is_iter_pos_evaluated ()

```
bool is_iter_pos_evaluated (void *pState, const size_type nPos) const
```

Description:

This method tests if an external state is referring to a specific linear offset.

Input parameters:

pState	- specifies pointer to external state
nPos	- specifies linear offset to be tested for

Remarks:

To test if the position specified by an external state is matching a linear offset specified by nPos, this method can be called.

Return value:

If the offset specified by the external state is the same as the offset specified by nPos, then the return value is true. Otherwise, the return value is false.

get_iter_data ()

```
value_type *get_iter_data (void *pState) const
```

Description:

This method retrieves a data entry from the associated container.

Input parameters:

pState	- specifies pointer to external state

Remarks:

To retrieve a data entry referred to by an external state, this method can be called. Before this method is called, the external state must have been evaluated, otherwise this call has the potential to throw an ::std::run_time_error exception. Once this method has returned, a copy of the returned data set must be taken, since the next access to the associated container may invalidate the data set being pointed at.

Return value:

The return value is a pointer to the data entry of interest.

Note:

This method is not thread safe and therefore doesn't allow for concurrent accesses on the same container via one or more iterators. This comes from the fact that some Data Layer types are not always keeping all data within the application's address space and as a result one read or write access may invalidate the result being pointed at. Hence, concurrent accesses must be managed by the application and an iterator using this method must take a copy to ensure access safety for the calling application.

set_iter_data ()

```
void set_iter_data (void *pState, const value_type &rData)
```

Description:

This method modifies an existing data entry referred to by an external state.

Input parameters:

pState	- specifies pointer to external statement

rData - specifies reference to replacing data entry

Remarks:

To change an existing data entry referred to by an external state, this method can be used. Before this method is called, the external state must have been evaluated, otherwise this call has the potential to throw an ::std::run_time_error exception.

Template Parameters

This section contains descriptions about the template parameters that help defining any of the classes being shown in section Class Hierarchy. Those descriptions also involve template parameters of the Data Layer Properties (_t_datalayerproperties), which display Data Layer Dimension Types in more detail.

Data Type (_t_data)

This template parameter specifies the data type defining what an individual data entry is and must be set. Via the container's interface the application is then presented with an interface equivalent to a linear array of type _t_data.

Key Type (_t_key)

The _t_key template parameter is only used by associative containers that have an order and defines what a key value is, while each key value is part of or derived from its associated Data Type (_t_data) instance. This means that, if two data entries are being compared, to determine their order, then both data entries' key values are extracted using the method extract_key (), while the method comp () is employed to determine the order between the two key values. By default, this template parameter is set to what type the template parameter Data Type (_t_data) is defining and needs to be a different type if any of the following cases occurs:

- the key value is only a part of Data Type (_t_data)
- the key value is not part of Data Type (_t_data), but is derived from it

All of the above cases require the programmer to either A) have the Data Type (_t_data) capable to cast to _t_key or being outfitted with an overloaded cast operator or B) to create a specialised data class or an application class that inherits from CBTreeKeySort and provides its own versions of extract_key () via polymorphism. For more information see: extract_key () addendum

Data Layer Properties (_t_datalayerproperties)

This template parameter defines all data layer dimension types, explained in the sub-sections of this section, and also, once instantiated, contains the data layer configuration. Furthermore, this type provides the Data Layer Properties Types (data_layer_properties_type), which tell a container instance what the underlying data layer looks like. So, when a b-tree container is called into existence, then a copy of the D. instance provided via a constructor parameter is taken, so that the container instance is able to use the D. copy as an input parameter for the data layer instance. This allows an application to set data layer parameters, such as a path name or cache sizes, at run-time in addition to the Data Layer Properties Types (data_layer_properties_type) set at compile time.

Size Type (_t_sizetype)

The template parameter _t_sizetype specifies the address type to be used when individual data sets within a class' instance or a key sub set need to be addessed. By default the parameter is set to unsigned int64, which ought to be sufficient for most applications and therefore can be left unchanged. The intention why this address type was being made a template parameter is to allow either for larger types in the future or to have unsigned int32 or 16 bit types to be compiled on such architectures which don't have intrinsic 64 bit capabilities or run significantly faster with smaller types. The latter point would also result in smaller and hopefully faster code on those platforms.

It is strongly suggested to keep this parameter unsigned, since none of the b-tree classes can make use of a signed type. Also binary operations in the code are expecting an unsigned type and may created undefined results, if a signed type is in use.

Note: This template parameter also sets a type in the node descriptor, which displays the maximum linear index of that node. This means that, it contains the number of application data items as well as all sub-trees total item count of the node in question combined. Hence, for the root node that means, this type must be capable to display the fill state of the entire b-tree.

Node Iterator Type (_t_nodeiter)

A container instance uses nodes to display a tree structure, that is used to contain the application's data and as a result the individual nodes need to be addressable. This type defines what is used by the container instance to do so and ought to be set to an unsigned integer type. By Default, this template parameter is set to an unsigned 64 bit integer and since this ought to sufficient for any application, it rarely needs to be set to anything else. However, this address type was made a template parameter to allow for future extensions. This means that, larger type may be used in order to gain a larger address space in terms of nodes and therefore space for the application to store data or smaller types may be set to slightly increase access speed, since smaller platforms may not handle large types all that well and don't have the physical space available to accommodate a high number of nodes to begin with.

It is strongly suggested to keep this parameter unsigned, since none of the container can make use of a signed type. Also binary operations in the code are expecting an unsigned type and may created undefined results, if suddenly a signed type is in use. Note: This template parameter also sets a type in the node descriptor, which displays the parent node of the node in question. See: Node Descriptor

Sub-Node Iterator Type (_t_subnodeiter)

Each node contains a list of sub-nodes and an array of application data items. This template parameter defines the address type to address individual data sets in said list and array. By default, it is set to an unsigned 32 bit integer and usually there is no need to modify it. For pretty much any application, the default type offers a range wide enough allowing for sub-node lists and application data arrays to contain sufficient amounts of items.

This parameter must remain an unsigned type, since this template parameter also sets a type in the node descriptor, which displays the current number of application data items stored within that node. If a node becomes a leaf node, then the number of data items are being displayed as 2's complement values, although this is an unsigned type. This means that, the b-tree classes contain code manually dealing with negative magnitudes stored using this type, which is expected to be unsigned. A signed type for this template parameter will result in unexpected behaviour.

Absolute Address Type (_t_addresstype)

The A. is used by the data layer type to calculate any absolute byte address within the linear address space of the physical data layer medium. For the data layer to work correctly it is expected that the set type is an unsigned integer, which is large enough to address all data to be accommodated.

Relative Address Type (_t_offsettype)

The R. is used by the data layer type to calculate any byte address within one block. For the data layer to work correctly it is expected that the set type is an unsigned integer, which is large enough to address all data within one block. Also, it is assumed that the R.'s size is smaller or equal to the Absolute Address Type (_t_addresstype) size.

Internal Position Template Parameter (_ti_pos)

Since the classes CBTreeBaseIf (Abstract Container to b-tree Interface (CBTreeBaseIf)) and CBTreeBaseDefaults (Specific Default Container to b-tree Methods (CBTreeBaseDefaults)) have no concept of what a position is or what dictates where entries would end up relative to other entries inside the b-tree structure, a set of methods, which can be found in section Virtual Base Class Methods, is required to define that. When a data class is inheriting from CBTreeBaseDefaults, not only must it deliver code for the afore mentioned methods, it also must set a type for _ti_pos, which allows to display relative positions within the tree structure. This means that, in general any method allowing for access to a data container must translate a position set by the calling application to _ti_pos. It is recommended to keep this translation as simple as possible, not only because the generated code could create speed penalties, but also a programmer reading or manipulating the source code must translate back and fourth between the problem the application tries to solve and the positions are handle inside the container.

Container Type Definitions

The C. section discusses any type definition present in all some of the framework's container types. However, some of those type definitions already have been discussed in the above Template Parameters section and it would be redundant to show them in this section again. The table below displays which Template Parameters are an alias to what type definition.

Typename	Template Parameter or Definition	Description
value_type	Data Type (_t_data)	data type that is linearly accessible via a container instance
key_type	Key Type (_t_key)	key type that is used as an operand type when ordering data in an associative container
size_type	Size Type (_t_sizetype)	absolute address type employed by the container for accessing individual data entries
difference_type	typename ::std::make_signed<size_type>::type	relative address type employed by the container to display distance between two data entries
reference	value_type &	type in use when referring to a read / write data entry
const_reference	const value_type &	type in use when referring to a read-only data entry
pointer	value_type *	type in use when pointing to a read / write data entry
const_pointer	const value_type *	type in use when pointing to a read-only data entry
position_t	Internal Position Template Parameter (_ti_pos)	internal type employed by the Specific Default Container to b-tree Methods (CBTreeBaseDefaults) to abstract accesses from the Abstract Data Classes
CBTreeIf_t	CBTreeIf<value_type, size_type>	type aids defining iterator types – see below
iterator	CBTreeIterator<CBTreeIf_t>	type defines abstract read / write iterator
const_iterator	CBTreeConstIterator<CBTreeIf_t>	type defines abstract read-only iterator
reverse_iterator	CBTreeReverseIterator<iterator>	type defines abstract read / write reverse iterator
const_reverse_iterator	CBTreeConstReverseIterator<const_iterator>	type defines abstract read-only reverse iterator
data_layer_properties_type	Data Layer Properties Types (data_layer_properties_type)	parameter defines type use by Specific Default Container to b-tree Methods (CBTreeBaseDefaults) to

Typename	Template Parameter or Definition	Description
		call a data into existence
node_iter_type	typename data_layer_properties_type::node_iter_type or Node Iterator Type (_t_nodeiter)	type defines node address type
sub_node_iter_type	typename data_layer_properties_type::sub_node_iter_type or Sub-Node Iterator Type (_t_subnodeiter)	type defines sub-node address type
address_type	typename data_layer_properties_type::address_type or Absolute Address Type (_t_addresstype)	type defines absolute linear byte address type
offset_type	typename data_layer_properties_type::offset_type or Relative Address Type (_t_offsettype)	type defines linear byte address type within one block
data_layer_type	typename data_layer_properties_type::data_layer_type or Data Layer Type Definition (data_layer_type)	type defines data layer type to be instantiated once data need to be stored
has_latency	typename data_layer_properties_type::has_latency or Latency Property Definition (has_latency)	type defines if the data layer has the potential for a significant latency once being accessed

Data Layer Properties Types (data_layer_properties_type)

The role of D. already has been explained in section Data Layer Properties (_t_datalayerproperties). This section only exists to introduce the sub-section containing the type definitions of the D., rather than the template parameters.

Data Layer Type Definition (data_layer_type)

The _t_datalayer template parameter must be set to a class type compatible to the interface described in section Data Layer. An instance of this type is called into existence as soon as data is inserted into the b-tree. The data layer class must be able to manage how and where data, coming from the b-tree, is stored and on request read back. By default this parameter is set to CBTreeRAMIO, which, as the name already suggests, stores all data in the local memory of the machine the application is running on. An alternative is CBTreeFileIO, which is capable to hold large parts of the b-tree data in a file and can be configured to only use a tiny fraction of memory, if the local memory is insufficient or it is not known if a target machine has enough RAM.

The data layer classes have their own set to template parameters, which need to be compatible with the above b-tree class template parameters. For more information see: Data Layer Template Parameters.

Latency Property Definition (has_latency)

The data layer properties has_latency type tells the container type to be compiled, whether an access to the data layer has a latency long enough where, in terms of optimisation approaches, it is feasible to consider other options to gain the same result. For instance, if the data layer is using RAM as a medium to store data, then calculating a from other sources is likely to take longer than to simply request the result from the data layer. Otherwise, if a file is in use, then an access has the potential to take milliseconds, which is an eternity in terms processing speed on a modern CPU.

Note: This type only tells the container that there is a potential for a significant latency and does not consider data layer caches or operating system's buffering strategies.

Application Classes

When writing applications, it is highly recommended to always create application classes by inheriting from one of the already existing Specific Data Classes, even if the newly created application class would remain empty at first. This allows for modifications later on, since it is doubtful that the Specific Data Classes, will be sufficient during the entire life span of a project. Creating application classes also enables to write interfaces, which address a specific problem the application is trying to solve and hide the inner workings of the b-tree. In other words, it ought to aid the solving process greatly, by not being required to translate back and fourth between the problem to solve and an abstract container interface. Also, an application class then may act as an API wrapper, as the base for more specialised application classes or as a container type directly.

Assign Methods

Containers can be assigned with data via a copy-constructor, assignment operators or in some case via an API method. In any event, the container is supposed to end up with the newly assigned data, which means, in any of the above scenarios, the same code can achieve that.

Therefore, this section is not discussing the API method assign () or assignment operators, but the internal method _assign (). As for the copy-construction, please see the next section (Assignment During Copy-Construction) as well, since the situation of calling virtual methods during any construction is different than calling virtual methods from regular code. This method is virtual so that application classes are capable to provide their own version, in case additional content needs to be transferred during the assignment operation.

The default _assign () methods provided by each CBTreeArray and one of the Abstract Data Classes called CBTreeAssociative, are processing data in a skewed fashion, instead of walking, data entry-by-data entry, through the source container. Because, if all source entries were linearly copied from beginning to end, then the resulting destination b-tree would have a drift in term of balance towards the end. To avoid that data is copied out of order. So, once all existing content in the destination container has been destroyed, the destination container's future depth is determined based on the size of the source container and the destination container's node size. The destination container's depth is dd = INT (t * 3 / 2) * INT (round up) ((log (ssize) / log (t))), whereas t is the destination container's node size and ssize is the source container's number of data entries. Next, the skew width is determined by n = dd + t – 1. This means, the copy process happens in n passes with each pass copying every n-th entry, while skewing the start location by one per pass. As an example: If it is assumed 10 entries are to be copied and a skew width of 3 has been determined, then the first pass copies the entries: 0, 3, 6, 9 the second pass the entries: 1, 4, 7 and the final pass the entries 2, 5, 8. As a result the destination b-tree will be balanced and performance issues towards the end of the destination container are less likely. However, since the transfer is out or order, the sequence of entries sharing the same key is not guaranteed to be the same between the source and the destination container.

Assignment During Copy-Construction

As mentioned in the above section, calling _assert () during the construction of an instance is problematic, since said method is virtual and the v-tables are set up, once all sub-constructors have returned. This means that, if an application class provides its own version of _assign (), since it needs its content assigned in a different way than the default _assign () code of the Specific Data Classes processes it or additional content needs to be copied across, then the application class' version is not effective during the copy-construction within the b-tree framework. An application class' _assign () method during a copy-construction is only effective in the application class' copy-constructor itself, although the method is virtual. As a result, the copy-constructors of CBTreeArray and CBTreeAssociative (part of the Abstract Data Classes), would call the default method, which in this case is insufficient. Hence, _assign () has to be called from the application class' copy-constructor and also any of its insufficient versions would be called on lower layer class' copy-constructors within the framework, which has the potential to be a huge performance issue or in the worst case to create unexpected behaviour. To avoid that, during the copy-construction of an application class, the parameter bAssign of the respective Specific Data Class (Specific Data

Classes) within the sub-constructor call needs to be set to false, so that the default code of _assign () is not called. Also, once all sub-constructors have returned and the focus is back with the application class' code, then _assign () ought to be called. This call ought to be conditional, the same way it is within the Specific Data Classes and the Abstract Data Classes. That means, the application class' copy-constructor should have a boolean parameter, which must have a default value being set to the value true, in addition to the constant reference to the source container's type. If the value of that parameter is false, then the application class' _assign () method is not called. It is strongly recommended to do so, in order to avoid calling the wrong _assign () method, in case an application class becomes an abstract type within the application classes hierarchy, while more specific application classes provide their own _assign () method code.

Application Specific Key Sort Classes

As already pointed out in one of the sections above, Application Classes ought to inherit from one of the existing Specific Data Classes to modify these by employing polymorphism. CBTreeKeySort contains two methods that most likely need be changed. One of which is comp () and by default this method arithmetically or binary compares entire keys extracted from data sets, when it comes to searching. For more information see: Default comp () addendum. The other method is extract_key () and by default casting from type _t_data to type _t_key or calls a version of this method provided by the application class via polymorphism. For more information see: extract_key () addendum. As for the Associative Containers (CBTree[Multi]Map|Set), only comp () needs to be considered, since if any of the Associative Containers (CBTree[Multi]Map|Set) needs to provide its own version of extract_key (), then it is probably best CBTreeKeySort to begin with. This means that, the Associative Containers (CBTree[Multi]Map|Set) already provide an extract_key () method so that these container to reflect the behaviour of the respective STL equivalent containers and a modified version of extract_key () void the point of using these conainers.

comp ()

```
int comp (const _t_key &rKey0, const _t_key &rKey1) const
```

Description:

This method is employed to compare two keys.

Input parameters:

rKey0	- specifies reference to first key
rKey1	- specifies reference to second key

Remarks:

This method determines which of the input keys is deemed greater or if they both are equivalent.

Return value: If both keys display the same value, then the return value is zero, otherwise greater than zero if rKey0 is greater than rKey1 and less than zero if rKey0 is less than rKey1.

Note:

This method is called frequently during any access, therefore, when an application class provides its own version of this method, the code ought to be optimised, otherwise performance issues are to be expected.

Default comp () addendum

By default this method acts as an arbiter between an arithmetic and a binary version of comp (), by using `typename ::std::is_arithmetic<_t_key>::type` to select which method is to be used. In other words, if the key template parameter is deemed to be an arithmetic type, then `int comp (const _t_key &, const _t_key &, ::std::true_type)` is called, otherwise `int comp (const _t_key &, const _t_key &, ::std::false_type)`.

In case an application class has its own version of comp (), then some coding issues ought to be kept in mind. Firstly, if the key template parameter is an arithmetic type, a simple subtraction could be employed to create the return value immediately. While an arithmetic subtraction is the fastest way to get a result, it isn't always the best. The reason being, the return value is always an integer and the subtraction has the chance to overflow and also, if a floating point type is in use, which is then type cast to the result integer type, to underflow. As a result, this creates undefined behaviour. Secondly, on big endian architectures the binary comp () method may seem sufficient enough, as it creates results very similar to what an arithmetic comparison would, given the key template parameter is an integer. However, this is likely to create problems when the code is compiled for little endian architectures and the erroneous results are not immediately obvious. For some cases the second point can be ignored. For instance, if the order of keys is not relevance to solve a problem and the key is only used as a mean address a sub-set of data items, or in case it is only required to know if a key is present in the container or not.

extract_key ()

```
_t_key *extract_key (_t_key *pKey, const _t_data &rData) const
_t_key *extract_key (_t_key *pKey, const _t_nodeiter nNode, const _t_subnodeiter nEntry) const
```

Description:

These methods are used to convert internal or external data sets to their associated keys.

Input parameters:

rData	- specifies the reference to a data set to be converted
nNode	- specifies the node of the node / entry pair containing the data set to be converted
nEntry	- specifies the entry of the node / entry pair containing the data set to be converted

Input- / Output parameter:

pKey	- specifies the pointer where the key is to be stored at

Remarks:

When the key value of a data set is required to be compared against another key value, then these methods are used to perform the conversion. Since that happens frequently, when it comes to any access, it is required to optimise the code as well as the data sets to suit fast processing if a new key sort class with a new e. method is created, otherwise performance issues are to be expected.

Return value:

The return value always is pKey again. These methods are incapable to return error conditions.

Note:

extract_key (_t_key *, _t_nodeiter, _t_subnodeiter) in general calls extract_key (_t_key *, const _t_data &). This means that, if a new class inheriting from CBTreeKeySort is created, which requires a different way to convert a data set into its key, then only a new version of extract_key (_t_key *, const _t_data &) needs to be created.

extract_key () addendum

The default method extract_key () of the class CBTreeAssociative (Abstract Data Classes) acts as an arbiter between an extract_key () version that casts from the data type template parameter (Data Type (_t_data)) to the key type template parameter (Key Type (_t_key)) and a virtual method call of extract_key () potentially given by an application class that has been derived from CBTreeAssociative. Based on what `typename ::std::is_convertible<_t_data, _t_key>::type` returns at link time it is determined whether `_t_key *CBTreeAssociative::extract_key (_t_key *, const _t_data &, ::std::true_type)` or `_t_key *CBTreeAssociative::extract_key (_t_key *, const _t_data &, ::std::false_type)` is called. The former call casts the input parameter `const _t_data &` to `_t_key` and returns it. The later method simply calls `this->extract_key (const _t_key *, const _t_data &)`, which is the version of extract_key () given by an application class. This means that, (1) if _t_data can be cast to _t_key and the application class has not provided its version of extract_key (), then `_t_key *CBTreeAssociative::extract_key (_t_key *, const _t_data &)` re-routes to `_t_key *CBTreeAssociative::extract_key (_t_key *, const _t_data &, ::std::true_type)` and the afore mentioned cast is applied. (2) If _t_data cannot be cast to _t_key and the application class has provided its version of extract_key (), then application class version supersedes `CBTreeAssociative::extract_key ()` and every conversion goes through the application version *. (3) If _t_data can be cast to _t_key and the application class has provided its version of extract_key (), then application class version still supersedes `CBTreeAssociative::extract_key ()` and case (2) applies anyway *. (4) If _t_data cannot be cast to _t_key and the application class has not provided its version of extract_key (), then `_t_key *CBTreeAssociative::extract_key (_t_key *, const _t_data &)` re-routes to `_t_key *CBTreeAssociative::extract_key (_t_key *, const _t_data &, ::std::false_type)`, which calls `this->extract_key (const _t_key *, const _t_data &)` and since it has not been replaced `_t_key *CBTreeAssociative::extract_key (_t_key *, const _t_data &)` is called again. This is going to create a stack overflow at run time! Hence, a conversion via a poly-morphed extract_key () method or an overloaded cast operator in _t_data that casts to _t_key has to be provided.

* In the event that an application class (A) derives from an application class (B) and class (B) derives from CBTreeKeySort or any other of the Associative Containers (CBTree[Multi]Map|Set)and class (A) provides its own version of extract_key () and class (B) needs to temporarily call an instance of its own type into existence, then the temporarily created instance of class (B) will be using the extract_key () version of CBTreeAssociative and not the version provided by class (A). This is likely to result in unexpected behaviour, which a programmer needs to keep in mind. For that reason, it is recommended not to instantiate any application class inside an application class, even if it seems to be the same type.

What are b-tree classes incapable of?

This section addresses issues or situations this framework's classes are incapable to handle and may not be obvious. The aim of this section is to veer programmers away from trying to create solutions based on assumptions that aren't correct, by making them aware of this framework's limitations.

Concurrent Accesses

Any concurrent access, including concurrent read accesses, leads to an undefined result, because it is not guaranteed, that on data layer level, every cache line access is exclusive. This means that, if two or more concurrent processes are accessing the same b-tree instance, then this is likely to result in a race condition, since this provokes cache line conflicts. To work around that problem, a new data class ought to be created by inheritance, wrapping all access methods, which are not exclusive used by design.

Although the above statement is only true for data layer classes that involve the usage of exclusive resources, such as cache lines, when accessing data, it is highly recommended to keep accesses exclusive at all times. Since the data layer type is set in the template parameter list, it can be changed at any point during the development of an application. If, for instance, a b-tree class was developed with _t_datalayer being set to CBTreeRAMIO and it is decided CBTreeFileIO has to be used instead, then retro fitting all concurrent accesses to the b-tree are likely to be very time consuming.

Data Layer

When a b-tree instance requires IO access, an instance of a d. object will be called into existence, if it hasn't been already. That d. instance follows either the linear or the block d. model, explained below. The d. instance will handle every IO access by translating any requested item to a pointer within the application's address space and as a result allowing for read and write operations.

Data Layer Models

Currently two major d. are in existence, which are the linear and the block data layer model.

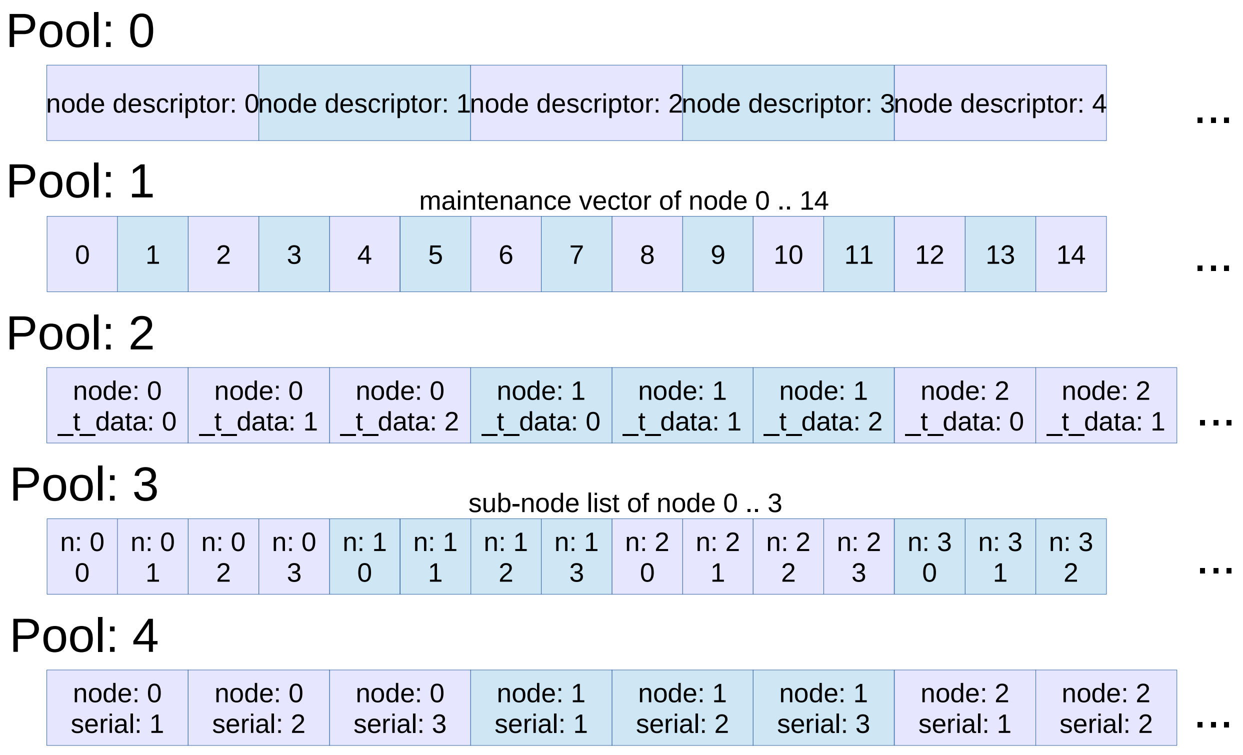
While the Linear Data Layer Model keeps data persistently available, the Block Data Layer Model loads and unloads blocks to keep the required amount of resources limited.

If a new data class is created or modified, then the following needs to be taken into account: The pointers returned by any data layer instance are not guaranteed to stay the same and may change when the data layer is requested to change its size. This means that, when certain methods are being called, then previously acquired pointers must be re-acquired. A list of those methods can be found in this section: List of Base Class Methods Potentially Causing a Re-Allocation

Linear Data Layer Model

Data layers following the L. create a persistent linear data array per pool. This means that, the potential to have data loaded before it can be processed doesn't exist. Also, any kind of caching or buffering doesn't exist.

When the pointer to specific data item is requested, the respective pool gets selected, the offset to that node's initial item in the selected pool is calculated by multiplying the node ID in question with the return value of get_pool_total_size () and finally the entry's offset within the node is obtained by multiplying the entry number and the return value of get_pool_entry_size (). See the diagram below:

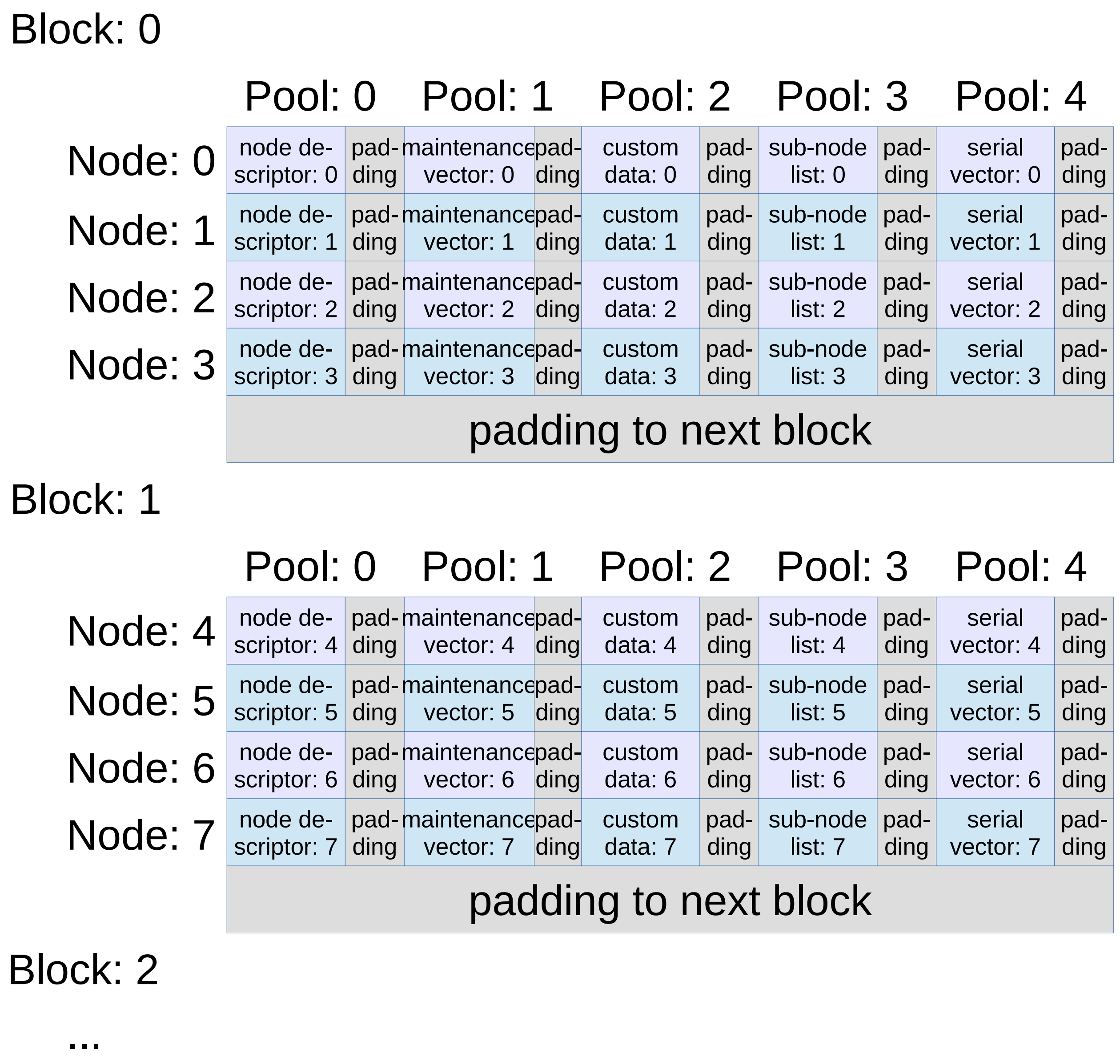


As it can be seen, the L. is basically an SOA, so that any data item associated with any node in any pool can be accessed with the least amount of afford possible.

Block Data Layer Model

Data layers following the B. organise nodes in blocks, while each block has the same size and therewith the number of nodes contained by each block is equal. As oppose to the Linear Data Layer Model data may not be stored persistently and parts of the data layer are written back to the underlying physical medium once an operation has finished. The B. is not using a caching approach to decide when to read and write blocks and instead controls that by maintaining a descriptor list, which is permanently present. Once a block has been made available to allow for requests being made, it is flagged as present in its respective descriptor and guaranteed not to be unloaded during the rest of the operation. Also, the descriptor is keeping track of how many times a block is being accessed, so that when an operation completes, based on that statistical information it can be decided, which blocks are staying ready for the next operation and which blocks are written back to the physical medium. In particular this means, each time a block is being accessed either that block's data is being made available in the application's address space and the respective descriptor's access counter is reset or the access counter of the block in question is incremented, if the block was present already. If an access counter exceeds a given threshold value (currently set to 127), then all active descriptor's access counters are divided by two, which means that infrequently accessed blocks are more likely to be unloaded later in the process. However, as soon as a block has been made available it stays resident until terminate_access () is called, which happens when a read, write, insert or remove operation has completed. terminate_access () is called when an operation is about to return to the calling application. That method starts to unload blocks, beginning with those having the lowest access counter, by writing them back to the physical medium, until the total amount of resources being occupied by those blocks is less or equal the address space soft limit. The only exception is the block containing the root node, which always stays resident, as the root node is definitely being employed for any access. Even simple methods like empty () or size () require root node access. The only way to unload the root node containing block as well, is to call unload ().

Since every access requires the conversion from a node ID to its block, which involves a division and can be very slow, the number of nodes per block is always part of the 2 to the power of n series. This allows for a shift operation to be used instead a division. As the number of nodes per block = (block size in bytes / node size in bytes) cannot go up, the selected value for n creates a result to the next smaller or equal value of nodes per block. See the diagram below:



The diagram above shows, the data layer is divided into a linear list of blocks. Those are involving a padding to align each block to the data layer's ideal block size. This is being done to optimise accesses in terms of speed. As mentioned before, every block contains a linear list of one or more nodes depending on the block's size. However, if a node is larger than a the data layer's block size, then blocks are being grouped, until at least one node fits into a group of blocks. Each group of blocks is trailed by padding to align them to the next same sized group of blocks.

Every node's pool items have a padding, which aligns the following pool item to the next address divisible by the retuned value of get_dataAlignment ().

List of Base Class Methods Potentially Causing a Re-Allocation

This section contains a list of Specific Default Container to b-tree Methods (CBTreeBaseDefaults) methods, which may cause a change of size for the data layer instance. It is important to understand, since pointers that have been obtained prior to one of those methods below, are no longer valid once any of them has been called and they must be re-obtained. If a new data class is created or modified, this must be accounted for, otherwise this section may be omitted.

add_to_node ()

This method recursively traces into the tree structure and eventually calls insert_data_into_node (), which has the potential to require more nodes. Please find that method below.

insert_data_into_node ()

To insert data into leaf node, this method is being called. As soon as the leaf node in question runs out of space, the method split_node () (see below) is called.

split_node ()

The method split_node () splits the content of the node in question between said node and a newly created node returned by create_node (). Also, the middle entry gets pushed up into the parent node, which results in a call to insert_data_into_node ().

create_node ()

This method searches through the list of existing nodes to find a node that has not been occupied yet. If it cannot find any, then it sends a request to the data layer to make more space available for a number of nodes. The number of nodes to be reserved in addition to the already existing nodes is set by the member variable m_nGrowBySize.

remove_data_from_node ()

This method doesn't request the data layer to shrink its size per se, but instead invokes a method called auto_shrink_inc () to see if there is a chance to decrease the size of the data layer, once the remove operation has completed.

Data Layer Template Parameters

This section describes all template parameters of the data layer and data layer property classes and also explains what the limitations of those parameters are.

The data layer code has been written under the assumption that all template parameters are an unsigned integer type. Although every template parameter allows for any type being set, due to the keyword 'class', the resulting behaviour must adhere to the rules of an unsigned integer, otherwise the behaviour of the resulting solution is undefined. The reason why the keyword 'class' is in use for every template parameter is to allow for potential future extensions.

_t_sizetype (size_type)

The template parameter _t_sizetype is the type used by the container type and the data layer type to display the address of individual data entries being used by the application. A larger type allows for more entries to be addressed, but might create slower code, since a larger type has a higher chance to create CPU cache line oscillations. On the other hand, a smaller type has the opposite effects. Furthermore, setting this type to size_t ensures that iterator types of the btree-framework being used as an input to regular STL containers, will have type compatibility.

_t_nodeiter (node_iter_type)

The template parameter _t_nodeiter is the type used by the container type and the data layer type to address individual nodes in the tree structure or on the physical medium respectively. It is the data layer's responsibility to translate between the two. A too small type limits the maximum number of nodes and therewith may limit the overall maximum capacity of the container. However, a too large type may create performance issues, since extra calculations are required to handle larger types. Currently the default type is set to an unsigned 64 bit integer, which may not be ideal for 32 bit architectures. See also: Node Iterator Type (_t_nodeiter)

_t_subnodeiter (sub_node_iter_type)

The template parameter _t_subnodeiter is the type used by the container type and the data layer type to address individual sub-nodes of a node in the tree structure or on the physical medium respectively. It is the data layer's responsibility to translate between the two. A too small type limits the maximum number of addressable sub-nodes and therewith may limit the overall maximum capacity of the container. However, a too large type may create performance issues, since extra calculations are required to handle larger types. Currently the default type is set to an unsigned 32 bit integer. See also: Sub-Node Iterator Type (_t_subnodeiter)

_t_addresstype (address_type)

The template parameter _t_addresstype is the type in use by the data layer to display absolute addresses when accessing the physical medium. This type sets the maximum addressable range on the physical medium accessible at run time. Currently the default type is set to an unsigned 64 bit integer, which ought to be sufficient for any existing application. Also, this must greater or equal in terms of bit width than the _t_offsettype (offset_type) type. See also: Absolute Address Type (_t_addresstype)

_t_offsettype (offset_type)

The template parameter _t_offsettype is the type in use to display an absolute address within a block. Currently the default type is set to an unsigned 32 bit integer and in terms of bit width this type must be smaller or equal than the _t_addresstype (address_type). See also: Relative Address Type (_t_offsettype)

Target Data Layer Types and Their Property Types

The target data layers and their respective properties are working in tandem. While the properties hold the configuration, which is used by the target data layer to be set up and also provides the container class with the Size Type (_t_sizetype), Data Layer Type Definition (data_layer_type) and the Latency Property Definition (has_latency) it is going to be compiled with. Furthermore, the property instance can be copied and therefore reused by another container. This design choice has been made, since the capability to copy a target data layer instance from one container would create confusing. The programmer could think either only the configuration is copied across, which contradicts the idea of a copy constructor or the data layer’s content plus its configuration are copied, which on that level is illegal, because it is the containers task to transfer data from one container to another and not the data layer’s responsibility.

RAM Data Layer (CBTreeRAMIO)

The RAM data layer uses local memory as a physical medium to store data. This may also involve local swapping, which depends on the operating system's memory settings. The RAM data layer uses the Linear Data Layer Model as a method to order its data and uses an instance of the RAM Data Layer Properties (CBTreeIOpropertiesRAM) to get hold of its configuration when being called into existence.

RAM Data Layer Properties (CBTreeIOpropertiesRAM)

The R. hasn’t got any configuration parameters and only provides the container class with the Size Type (_t_sizetype), Data Layer Type Definition (data_layer_type) and the Latency Property Definition (has_latency) it is going to be compiled with.

File Data Layer (CBTreeFileIO)

The file data layer uses a temporary file (delete_on_close is enabled) within a file system to store data. This data layer uses the Block Data Layer Model to order its data and uses an instance of the File Data Layer Properties (CBTreeIOpropertiesFile) to get hold of its configuration when being called into existence.

File Data Layer Properties (CBTreeIOpropertiesFile)

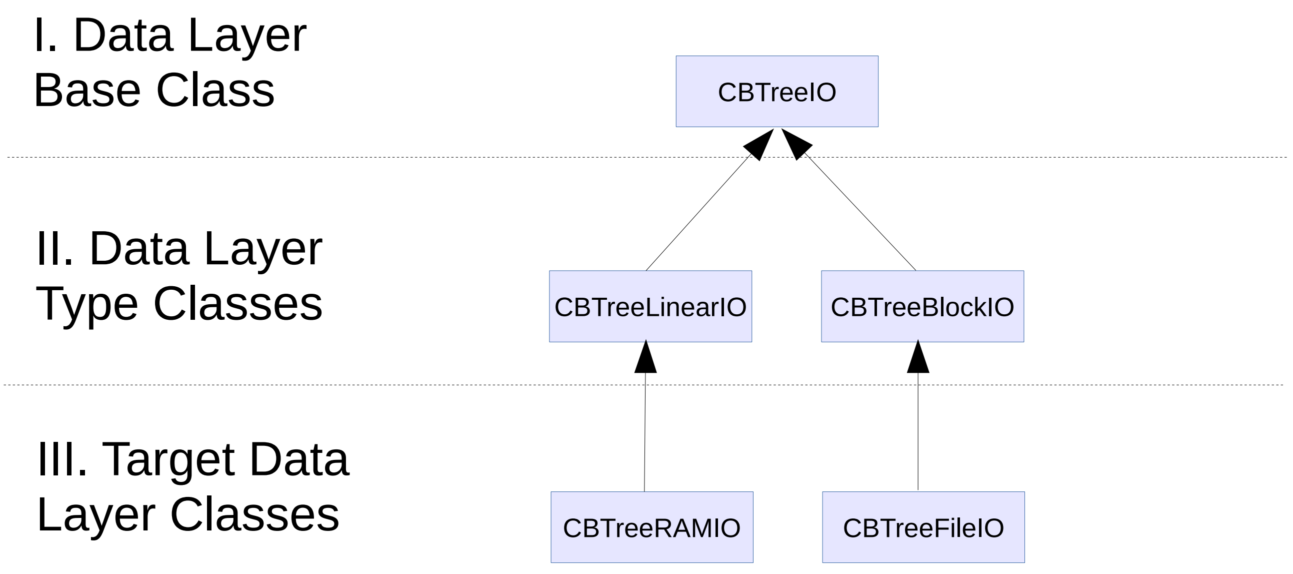
The F. are configuring the File Data Layer (CBTreeFileIO) with a pathname telling the data layer where to put the temporary file used as the data layer’s storage medium, a soft address space limit telling the maximum amount of data to be left in memory once an operation has completed as well as the optimal block size in bytes.

Pools

P. describe what data in which quantities is stored within each node. This means, in addition to the application's data also each node's associated descriptor as well as the maintenance vector, the list of sub-nodes and the serial vector is put in there. Data classes or classes inheriting from a data class can create additional pools, to optimise access by creating search vectors or any other per node data the programmer deems necessary. However, creating additional pools also means creating new code by abstracting existing methods or to write methods for accessing data tailored for specific pools. If pools have been created to optimise data access or at least the way data is being accessed, then see section Methods Utilising Pools for the list of methods that a new data class needs its own version of.

Container To Data Layer Interface

This section gives a brief overview how the data layer hierarchy is structured and how it is working internally. Target data layer types are the visible part of the data layer hierarchy for a calling container instance and have to be interface agnostic for said container. Currently the data layer hierarchy looks as follows:



The base data layer type provides access to a storage medium via the interfaces described in section Construction and Specific Base Methods and Abstract Base Class Methods. Whereas the data layer type classes provide methods for the target data layer classes to translate addresses from container requests to the physical medium and vice versa. Those classes describe the data layer model being used by the target data layer classes. And finally the target data layer types are those which are instantiated by a container instance and provide the actual functionality for the methods displayed by sections section Construction and Specific Base Methods and Abstract Base Class Methods.

Base Data Layer Interface

The base data layer type provides access to a storage medium via the interfaces described in the two sections below and manages the Pools of the container instance.

Construction and Specific Base Methods

This section contains descriptions of the data layer type definitions, a generic target data layer constructor and specific base methods. The Type Definitions are to be used by the container type to specify what type have to be used in order to display the dimensions of the container instance. The section Abstract Data Layer Construction displays said generic constructor, one of which every target data layer has to provided by any target data layer class for container types to instantiate a target data layer while being agnostic about its type.

The two methods get_pooledData () and insert_dataIntoPool () have to be provided by each target data layer type although they ought to be abstract. This is due to the fact that both methods involve a template parameter in their declaration and a limitation in C++, which cannot allow for virtual template methods, since templates are resolved at compile time and virtual calls at run time. To avoid redundancy in the API sections of each target data layer, they are being listed as Specific Base Methods.

Type Definitions

Type	Definition	Description
data_layer_properties_type	_t_datalayerproperties	This defines what is used to define all data layer dimension types and will contain the data layer's configuration once instantiated.
size_type	typename data_layer_properties_type::size_type	This defines what type is used to address individual data entries stored within the container.
node_iter_type	typename data_layer_properties_type::node_iter_type	This defines what type is used to address individual nodes to display the container's tree structure.
sub_node_iter_type	typename	This defines what type is used

Type	Definition	Description
	data_layer_properties_type::sub_node_iter_type	to address a node's individual sub data entries to display the container's tree structure.
address_type	typename data_layer_properties_type::address_type	This defines what type is used to display absolute addresses for the entire range of the data layer.
offset_type	typename data_layer_properties_type::offset_type	This defines what type is used to display addresses within a block.
data_layer_type	typename data_layer_properties_type::data_layer_type	This type determines the data layer type that will be in use.

Constructor

```
CBTreeIO<_t_datalayerproperties>
(
        const uint32_t nNumDataPools,
        const CBTreeIOperBlockPoolDesc_t *psDataPools
)
```

Description:

This constructs the basic part of a data layer instance.

Input parameters:

nNumDataPools	- specifies the number of data pools to be set up
psDataPools	- specifies pointer to array containing pool set up information

Remarks:

The base constructor takes a copy of all pool descriptions (Pools) and returns.

The parameters nNumDataPools and psDataPools are the number of pools to be created per node and the pointer to the pool descriptor array of size nNumDataPools respectively. Please see section Pools to find out what pools are and please see section Pools (Technical Reference) to find out how a data class has to set up pool descriptors to be used by a data layer class.

get_pooledData ()

1)
```
_t_dl_data *get_pooledData<_t_dl_data> (const uint32_t nPool, const node_iter_type nNode,
const sub_node_iter_type nEntry)
```
2)
```
_t_dl_data *get_pooledData<_t_dl_data> (const uint32_t nPool, const node_iter_type nNode)
```

Description:

These method returns the pointer to a data item of a specific pool, node and optional sub-entry.

Input parameters:

nPool	- specifies pool ID
nNode	- specifies node ID
nEntry	- specifies the data item offset within node

Remarks:

These methods enable access to a data item or list of data items by returning a pointer within the address space of the application. Said pointer refers to the base of the data item or data items associated with the specified pool, node and optional entry (method 1). The referred to address space is read / write enabled, so that if modifications need to be made a write back operation is not required. Since these methods enable access to any pool, this also involves custom data items to be returned to the calling application. It is strongly suggested that the calling application only obtains a copy of the custom data for two reasons. Firstly, it would enable the application to write directly to the data layer, which is dangerous, because the application is hardly aware of the contained b-tree structure or how data is organised on data layer level. Secondly, once an operation has completed and the application has the focus again, it is not guaranteed that the location being pointed at is still valid.

Return value:

The return value is a pointer of type _t_dl_data, pointing to the base of the requested data item.

Note:

nEntry must be less than the pool's maximum size, otherwise the result is undefined.

insert_dataIntoPool ()

```
void insert_dataIntoPool<_t_dl_data> (const uint32_t nPool, const node_iter_type nNode, const
sub_node_iter_type nNodeLen, const sub_node_iter_type nOffset,
const sub_node_iter_type nDataLen)
```

Description:

This method creates space in a pool of a specific node.

Parameters:

nPool	- specifies pool ID
nNode	- specifies node ID
nNodeLen	- specifies the number of data items stored in node before the insertion
nOffset	- specifies before which data item the new data is to be inserted
nDataLen	- specifies the number of items to be inserted

Remarks:

This method takes all data within pool nPool of node nNode beyond nOffset and moves it back by nDataLen items. This results in a gap at the location of nOffset, which is left open to be filled by a move construction or move assignment later. The pool in question then results in having the size worth of nNodeLen and nDataLen entries combined.

Note:

nOffset must not exceed nNodeLen, otherwise the result is undefined.

nNodeLen must not exceed the pool size, otherwise the result is undefined.

nNodeLen and nDataLen combined must not exceed the pool size, otherwise the result is undefined.

Pure Abstract Base Class Methods

The methods container by this section are all pure virtual methods, which means, a target data layer class needs to provide its own version of these.

set_size ()

void set_size (const node_iter_type nMaxNodes)

Description:

This method is used when the data layer size needs to be changed.

Parameters:

nMaxNodes	- specifies the least amount of total nodes that must be available after this call

Remarks:

This method guarantees when it returns, that at least nMaxNodes nodes can be stored via the data layer. In case the method fails an exception of type std::runtime_error is thrown.

Note:

Calling this method frequently will result in performance issues.

unload ()

void unload ()

Description:

This method writes all cached and buffered data back to the physical medium.

Remarks:

When calling this method all caches and buffers are flushed. In addition to that every resource the data layer uses, such as allocated memory, is destroyed. Only the bare minimum to recover on the next call is left. This is useful if the environment has scares resources and it is known that this data layer is not needed for a long time. On the next method being called using this data layer instance caches, buffers and any other resource required will be reinstated.

Note:

Calling this method frequently will result in performance issues.

unload_from_cache ()

```
void unload_from_cache (const node_iter_type nNode)
```

Description:

This method writes back one node to the physical medium.

Parameters:

nNode	- specifies node

Remarks:

This method writes back every cached pool of node nNode and invalids the respective cache lines.

Note:

This method is for debugging purposes only and ought not to be called during normal operations.

is_dataCached ()

```
bool is_dataCached (const uint32_t nPool, const node_iter_type nNode) const
```

Description:

This method determines if a pool of a node is cached.

Parameters:

nPool	- specifies pool ID
nNode	- specifies node

Remarks:

To determine if a specific pool is currently cache, this method can be used.

Return value:

The return value is true if the pool in question is cache, otherwise false.

Note:

This method is for debugging purposes only and ought not to be called during normal operations.

showdump ()

```
void showdump (std::ofstream &ofs, const node_iter_type nTreeSize) const
```

Description:

This method writes the content of the data layer to an output stream in HTML format.

Parameters:

ofs	- reference to output stream
nTreeSize	- specifies the number of total nodes available via the data layer

Remarks:

This method outputs two parts into the output stream. Part one is a list of all registered pools and their respective settings. Part two is a list of all block containing their nodes containing their pools. Every pools displays its raw content in hexadecimal format.

Note:

This method is for debugging purposes only and ought not to be called during normal operations.

Abstract Base Class Methods

All methods displayed by this section are virtual but have their default code. This means that, the option to change the behaviour of those methods exists.

set_root_node ()

```
void set_root_node (const node_iter_type nRootNode)
```

Description:

This method tells the data layer, which node is root.

Parameters:

nRootNode	- specifies root node

Remarks:

This method tells the data layer, which node is root and to optimise accesses ought not to be unloaded. Data layers following the Block Data Layer Model, have to decide what data has to stay resident and what to unload. Those data layers may use that information to support any container class in terms of access speed. For data layers following the Linear Data Layer Model, this method is ineffective, since all data remains available at any time.

terminate_access ()

```
void terminate_access ()
```

Description:

This method tells the data layer that an operation has completed.

Remarks:

Some data layers use a resource management system to keep resources being in use restricted.

Resources made available to the container classes on any level must stay resident for the duration of an access, so that references given to the accessing container instance are remaining valid. To tell data layer that some of the resources can be released again this method has to be called, when an operation is about to return to the calling application.

Base Class Methods

The methods displayed in the sub-sections below don't require a target data layer class to provide or alter their code.

get_maxNodes ()

```
node_iter_type get_maxNodes () const
```

Description:

This method returns the maximum number of nodes the data layer is able to currently store.

Remarks:

The data layer has the potential to fragment or require more space than needed due to the following situations:

- a node is removed leaving an unallocated spot
- space for an additional node is not available and therefore space for a number of new nodes is allocated

In order to determine what the maximum number of nodes, including all allocated and unallocated nodes, present on the data layer, this method can be called.

Return value:

The return value is the maximum number of linear nodes in accordance with the space available on the data layer.

get_dataAlignment ()

```
offset_type get_dataAlignment () const
```

Description:

This method returns the byte alignment for pools within a node.

Remarks:

To determine what the optimal alignment for the local architecture is, this method can be called. Currently 4 and 8 are returned on 32 and 64 bit architectures, respectively.

Return value:

This method returns the byte alignment employed to store pools within a node.

get_alignedOffset ()

```
offset_type get_alignedOffset (const offset_type nOffset) const
```

Description:

This method rounds an offset up to the next optimal byte alignment.

Parameters:

nOffset	- specifies the byte offset to be rounded up

Remarks:

This method takes the input parameter nOffset and returns the next higher offset divisible by the optimal byte alignment, based on the result of method get_dataAlignment ().

Return value:

The value returned by this method is the next higher offset divisible by the optimal byte alignment.

set_cacheFreeze ()

```
void set_cacheFreeze (const bool bEnabled)
```

Description:

This method sets the data layer to read through cache mode.

Parameters:

bEnabled	- enables or disables cache freeze mode

Remarks:

This method tells the data layer to not to load anything into the cache when reading data and makes it obtaining data directly from the physical medium, if bEnabled is set. However, when data is written, caches remain effective.

s. ought not to be called during normal operations, since it is for debugging purposes and offers some room for optimisation, in case an access has to pollute the cache. This means that, if a method, such as rebuild_node () doing a full rebuild, has to successively access a number of pool entries, then the pool cache gets populated with data, of which the majority won't be used again any time soon. Furthermore, the cache then needs to be re-populated with frequently used data. The described process is very expensive in terms of processing power and can be avoided by freezing caches for the time being. By doing this, any read request is pushed to the underlying data medium, which is most likely outfitted with a cache being better prepared for larger linear reads.

Note:

This method is for debugging purposes only and ought not to be called during normal operations.

get_pool_entry_size ()

```
uint32_t get_pool_entry_size (const uint32_t nPool) const
```

Description:

This method returns the byte size of one data item of a specific pool.

Parameters:

nPool	- specifies the pool ID

Remarks:

This method returns the number of bytes required to store one data item within the specified pool.

Return value:

The returned value is the byte size of one data item of a specific pool.

get_pool_total_size ()

```
uint32_t get_pool_total_size (const uint32_t nPool) const
```

Description:

This method returns the byte size of maximum number of data items of one node within a specific pool.

Parameters:

nPool	- specifies the pool ID

Remarks:

This method returns the number of bytes required to store one node's maximum number of data items within a specific pool.

Return value:

The returned value is the maximum byte size of one node within a specific pool.

Data Layer Type Specific Interfaces

This section contains data layer type specific methods, which are translating addresses and requests between the Target Data Layer Specific Interfaces and the underlying physical medium.

Linear Data Layer Methods

The L. provide the functionality to translate any node / pool request to the Linear Data Layer Model and back in order to access the physical medium.

Constructor

```
CBTreeLinearIO<data_layer_properties_type>
(
	const sub_node_iter_type nNodeSize,
	const uint32_t nNumDataPools,
	const CBTreeIOperBlockPoolDesc_t *psDataPools
)
```

Description:

This constructs a data layer class following the Linear Data Layer Model.

Input parameters:

nNodeSize	- specifies node size parameter
nNumDataPools	- specifies the number of data pools to be set up
psDataPools	- specifies pointer to array containing pool set up information

Remarks:

The linear data layer constructor allocates an array of pool base pointers and sets each of those to an invalid value. The number of base pointers created depends on the input parameter nNumDataPools and they are later on used to serve as a reference for their respective pool. What those pointers will eventually point at depends on the actual data layer in use, since this constructor is only initialising the linear data layer model part.

get_node_base ()

```
const uint8_t   *get_node_base (const uint32_t nPool, const node_iter_type nNode) const
```

Description:

This method generates the base address of a specific node within a selected pool.

Parameters:

nPool	- specifies pool ID
nNode	- specifies node ID

Remarks:

This method returns the base address of the node in question within the specified pool, while the returned base address is within the address space of the application.

Return value:

The return value is the pointer to the base of the specified node / pool pair.

Block Data Layer Methods

The L. provide the functionality to translate any node / pool request to the Block Data Layer Model and back in order to access the physical medium. For access optimisation purposes, the method set_root_node () provides its own code in case the Block Data Layer Model is in use.

Constructor

```
CBTreeBlockIO<data_layer_properties_type>
(
	const sub_node_iter_type nNodeSize,
	const uint32_t nNumDataPools,
	const CBTreeIOperBlockPoolDesc_t *psDataPools
)
```

Description:

This constructs a data layer class following the Block Data Layer Model.

Input parameters:

nNodeSize	- specifies the node size parameter
nNumDataPools	- specifies the number of data pools to be set up
psDataPools	- specifies pointer to array containing pool set up information

Remarks:

The construction of the block data layer instance consists of two parts:

1. The constructor takes a copy of parameter nNodeSize and every pool descriptor to prepare the data layer instance derived from this class for accesses. Also generate_pool_offsets () is called and based on the input parameter nNodeSize generates a fast look-up table in order to optimise future accesses.
2. Once this constructor has completed and the execution has returned to the target data layer constructor, the method setup () must be called to finalise the set up. The reason why this is separate, is to give the target constructor a chance to alter run-time variables. Currently the only variable that needs modifying by the target constructor is the block size variable (m_nBlockSize).

As a result, the construction of a data layer instance looks as follows:

1. target constructor is called
    - CBTreeIO constructor is executed
2. target constructor re-determines m_nBlockSize
3. setup () is called by target constructor
4. target constructor continuous

The parameter nNodeSize determines the maximum number of data items stored per node as well as the maximum number of sub-nodes one node can link to and in a further sense how large each node is going to be when stored using the data layer. In other documentations this parameter is often referred to as “t”. See below how the maximum number of data items and sub-nodes are being determined:

- minimum number of data items is t – 1
    - except for when the root node is also a leaf node → in that case it is 0
- maximum number of data items is 2 * t - 1

- maximum number of sub-nodes is 2 * t

get_blockAddr ()

```
address_type get_blockAddr (const node_iter_type nNode) const
```

Description:

This method translates a node ID to the linear address of its respective block address the node's data is stored in.

Input parameter:

nNode	- specifies node ID

Remarks:

To find the linear base of the block containing the specified node ID, this method has to be used. It aids the process of finding the linear address of a node or pool within a node and is usually called by other address generator methods. Since this method is part of the address generation, it won't sanity check the input parameter and just act as a function to its input parameter.

Return value:

The return value is the linear base address of the block the specified node is within.

get_poolOffset ()

```
offset_type get_poolOffset () const
```

Description:

This method returns the number of bytes padded in before each node.

Remarks:

When the pool layout per node is generated by the method generate_pool_offsets (), then this method is employed to tell at which byte offset the initial pool starts within every node. If a data layer needs to have a padding prior to each node, it must provide its own version of this method. In general this is not required and therefore this method returns zero by default.

Return value:

The return value is the number of bytes padded in prior to each node.

get_node_offset ()

```
address_type get_node_offset (const node_iter_type nNode) const
```

Description:

This method translates a node ID to the offset relative to the start of the block the node is within.

Input parameter:

nNode	- specifies node ID

Remarks:

To find the offset of a specific node within a block, this method has to be called. Since this method is part of the address generation, it won't sanity check the input parameter and just act as a function to its input parameter.

Return value:

The return value is the byte offset of the specified node within the block it is container by.

get_nodeAddress ()

```
address_type get_nodeAddr (const node_iter_type nNode) const
```

Description:

This method translates a node ID to its linear address.

Input parameter:

nNode - specifies node ID

Remarks:

To find the linear address of a specific node, this method has to be called. Since this method is part of the address generation, it won't sanity check the input parameter and just act as a function to its input parameter.

Return value:

The return value is the linear base address of the specified node.

get_per_node_pool_offset ()

```
offset_type get_per_node_pool_offset (const uint32_t nPool) const
```

Description:

This methods returns the byte offset of a selected pool within a node.

Input parameter:

nPool - specifies pool ID

Remarks:

To convert a pool ID into the byte offset that pool has in a node, this method must be called. It uses the selected pool to find the offset via a look up table and returns it. If the input parameter exceeds the number of available pools, then an exception is thrown.

Return value:

The return value is the byte offset of the selected pool.

get_pool_address ()

```
address_type get_pool_address (const uint32_t nPool, const node_iter_type nNode) const
```

Description:

This methods returns the linear address of a selected pool and node.

Input parameter:

nPool	- specifies pool ID
nNode	- specifies node ID

Remarks:

To convert a node with selected pool therein into its linear address on the data layer, this method must be called. The methods get_nodeAddress () and get_per_node_pool_offset () are called to respectively generate the node's linear address and the pool offset of a node and finally combines the two results. If the input parameter nPool exceeds the number of available pools, then an ::std::runtimeerror exception is thrown. The input parameter nNode is not sanity checked and just acts like an input parameter to a function.

Return value:

The return value is the linear address of a selected pool within a selected node.

generate_pool_offsets ()

```
offset_type generate_pool_offsets ()
```

Description:

This method populates the pool offset lookup table.

Remarks:

This method is called during the construction and populates the pool offset lookup table involving the padding prior to the initial pool returned by get_poolOffset (). Any further pool offset is set up by calculating the end of the preceding pool plus a padding large enough to make the current pool's offset divisible by the return value of get_dataAlignment (). The return value of this method is the final pool's end offset combined with the same padding that is trailed by any of the previous pools. If any of the pools sizes is already a multiple of the return value of get_dataAlignment (), then the respective padding following that pool has zero length.

Return value:

The return value is the size of one node involving all padding prior, trailing and in between pools there are.

realloc_descriptor_vector ()

```
void realloc_descriptor_vector (const node_iter_type nMaxNodes)
```

Description:

This method allocates or re-allocates a description vector large enough to accommodate at least the number of nodes specified.

Input parameter:

nMaxNodes	- specifies the number of nodes the descriptor vector must be able to address

Remarks:

This method makes sure the descriptor vector has sufficient space to accommodate a large enough number of descriptors for blocks or block groups to contain at least the specifies amount of nodes once this call returns. If the descriptor vector wasn't called into existence before, then it is created and populated with entries set to an invalid state, otherwise the existing description vector is resized. In case of a resize, it is being tested whether the resulting vector has the same, a larger or a smaller size as before. If the resulting size is the same, then the call is ineffective and the method returns immediately. An increase in size creates a number of new descriptors, which are initialised with an invalid state, as oppose to a reduction in size. In that case a number of descriptors are dropped. Those descriptors must have been de-initialised prior to this method, otherwise they are lost. If it is not possible to allocate or re-allocate space large enough to contain the required number of descriptors, then an ::std::runtimeerror exception is thrown.

convert_node_to_descriptor ()

```
uint32_t convert_node_to_descriptor (const node_iter_type nNode, const bool bRoundUp = false)
const
```

Description:

This method converts a node ID to its descriptor.

Input parameter:

nNode	- specifies node ID
bRound	- selects if this method is in conversion or size calculation mode

Remarks:

This method has two purposes. If the parameter bRound is set to false, then the method converts the specified node ID to the descriptor number associated with the block the node is stored in. In case bRound is set to true, then the input parameter is interpreted as the amount of nodes that need to be at least available after a resize operation and the returned value is the required number of descriptors to fulfil said requirement.

Return value:

The return value is the descriptor of the block the input node is stored, if bRound is false, otherwise it is the minimum amount of descriptors to have at least the specified number of nodes available.

setup ()

```
void setup (const address_type nBlockSize)
```

Description:

This method sets up internal variables of the base data layer class.

Input parameters:

nBlockSize	- specifies the ideal block size of the data layer in question

Remarks:

During its construction a data layer instance determines its ideal block size, which happens after the base data layer constructor has been executed. (See: Constructor) This means that the base data layer initialisation is incomplete and the member variables below still need to be set up as explained.

- m_nBlockSize
  - This is the actual block size to be used, which can be a multiple of the input parameter and is expanded increasing the multiplier until at least one node fits.
- m_nNodesPerBlock
  - This is the number of nodes stored per block and is always part of the 2 to the power n series to optimise access speed.
- m_nNodesPerBlockVectorSize
  - This is an internal parameter used to calculate addresses fast and contains the number of bits used access a node within one block. It is determined as follows: n = log (m_nNodesPerBlock) / log (2)

increment_access_counter ()

```
void increment_access_counter (const uint32_t nDescriptor) const
```

Description:

This method controls a descriptor's access counter incrementation.

Input parameter:

nDescriptor	- selects descriptor ID for incrementation

Remarks:

As part of the resource management, every data layer following the Block Data Layer Model counts the number of accesses per descriptor. This method is called to manage that incrementation and also to handle situations when an access counter of descriptor exceeds a threshold value displayed by m_nMaxAccessCtr. Once an access counter is exceeding its threshold value, the access counter of every valid descriptor is divided by two. Since the access counters are used as statistical information by terminate_access () to decide what descriptors to unload as soon as an operation has been completed, the reduction of the access counters is being done to prevent perverted situations. Those situations are, where one or more access counters become very large and also irrelevant for the majority of the accesses, but aren't being unloaded. As a result more frequently used descriptors aren't staying resident and the calling application would see long call times resulting in performance issues.

Target Data Layer Specific Interfaces

This section contains all target data layer methods, may those be protected or not. Protected methods cannot be used by the container instance and are described in the sub-sections in case future expansions of the code are required. If a list of container to target data layer interface methods is needed, then see below:

- get_pooledData ()
- insert_dataIntoPool ()
- get_maxNodes ()
- get_dataAlignment ()
- get_alignedOffset ()
- get_pool_entry_size ()
- set_size ()
- unload ()
- set_root_node ()
- terminate_access ()
- unload_from_cache ()
- set_cacheFreeze ()
- is_dataCached ()
- showdump ()

RAM Data Layer Methods

The R. provide code for the following methods, which already have been described:

- get_pooledData ()

- insert_dataIntoPool ()
- set_size ()
- unload ()
- unload_from_cache ()
- is_dataCached ()
- showdump ()

Constructor

```
CBTreeRAMIO<data_layer_properties_type>
(
      const data_layer_properties_type &rDataLayerProperties,
      const sub_node_iter_type nNodeSize,
      const uint32_t nNumDataPools,
      const CBTreeIOperBlockPoolDesc_t *psDataPools
)
```

Description:

This constructs a RAM data layer instance to be used by a container instance of this framework.

Input parameters:

rDataLayerProperties	- specifies data layer set up parameters
nNodeSize	- specifies the node size through-out the tree structure
nNumDataPools	- specifies the number of data pools to be set up
psDataPools	- specifies pointer to array containing pool set up information

Remarks:

This constructor calls the linear data layer Constructor first and then takes a copy of the input properties for later use.

File Data Layer Methods

The F. provide code for the following methods, which already have been described:

- get_pooledData ()
- insert_dataIntoPool ()
- set_size ()
- unload ()
- unload_from_cache ()
- is_dataCached ()
- showdump ()

Constructor

```
CBTreeFileIO<data_layer_properties_type>
(
	const data_layer_properties_type &rDataLayerProperties,
	const sub_node_iter_type nNodeSize,
	const uint32_t nNumDataPools,
	const CBTreeIOperBlockPoolDesc_t *psDataPools
)
```

Description:

This constructs a file data layer instance to be used by a container instance of this framework.

Input parameters:

rDataLayerProperties	- specifies data layer set up parameters
nNodeSize	- specifies the node size through-out the tree structure
nNumDataPools	- specifies the number of data pools to be set up
psDataPools	- specifies pointer to array containing pool set up information

Remarks:

This constructor calls the block data layer Constructor first, then takes a copy of the input properties for later use, continues by determining the ideal block size. The next step is to create a temporary file which is being deleted on close. Said file is then initialised with a length worth of one block, since some system return with an error if mapping is being attempted on a zero length file. Lastly, the mapping is initialised by calling init_mapping ().

init_mapping ()

```
void init_mapping ()
```

Description:

This method initialises the mapping of the file to be mapped.

Remarks:

To prepare a file to be mapped into the application's memory space, this method is called. Some system have the requirement for that additional step and for those systems a mapping handle is created to be used for the mapping. Other system don't need that extra layer and for those systems this method is empty and thus ineffective. This method is called during the construction of this data layer instance and also when the mapping is re-initialised after a resize took place.

If the file to be mapped was not called into existence, the file was already prepared for mapping or the process of creating a mapping handle was not successful, then an ::std::runtimeerror exception is thrown.

exit_mapping ()

```
void exit_mapping ()
```

Description:

This method de-initialises the mapping of the file that was mapped.

Remarks:

To shut down the mapping of a file that was used as physical back end to store its associated container's data, this method is called. Some system have the requirement for that additional step and for those systems the previously created mapping handle is closed. Other system don't need that extra layer and for those systems this method is empty and thus ineffective. This method is called during the destruction of this data layer instance and also when the mapping has to be de-initialised before a resize takes place.

If the mapping was already de-initialised or closing the mapping handle was not successful, then an ::std::runtimeerror exception is thrown.

map_descriptor ()

```
void map_descriptor (const uint32_t nDescriptor)
```

Description:

This method initialises the mapping of one block or block group.

Input parameter:

nDescriptor	- specifies the descriptor ID

Remarks:

This method maps a block or a block group associated with the given descriptor to the application's address space as well as resetting the descriptor's access counter. Said blocks are read / write enabled and therefore neither the data layer nor the data container have to make any afford in terms of copy-on-write approaches when modifying data. Also, the blocks are guaranteed to stay resident until terminate_access () is being called.

In the event the mapping was not successful an ::std::runtimeerror exception is thrown.

sync_descriptor ()

```
void map_descriptor (const uint32_t nDescriptor)
```

Description:

This method flushes the descriptor to the underlying physical medium.

Input parameter:

nDescriptor	- specifies the descriptor ID

Remarks:

This method asynchronously writes the content the descriptor set by nDescriptor back to the physical medium and flags it as to be unmapped. Since this method is non-blocking, a call to unmap_descriptor () using the same descriptor is required to synchronise the access again. In order to optimise mass write backs when terminate_access () or unload () is being called, this is useful to mark descriptors for unloading, while leaving it to the operating system to determine the optimal order.

unmap_descriptor ()

```
void unmap_descriptor (const uint32_t nDescriptor)
```

Description:

This method de-initialises the mapping of one block or block group.

Input parameter:

nDescriptor	- specifies the descriptor ID

Remarks:

This method un-maps a block or a block group associated with the given descriptor.

In the event the un-mapping was not successful an ::std::runtimeerror exception is thrown.

unmap_all_descriptors ()

```
void unmap_all_descriptors (const bool bExceptRoot)
```

Description:

This method un-maps all mapped blocks.

Input parameter:

bExceptRoot	- selects if the root node containing block stays resident or not

Remarks:

This method is executed when unload () is called, since system's resources are getting scares or the container instance won't in use for a very long time or when set_size () is called to have the data layer change in size. In case the data layer is unloaded all but the root node containing descriptor are unloaded, since that node will definitely be in use again. To achieve that the parameter bExceptRoot is set to true. If the data layer needs to be resized, then all resources must be de-initialised and bExceptRoot is set to false.

Abstract Data Layer Construction

This section displays the construction of any target data layer as it is being constructed by a container instance. Hence, the description below is using abstract terms, such as data_layer_type, instead of a specific target data layer type. The data layers to be instantiated by a container can be found in section Target Data Layer Types and Their Property Types.

```
data_layer_type
(
	const data_layer_properties_type &rDataLayerProperties,
	const sub_node_iter_type nNodeSize,
	const uint32_t nNumDataPools,
	const CBTreeIOperBlockPoolDesc_t *psDataPools
)
```

Description:

This is a generic data layer constructor as it is called by the CBTreeBaseDefaults class (Specific Default Container to b-tree Methods (CBTreeBaseDefaults)).

Parameters:

rDataLayerProperties	- specifies the data layer properties / parameters which the data layer is to be set up with (see: Data Layer Properties (_t_datalayerproperties))
nNodeSize	- specifies the node size through-out the tree structure
nNumDataPools	- specifies the number of data pools to be set up
psDataPools	- specifies pointer to array containing pool set up information

Remarks:

When a data layer is called into existence by a b-tree container instance, this constructor is called to initialise the instance. While the parameter type of rDataLayerProperties is a template parameter and contains the configuration specific to the data layer in use, all other parameters are generic and are explained below:

The parameter nNodeSize determines the maximum number of data items stored per node as well as the maximum number of sub-nodes one node can link to and in a further sense how large each node is going to be when stored using the data layer. In other documentations this parameter is often referred to as “t”. See below how the maximum number of data items and sub-nodes are being determined:

- minimum number of data items is t – 1
  - except for when the root node is also a leaf node → in that case it is 0
- maximum number of data items is 2 * t - 1
- maximum number of sub-nodes is 2 * t

The parameters nNumDataPools and psDataPools are the number of pools to be created per node and the pointer to an array of size nNumDataPools respectively. Please see section Pools to find out what pools are and please see section Pools (Technical Reference) to find out how a data class has to set up for them to be used by a data layer class.

API Reference Guide

This section provides a description of any method contained by the base, array and key sort data class as well as their respective constructors.

CBTree API

This section contains descriptions about the most basic API methods, which are always available regardless to what degree an application class has been derived from this base class. Also, methods shown here won't be re-iterated in the data classes or STL compatible data class sections.

CBTreeIf Constructor

```
1)
CBTreeIf<_t_data, _t_sizetype> ()
2)
CBTreeIf<_t_data, _t_sizetype> (const CBTreeIf<_t_data, _t_sizetype> &rContainer)
3)
CBTreeIf<_t_data, _t_sizetype> (CBTreeIf<_t_data, _t_sizetype> &&rRhsContainer)
```

Description:

These constructors initialise a container on the most interface abstract level.

Input parameter:

rContainer	- specifies the reference to a container to be copied
rRhsContainer	- specifies the right hand side reference to the container to be moved

Remarks:

Since this class acts as an abstract interface, as described in section Abstract Container Interface (CBTreeIf), these constructors are only calling the constructor of Super Class (CBTreeSuper) and return immediately.

get_performance_counters ()

```
void get_performance_counters (uint64_t (&rHitCtrs)[], uint64_t (&rMissCtrs)[])
```

Description:

This method returns all cache hit and miss counters.

Output parameters:

rHitCtrs	- reference to array in which all hit statistics will be stored
rMissCtrs	- reference to array in which all miss statistics will be stored

Remarks:

To identify bottlenecks caused by any cache, this method can be used. Both arrays are being filled with the hit and miss cache statistics on any level from the data class down to the data layer. The arrays must be able to hold a size of at least PERFCTR_TERMINATOR entries. Each of those arrays contains the following elements in the current implementation:

PERFCTR_SUPER_BLOCK_ADDR	super block address generator TLB *
PERFCTR_BLOCK_ADDR	block address generator TLB *
PERFCTR_RESERVATION_VECTOR_ADDR	reservation vector address generator TLB *
PERFCTR_NODE_ADDR	node address generator TLB *
PERFCTR_SUBNODE_ADDR	sub-node data address generator TLB *
PERFCTR_SERIAL_VECTOR_ADDR	serial vector address generator TLB *
PERFCTR_DATA_ADDR	data address generator TLB *
PERFCTR_FILE_CACHE	file cache of internal CFileMem instance **
PERFCTR_NODEDATA_DATA	node information data cache ***
PERFCTR_SUBNODE_DATA	sub-node information data cache ***
PERFCTR_DATA	top level data cache ***

TLB stands for translation look-a-side buffer.

* obsolete

** only if file data layer in use, otherwise always zero

*** element has not yet been updated to pool infrastructure

Note:

For this method to work as described above, the code must be compiled with the compile flag USE_PERFORMANCE_COUNTERS being set.

The current implementation of this method dates back to a code version where pools didn't exist and the any address generation was optimised by using TLBs.

The implementation is likely to be changed towards a model where the arguments fit the pool infrastructure. Hence, programmers are advised not to use this method as it is.

Resetting the counters has not been implemented, yet.

empty ()

```
bool empty () const
```

Description:

To find out if a b-tree instance contains any data, this method may be employed.

Remarks:

This method tells the caller if the container instance has any data present. Alternatively size () may be called to see the number of existing data entries contained by a b-tree instance.

Return value:

This method returns true if this b-tree instance contains no data, otherwise false.

Note:

This method works even if the data layer has not been initialised. If that is the case, then the return value is true, since it is assumed the b-tree is empty, instead of being invalid.

size ()

```
size_type size () const
```

Description:

This methods returns the number of data items stored in the container instance.

Remarks:

To obtain the total number of data items that have been stored in the data container, this method has to be used. If the data layer has been initialised, then the method reads the maxIdx value from the root node descriptor and returns it, otherwise zero is returned. This means that, if the data layer has not been called into existence yet, this is not deemed as an invalid case and only means the container is empty.

Return value:

This method returns the number of data entries present in this container instance.

clear ()

```
void clear ()
```

Description:

This method removes all data from the b-tree at once.

Remarks:

The method de-initialises the container's data layer, which is also true is only the root node exists while it doesn't contain any data. Also internal counters and pointers are reset to allow for a re-instantiation of a new data layer.

Note:

If this method is called while the data layer has not been initialised or was de-initialised before, then it is ineffective.

begin ()

1)
iterator begin ()

2)
const_iterator begin () const

Description:

These method return an iterator, pointing at the initial data entry.

Remarks:

Method (1) instantiates a read-write iterator pointing at the first data entry, whereas method (2) instantiates a read only iterator. In either case, the returned iterator has not been evaluated, which means calling this method does not result in a penalty in terms of CPU time, in case further arithmetic operations of that newly created iterator follow the creation.

Return value:

The return value is a read-write iterator pointing at the first data entry for method (1) and a read only iterator for method (2). In case the data container is empty, the returned iterator points to an invalid object, which, if accessed in that state, would result in an ::std::out_fo_range exception being thrown.

end ()

1)
iterator end ()

2)
const_iterator end () const

Description:

These method return an iterator, pointing at the data entry past the final entry.

Remarks:

Method (1) instantiates a read-write iterator pointing at the next data entry past the final entry, whereas method (2) instantiates a read only iterator. In either case, the returned iterator can not be evaluated, since the external pointer of the returned iterator has been set to the return value of size

(), which always points at an invalid object.

If it is not known what an external pointer in terms of iterators is, then please see section: Iterators

Return value:

The return value is a read-write iterator pointing at the next data entry past the last entry, for method (1) and a read only iterator for method (2).

rbegin ()

```
1)
reverse_iterator rbegin ()
2)
const_reverse_iterator rbegin () const
```

Description:

These methods return a reverse iterator, pointing at the final data entry.

Remarks:

Method (1) instantiates a reverse read-write iterator pointing at the final data entry, wherease method (2) instantiates a reverse read only iterator. In either case, the returned iterator has not been evaluated, which means calling this method does not result in a penalty in terms of CPU time, in case further arithmetic operations of that newly created iterator follow the creation.

Return value:

The return value is a reverse read-write iterator pointing at the final data entry for method (1) and a read only reverse iterator for method (2). In case the data container is empty, the returned iterator points at an invalid object, which the object prior the non-existing initial entry. If the iterator is accessed in that state, an ::std::out_of_range exception is thrown.

rend ()

1)
```
reverse_iterator rend ()
```
2)
```
const_reverse_iterator rend () const
```

Description:

These methods return a reverse iterator, pointing at the data entry before the initial entry.

Remarks:

Method (1) instantiates a reverse read-write iterator pointing at the data entry prior the first entry, whereas method (2) instantiates a reverse read only iterator. In either case, the returned iterator can not be evaluated, since the external pointer of the returned iterator has been set to -1 (~0x0), which always points at an invalid object.

If it is not known what an external pointer in terms of iterators is, then please see section: Iterators

Return value:

The return value is a reverse read-write iterator pointing at the data entry prior the first data entry for method (1) and a reverse read only iterator for method (2).

cbegin ()

```
const_iterator cbegin () const
```

Description:

This method returns a read-only iterator, pointing at the initial data entry.

Remarks:

This method instantiates a read-only iterator pointing at the first data entry. The returned iterator has not been evaluated, which means calling this method does not result in a penalty in terms of CPU time.

Return value:

The return value is a read-only iterator pointing at the first data entry. In case the data container is empty, the returned iterator points to an invalid object, which, if accessed in that state, would result in an ::std::out_fo_range exception being thrown.

cend ()

`const_iterator cend () const`

Description:

This method returns a read-only iterator, pointing at the data entry past the final entry.

Remarks:

This method instantiates a read-only iterator pointing at the next data entry past the final entry. The returned iterator can not be evaluated, since the returned iterator is initialised to the return value of size (), which always points at an invalid object.

Return value:

The return value is a read-only iterator pointing at the next data entry past the last entry.

crbegin ()

`const_reverse_iterator crbegin () const`

Description:

This method returns a read-only reverse iterator, pointing at the final data entry.

Remarks:

This method instantiates a reverse read-only iterator pointing at the final data entry. The returned iterator has not been evaluated, which means calling this method does not result in a penalty in terms of CPU time.

Return value:

The return value is a reverse read-only iterator pointing at the final data entry. In case the data container is empty, the returned iterator points at an invalid object, which the object prior the non-existing initial entry. If the iterator is accessed in that state, an ::std::out_of_range exception is thrown.

crend ()

```
const_reverse_iterator crend () const
```

Description:

This method returns a read-only reverse iterator, pointing at the data entry before the initial entry.

Remarks:

This method instantiates a reverse read-only iterator pointing at the data entry prior the first entry. The returned iterator can not be evaluated, since the returned iterator has been set to point at -1 (~0x0), which always points at an invalid object.

Return value:

The return value is a reverse read-only iterator pointing at the data entry prior the first data entry.

unload ()

```
void unload ()
```

Description:

This method unloads all data to the medium.

Remarks:

This method stores all data in the instance's associated temporary file and frees all temporary buffers and data caches. It may be useful if the resources on a system are scarce and it is known access is not needed for a long time.

It is not required to reload the data, since this is done by any call trying to access the instance.

However, even a call to a simple method, like size () or empty (), were a reload wouldn't be expected, will cause data to be reloaded. Hence, if it is not the intention to reload a container's data layer, then don't call any that container's methods.

Also calling the unload method too frequently is not recommended, as it will result in significant performance issues.

Note:

If this method is called while the data layer has not been initialised or was de-initialised before, then this method is ineffective. This is also the case, if the container has been unloaded before and has not been reloaded in the mean time.

CBTreeBaseDefaults

The class C. implements what is described in section Specific Default Container to b-tree Methods (CBTreeBaseDefaults).

Type Definitions

This container class is derived from CBTreeBaseIf (Abstract Container to b-tree Interface (CBTreeBaseIf)) as it can be seen in the Class Hierarchy diagram and defines the following types:

Type	Definition	Description
value_type	_t_data	This type defines what a data entry is and what is displayed to the application by the container when accessing it.
data_layer_properties_type	_t_datalayerproperties	This type aids the definition of the types below.
node_iter_type	typename data_layer_properties_type::node_iter_type	This defines what type is used to address individual nodes within the tree structure.
sub_node_iter_type	typename data_layer_properties_type::sub_node_iter_type	This defines what type is used to address individual data entries

Type	Definition	Description
		within a node.

Constructor

```
1)
CBTreeBaseDefaults<_ti_pos, _t_data, _t_datalayerproperties>
(
        const _t_datalayerproperties &rDataLayerProperties,
        const sub_node_iter_type nNodeSize
)
2)
CBTreeBaseDefaults<_ti_pos, _t_data, _t_datalayerproperties>
(
        const CBTreeBaseDefaults<_ti_pos, _t_data, _t_datalayerproperties> &rContainer
)
3)
CBTreeBaseDefaults<_ti_pos, _t_data, _t_datalayerproperties>
(
        CBTreeBaseDefaults<_ti_pos, _t_data, _t_datalayerproperties> &&rRhsContainer
)
```

Description:

The CBTreeBaseDefaults constructor is called when a b-tree data class of any type is called into existence.

Input parameters:

rDataLayerProperties	- references data layer properties *
nNodeSize	- specifies the node size parameter
rContainer	- specifies reference to instance to be copied
rRhsContainer	- specifies right hand side reference to container to be moved

* see section: Data Layer Properties (_t_datalayerproperties) and Data Layer Type Definition (data_layer_type)

** structure psCacheDescription points at:

Remarks:

The constructor initialises the most basic parts of the container to be called into existence, by taking a copy of what rDataLayerProperties is referring to, which is used as soon as the data layer needs to be called into existence. Also, copies are taken from the individual parameters contained by what the pointer of psCacheDescription is referring to.

psCacheDescription->nMinNumberOfBytesPerSuperBlock: This parameter is used during the data layer instantiation and sets parameter nBlockSize of the data layer constructor. Usually this parameter is set to zero and therefore the data layer set its ideal or default block size on its own. However, if an application needs to manually set the data layer block size, for instance to debug the data layer or to run a performance test, then this parameter is to be modified to the desired value. For more information see section: Abstract Data Layer Construction

The parameter nNodeSize (in more generic b-tree documentations often referred to as parameter “t”) determines how many data items and sub-node entries are stored in every node. The resulting number of sub-node entries is nNodeSize times 2 (t * 2), while the number data entries is nNodeSize times 2 minus 1 (t * 2 – 1).

test_integrity ()

```
bool test_integrity () const
```

Description:

This method performs a self test of the b-tree's integrity.

Remarks:

This method tells the b-tree to test every node's integrity. This means that, the b-tree is being walked through and every node is tested for the correct parent node identifier, if the sub-tree size is correct and additional tests may be done depending on what type of data class is this being called on (ie. CBTreeKeySort also tests the key order in every node).

Also it is being tested if two or more nodes are not sharing the same node identifier, if walking from a parent node to a child node and vice versa is working correctly and if a node falsely claims to be the root node.

Return value:

The return value is true in case the self test didn't bring up any errors, otherwise false.

Note:

If the method was incapable to walk through the entire tree, because it has encountered an undefined state in which it is impossible to proceed, then an std::exception is thrown. Therefore the call of this method must always happen within a try block.

This method ought not to be called during normal operations and only aids development purposes. This method is likely to cause performance issues when called too frequently, not only the walk through the tree structure can take very long, depending on the size of the tree, but also all caches are being brought into a state which is unlikely to be ideal for normal applications.

show_integrity ()

```
void show_integrity (const char *pFilename) const
```

Description:

This method prints the current b-tree structure as well as additional debug information into a HTML formatted file.

Input parameters:

pFilename	- specifies pointer to output file name

Remarks:

For development purposes the structure of a b-tree plus additional debug information can be printed into HTML file, which then can be viewed in any internet browser. The method recursively walks through the tree and prints any node descriptor it comes across in plain text as well as every node's content. Errors that had been identified by test_integrity () are high lighted as bold red text within the output tree, given that said method managed to complete. In case test_integrity () didn't complete due to an assertion being hit the output of the b-tree structure is likely to be incomplete.

The next part being output is a linear list of all node descriptors contained by all blocks that have been called into existence so far, while any node being invalid is displayed as grey text. The element nNumData is printed as its 2's complement decimal value.

The final part shows some data layer variables, the pool set up as it has been configured during the construction of the data class instance, followed a linear list of every node's pools, displaying the raw data therein. The data layer variables displayed are as follows:

m_nBlockSize	block size in bytes
m_nNodeSize	node size parameter as it has been given during the construction
m_nNodesPerBlock	number of nodes stored in one block

m_nNodesPerBlockVectorSize	log2 of m_nNodesPerBlock
m_nAlignedNodeSize	size of one node plus padding
m_nNumPerBlockDataPools	number of pools

The next sub part of the data dump section shows a table containing information about every registered pool. The information displayed per pool is as follows:

m_psPerBlockDataPools->nCacheVectorSize	number of cache mask bit as configured during the construction
m_psPerBlockDataPools->nEntrySize	size of one data item as configured during the construction
m_psPerBlockDataPools->nTotalSize	size of all data items as configured during the construction
m_pnPerBlockPoolCacheMaskes	cache mask of a pools determined by m_psPerBlockDataPools->nCacheVectorSize
m_pnPerBlockPoolOffset	offset of a pool within a node

The final sub part of the data dump section is a linear list of every block that has been called into existence on the data layer. Each block's section displays the nodes stored in it, while each node shows its pools raw contents preceded by the absolute data layer byte offset of the respective pool. All information coming from the data layer are displayed as grey text, except if the data layer employs caches. Pools being cached have their contents displayed twice. Once as it is directly stored on the data layer and once as it is coming from the pool cache, being printed as grey and blue text respectively.

Note:

For this method to correctly output the content of each data item within the b-tree structure, as it is required for the initial section of HTML file being written to, every data class must provide its version of the method show_data (), since it is impossible for the classes Specific Default Container to b-tree Methods (CBTreeBaseDefaults) as well as Specific Data Classes to interpret the raw data coming from the data layer. If show_data () is not provided, then in case s. is called the data items in the HTLM output are left blank, rendering the initial part of the debug output almost useless.

show_data ()

bool show_data (std::ofstream &ofs, std::stringstream &rstrData, std::stringstream &rszMsg, const node_iter_type nNode, const sub_node_iter_type nSubPos) const

Description:

This method displays the data of a node in HTML format and prints it to ofs or tests a node for its integrity.

Input parameters:

ofs	- reference to output file stream
rstrData	- reference to data output stream
rszMsg	- reference to temporary error message stream in case of an exception
nNode	- specifies node of node / sub-position pair to be displayed
nSubPos	- specifies sub-position of node / sub-position pair to be displayed

Remarks:

This method has two modes which are indirectly set by what ofs.is_open () returns. If ofs.is_open () returns true, then the method interprets the data item being specified by nNode and nSubPos to be printed in HTML format and writes the result to the output file stream (ofs) as well as testing for the data's integrity. In the event an error is detected, then rszMsg is used to store a human readable error message. If ofs.is_open () returns false, then only the data pointed by the nNode / nSubPos pair is tested for its correct integrity.

Return value:

Returns true if the data integrity is correct, otherwise false. Whether the stream (ofs) is open or not, has no impact on the return value.

CBTreeArray API

Type Definitions

This container class is derived from CBTreeBaseDefaults (Specific Default Container to b-tree Methods (CBTreeBaseDefaults)) as it can be seen in the Class Hierarchy diagram and defines the following types:

Type	Definition	Description
value_type	_t_data	This type defines what a data entry is and what is displayed to the application by the container when accessing it.
size_type	typename data_layer_properties_type::size_type	This defines what type is employed to address individual data entries when accessing a container as a linear array container type.
reference	value_type &	This type defines what a read / write reference of a data entry is.
const_reference	const value_type &	This type defines what a read-only reference of a data entry is.
pointer	value_type *	This type defines what a read / write pointer of a data entry is.
const_pointer	const value_type *	This type defines what a read-only pointer of a data entry is.
difference_type	typename ::std::make_signed<size_type>::type	This defines what types is used to display the distance between two data entries is.
CBTreeIf_t	CBTreeIf<value_type, size_type>	This type aids the definition of the types below.
iterator	typename CBTreeIf_t::iterator	This defines what is used as a read / write iterator type.

Type	Definition	Description
const_iterator	typename CBTreeIf_t::const_iterator	This defines what is used as a read -only iterator type.
reverse_iterator	typename CBTreeIf_t::reverse_iterator	This defines what is used as a read / write reverse iterator type.
const_reverse_iterator	typename CBTreeIf_t::const_reverse_iterator	This defines what is used as a read -only reverse iterator type.
CBTreeArrayIf_t	CBTreeArrayIf<value_type, size_type>	This defines what the most abstract type is within the framework.

CBTreeArray Constructor

```
1)
CBTreeArray<_t_data, _t_datalayerproperties>
(
	const data_layer_properties_type &rDataLayerProperties,
	const sub_node_iter_type nNodeSize
)
2)
CBTreeArray<_t_data, _t_datalayerproperties>
(
	const CBTreeArray<_t_data, _t_datalayerproperties> &rContainer,
	const bool bAssign = true
)
3)
CBTreeArray<_t_data, _t_datalayerproperties>
(
	CBTreeArray<_t_data, _t_datalayerproperties> &&rRhsContainer
)
```

Description:

The CBTreeArray constructor is called when a b-tree array data class is called into existence.

Input parameters:

rDataLayerProperties	- is a reference which specifies the properties for a data layer class *
nNodeSize	- specifies the node size parameter *
rContainer	- specifies the reference of a container instance to be copied
bAssign	- specifies if the source container's content is to be copied as well
rRhsContainer	- specifies the reference of a container instance to be moved

* This parameter has already been explained in section CBTreeIf Constructor and only a brief description is given here. If more information is needed, then please refer to said section.

assign ()

```
1)
void assign<_t_iterator> (_t_iterator sItFirst, _t_iterator sItLast)
2)
void assign (const size_type nNewSize, const value_type& rVal)
```

Description:

These methods assign new content to a container.

Input parameter:

sItFirst	- specifies iterator associated with initial element to be assigned
sItLast	- specifies iterator associated with subsequent element to be assigned
nNewSize	- specifies new size the container will end up with
rVal	- specifies fill element

Remarks:

These methods will destroy any previously stored content of the container in use and replace it with the content according to their specified parameters. This means that the data layer is destroyed and then constructed again, except if sItFirst equals sItLast or nNewSize is zero. In those cases the data layer remains destroyed, until another call needs to stored data in this container.

Method (1) iterates element by element from sItFirst onwards until sItLast has been found. In case sItLast is not being found and the container associated with sItFirst goes beyond its last element, then an ::std::out_of_range exception is thrown.

Note:

For method (1), in case the input iterators are referring to the target container itself, then all elements before sItFirst and all elements after sItLast are being deleted, which means the method is capable of self-referencing. However, if sItLast is before sItFirst or one of those input iterators is referring to a container other than this container, then the behaviour is undefined.

For method (1), type _t_iterator must be an iterator that is at least category ::std::forward_iterator_tag to function. To achieve best results in terms of performance, type _t_iterator must be an iterator that is category ::std::random_access_iterator_tag to allow the a. method to move back and forth quickly, creating an idealised order of elements in the target container.

push_back ()

```
void push_back (const value_type &rData)
```

Description:

This method appends a new data item to the end of the container.

Input parameter:

rData	- specifies reference to data item to be appended

Remarks:

This method adds the data item referred to by rData to the location this->cend () is pointing at, effectively increasing the target container size by one. This method is incapable to return an error and in case the item cannot be appended, then an ::std::runtimeerror exception is thrown.

pop_back ()

```
void pop_back ()
```

Description:

This method removes the last element of the target container.

Remarks:

This method removes the last element of the target container, effectively decreasing its size by one. In cast the target container is empty, then an ::std::out_of_range exception is thrown.

insert ()

```
1)
template <class _t_iterator>
iterator insert (const_iterator sCIterPos, _t_iterator sItFirst, _t_iterator sItLast)
2)
iterator insert (const_iterator sCIterPos, const value_type& rData)
3)
iterator insert (const_iterator sCIterPos, const size_type nLen, const value_type& rData)
```

Description:

These methods insert new data elements into a target container.

Input parameter:

sCIterPos	- specifies position where data items are to be inserted
sItFirst	- specifies iterator associated with initial element to be assigned
sItLast	- specifies iterator associated with subsequent element to be assigned
rData	- specifies data element (2) or fill element (3) to be inserted
nLen	- specifies how many fill elements to be inserted

Remarks:

According to their parameters, these methods will insert one or more data items to the target container. In case either sItFirst equals sItLast or nLen is zero, the call is ineffective neither the stored data nor the state of the target container is being altered, only an iterator version of parameter sCIterPos is returned.

Method (1) iterates element by element from sItFirst onwards until sItLast has been found. Any found element is inserted into the container in the same order they have been iterated through from their source. That series of elements is inserted at the position the parameter sCIterPos is pointing at. In case sItLast is not being found and the container associated with sItFirst goes beyond its last element, then an ::std::out_of_range exception is thrown.

Method (2) inserts the data item rData at the position sCIterPos is pointing at, into the container. If sCIterPos is exceeding the container, then an ::std::out_of_range exception is thrown.

Method (3) inserts nLen data items rData at the position sCIterPos is pointing at, into the container. If sCIterPos is exceeding the container, then an ::std::out_of_range exception is thrown.

Return value:

All methods return an iterator pointing at the initial location where data was being inserted.

Note:

For method (1), in case the input iterators are referring to the target container itself, then the method acts as if it would take a copy of all elements between sItFirst and sItLast, to then paste those back in where sCIterPos is pointing as. This means that, the method is capable of self-referencing, without destroying the order of elements. However, if sItLast is before sItFirst or one of those input iterators is referring to a container other than the target container, then the behaviour is undefined.

emplace ()

```
1)
template<class ..._t_va_args>
iterator emplace (const_iterator sCIterPos, _t_va_args && ... rrArgs)
```

```
2)
iterator emplace (const_iterator sCIterPos, value_type &&rData)
```

Description:

This method inserts a new data element into a target container by constructing it in place.

Input parameter:

sCIterPos	- specifies position where data items are to be inserted
rrArgs	- specifies constructor parameters to be forwarded to construct an entry in place
rData	- specifies right hand reference to be forwarded to construct an entry in place

Remarks:

These methods construct a data entry with the constructor parameters specified by rData or rrArgs at the position specified by sCIterPos. As a result, the container size is increased by one, since all data entries from sCIterPos and onwards will end up one position further down, while the new entry is constructed in place.

Return value:

These methods return an iterator pointing at the location where data was being inserted.

emplace_back ()

```
1)
template<class ..._t_va_args>
iterator emplace_back (_t_va_args && ... rrArgs)
2)
iterator emplace_back (value_type &&rData)
```

Description:

These methods append a new data element to a target container by constructing it in place.

Input parameter:

rrArgs	- specifies constructor parameters to be forwarded to construct an entry in place
rData	- specifies right hand reference to be forwarded to construct an entry in place

Remarks:

These methods construct a data entry with the constructor parameters specified by rData or rrArgs at the last position of the container. As a result, the container size is increased by one, since the new entry is constructed in place at the end of the container.

Return value:

These methods return an iterator pointing at the location where data was being inserted.

erase ()

```
1)
iterator erase (const_iterator sCIterPos)
2)
iterator erase (const_iterator sCIterFirst, const_iterator sCIterLast)
```

Description:

These methods remove one or more elements from the target container.

Input parameter:

sCIterPos	- specifies location of data item to be removed
sCIterFirst	- specifies location of initial item to be removed
sCIterLast	- specifies location of final item to be removed

Remarks:

To remove one or more entries, these methods are to be used. If any method is called with an empty target container or method (2) is called with sCIterLast pointing at an illegal location such as before sCIterFirst or beyond the end of the target container, then an ::std::out_of_range exception is thrown. In general, the only behavioural exception is; if sCIterFirst equals sCIterLast. In that case, the call of method (2) is ineffective and the return value is an iterator version of sCIterFirst.

Return value:

The return value is an iterator pointing to the location where data was being removed from.

swap ()

```
1)
void swap (CBTreeArray &rArray)
2)
void swap (CBTreeArrayIf_t &rArrayIf)
```

Description:

This method swaps the content of a source container with the content of the target container.

Input parameter:

rArray	- specifies reference to target container the content is to be swapped with
rArrayIf	- specifies reference to an abstract target container the content is to be swapped with

Remarks:

These method swap the content of two containers by their entire status as well as the pointers to the respective data layer instances. The only exceptions are, the subscript wrapper object, which is used as return value template by the assignment operator (operator=), the iterator registration list and the time stamp state. However, the time stamp state of the source and target container get updated, so that any iterator calling either container is forced to re-evaluate.

serialize ()

```
size_type serialize (const size_type nStart, const size_type nLen, value_type *pData) const
```

Description:

This method reads a sequence of data entries and writes those back into an output array.

Input parameters:

nStart	- specifies linear position of initial entry
nLen	- specifies number of data items to be output
pData	- specifies a pointer to an array large enough to hold all output data

Remarks:

This method finds the position associated with linear address nStart. From that point on the method walks along entries in an ascending order and progressively writes the data items to the array pData is pointing at, until at least nLen data items have been output.

If this method hits the final data item before nLen entries have been output, then the method aborts early and the array pData is pointing at may not be fully used. The method is ineffective if nLen is set to zero.

Return value:

The value returned is the number of data items that have successfully retrieved from the container and written to the output array. If everything went successful, then the return value is equal to nLen. In case the return value is lower than that, then the end of the container must have been encountered prematurely during the data entry walk.

operator=

```
1)
CBTreeArray<> &operator= (const CBTreeArray<_t_data, _t_datalayerproperties> &rContainer)
2)
CBTreeArray<> &operator= (CBTreeArray<_t_data, _t_datalayerproperties> &&rRhsContainer)
```

Description:

This is an overloaded array data class assignment and move operator.

Input parameter:

rContainer	- specifies the reference of a source container to take a copy of
rRhsContainer	- specifies the reference of a source container to be moved

Remarks:

These operators destroy all data in the target container and copy or swap all data from the source container rContainer to the target container. Also copied across will be parameters such as:

- data layer properties
- nodes size
- cache description

The assignment operator (1) tries to balance the tree structure as evenly as possible, which means the tree structure of rContainer and the target instance are not guaranteed to be the exact same, in terms of tree structure. However, this has no influence on how any process, accessing the container, sees the data. This means that, rContainer and the target instance will appear the same to the calling application.

The move operator (2) swaps any member of the source and target container, except for the subscript wrapper object, which is used as return value template by the assignment operator (operator=), the iterator registration list and the time stamp state. However, the time stamp state of the source and target container get updated, so that any iterator calling either container is forced to re-evaluate.

Return value:

The return value is a reference to the target container.

operator[]

1)
```
CBTreeArrayAccessWrapper<> operator[] (const size_type nPos)
```
2)
```
const value_type& operator[] (const size_type nPos) const
```

Description:

These are overloaded b-tree array class subscript operators.

Input parameter:

nPos	specifies the linear position to be accessed

Remarks:

The read / write sub-script operator (1) creates an instance of type CBTreeArrayAccessWrapper, which provides an overloaded cast operator to handle read accesses and an overloaded assignment operator to handle write accesses. Since the subscript operator cannot distinguish between read and write accesses by itself, it must divert accesses to the already mentioned wrapper class to aid this operator to that end.

The read-only subscript operator (2) copies the a data entry instance to a separate location, which is being referred to by the return value.

Return value:

The return value of operator (1) is an instance of CBTreeArrayAccessWrapper, which uses the same template parameter settings as this.

The return value of operator (2) is a const reference to a copy of the data entry that has been requested.

CBTreeArrayAccessWrapper

This class exists solely to aid the compiler to generate the correct code for lvalue and rvalue (reads and writes) accesses when a CBTreeArray data class is accessed via a subscript operator. Creating a generic subscript operator for class CBTreeArray is not possible, since, dependent on the data layer in use, the returned reference would have potentially pointed to a cache line, which results in a number of problems. Firstly, the cache can be overwritten by any other call to the same array class instance. This means that, the reference is volatile and the result may become invalid at any point. Secondly, the reference doesn't offer any trigger in case its content has been modified and therewith any write would be eventually discarded. Thirdly, if the data layer has been destroyed for any reason, such as clear () has been called or the b-tree class instance itself no longer exists, then the next access to the returned reference is guaranteed to fail with an access violation.

CBTreeArrayAccessWrapper Constructor

```
CBTreeArrayAccessWrapper<_t_data, _t_sizetype>
(
        CBTreeArrayIf<_t_data, _t_sizetype> &rInstance
)
```

Description:

This constructor initialises a sub-script wrapper instance aiding the container type CBTreeArray when it is not clear if a read or write access has to be performed.

Input parameters:

rInstance	- specifies a reference of a data class instance to be accessed

Remarks:

This constructor is executed when a sub-script wrapper type is called into existence telling the compiler how to handle an lvalue or rvalue access to an array data class instance. The constructor takes a copy of the parameter, which is then used by the overloaded operators being part of this class, once it is has determined if a read or write access is required.

set ()

```
void set (const _t_sizetype nPos)
```

Description:

This method sets the absolute linear offset of the data entry to be accessed.

Input Parameter:

nPos	- specifies the absolute linear offset to be accessed

Remarks:

This method sets the absolute linear offset of the data entry to be accessed. The input parameter is not sanity checked and therefore may be out of the container's range. If that is the case, then this would only be a problem once the wrapper instance is employed to access the container, which would result in an ::std::runtimeerror exception being thrown.

*operator**

```
_t_data & operator* ()
```

Description:

This overloaded indirection operator returns a read-only reference the wrapper is pointing at.

Remarks:

The operator returns a reference to a data entry the wrapper is referring to. The data entry of interest can be selected by using the method set (), otherwise the data entry this operator returns is the initial entry of the container this wrapper instance is associated with.

On access the operator copies the data entry to a member of the wrapper and returns a reference to that. This means that, in terms of container access, the result is read-only, since any write to the return value is discarded once the wrapper performs the next access. In case, the selected data entry is not in range of the associated container, then an ::std::runtimeerror exception is thrown.

Return value:

The return value is a reference to a copy of a data entry that has been selected prior to access.

operator->

```
_t_data & operator-> ()
```

Description:

This overloaded indirection operator returns a read-only reference the wrapper is pointing at.

Remarks:

The operator returns a reference to a data entry the wrapper is referring to. The data entry of interest can be selected by using the method set (), otherwise the data entry this operator returns is the initial entry of the container this wrapper instance is associated with.

On access the operator copies the data entry to a member of the wrapper and returns a reference to that. This means that, in terms of container access, the result is read-only, since any write to the return value is discarded once the wrapper performs the next access. In case, the selected data entry is not in range of the associated container, then an ::std::runtimeerror exception is thrown.

Return value:

The return value is a reference to a copy of a data entry that has been selected prior to access.

Note:

In the current implementation this operator redirects its call to operator*, resulting in equivalent behaviour.

operator const value_type &

`operator const value_type& ()`

Description:

This is an overloaded array subscript access wrapper class cast operator and acts as the lvalue part of the CBTreeArray's overloaded subscript operator.

Remarks:

This operator uses linear position being set during the construction to return the entry associated with said linear position. In case an error occurs, this operator throws an std::runtimeerror exception. This means that, code sections employing this operator ought to be embedded by a try-catch-section.

operator=

```
1)
CBTreeArrayAccessWrapper<_t_data, _t_sizetype> &operator= (const _t_data &rData)
2)
CBTreeArrayAccessWrapper<_t_data, _t_sizetype> &operator=
(
        const CBTreeArrayAccessWrapper<_t_data, _t_sizetype> &rData
)
```

Description:

These overloaded assignment operators take an rvalue and transfers it to the data entry being referred to.

Input parameter:

rData	specifies the reference of a data item to be written

Remarks:

These operators use the linear position being selected by the method set () as well as the input parameter rData to modify the existing entry associated with said linear position. In case an error occurs, this operator throws an std::runtimeerror exception. This means that, code sections employing this operator ought to be embedded by a try-catch-section.

Return value:

As per definition of the assignment operator, a reference to this is returned.

CBTreeKeySort API

Type Definitions

This container class is derived from CBTreeAssociative, which one of the Abstract Data Classes as it can be seen in the Class Hierarchy diagram and defines the following types:

Type	Definition	Description
value_type	_t_data	This type defines what a data entry is and what is displayed to the application by the container when accessing it.
key_type	_t_key	This defines what type is used as an operand for the sorting criteria when determining the order of data entries stored by the container.
size_type	typename data_layer_properties_type::size_type	This defines what type is employed to address individual data entries when accessing a container as a linear array container type.
reference	value_type &	This type defines what a read / write reference of a data entry is.
const_reference	const value_type &	This type defines what a read-only reference of a data entry is.
pointer	value_type *	This type defines what a read / write pointer of a data entry is.
const_pointer	const value_type *	This type defines what a read-only pointer of a data entry is.
difference_type	typename ::std::make_signed<size_type>::type	This defines what types is used to display the distance between two data entries is.
CBTreeAssociative_t	CBTreeAssociative<value_type, key_type, data_layer_properties_type>	This type aids the definition of the types below.
iterator	typename CBTreeAssociative_t::iterator	This defines what is used as a read / write iterator type.
const_iterator	typename CBTreeAssociative_t::const_iterator	This defines what is used as a read -only iterator type.
reverse_iterator	typename CBTreeAssociative_t::reverse_iterator	This defines what is used as a read / write reverse iterator type.
const_reverse_iterator	typename CBTreeAssociative_t::const_reverse_i	This defines what is used as a read

Type	Definition	Description
	terator	-only reverse iterator type.

CBTreeKeySort Constructor

```
1)
CBTreeKeySort<_t_data, _t_key, _t_datalayerproperties>
(
	const data_layer_properties_type &rDataLayerProperties,
	const sub_node_iter_type nNodeSize
)
2)
CBTreeKeySort<_t_data, _t_key, _t_datalayerproperties>
(
	const CBTreeKeySort<_t_data, _t_key, _t_datalayerproperties> &rContainer,
	const bool bAssign = true
)
3)
CBTreeKeySort<_t_data, _t_key, _t_datalayerproperties>
(
	CBTreeKeySort<_t_data, _t_key, _t_datalayerproperties> &&rRhsContainer
)
```

Description:

The CBTreeKeySort constructor is called when a key sort data class is called into existence.

Input parameters:

rDataLayerProperties	- is a reference which specifies the properties for a data layer class *
nNodeSize	- specifies the node size parameter *
rContainer	- specifies the reference of a container instance to be copied
bAssign	- specifies if the source container's content is to be copied as well
rRhsContainer	- specifies the reference of a container instance to be moved

* This parameter has already been explained in section CBTreeIf Constructor and only a brief description is given here. If more information is needed, then please refer to said section.

insert ()

```
1)
template <class _t_iterator>
void insert (_t_iterator sItFirst, _t_iterator sItLast)
2)
iterator insert (const value_type &rData)
3)
iterator insert (const_iterator sCIterHint, const value_type &rData)
```

Description:

These methods insert new data elements into a target container.

Input parameters:

sItFirst	- specifies iterator associated with initial element to be inserted
sItLast	- specifies iterator associated with subsequent of the last element to be inserted
rData	- specifies reference to data element to be inserted
sCIterHint	- indicates position near the place where data entry is to be inserted

Remarks:

These methods will insert one or more data items to the target container, according to their parameters. In case sItFirst equals sItLast, the call is ineffective neither the stored data nor the state of the target container is being altered.

Method (1) iterates element by element from sItFirst onwards until sItLast has been found. In case sItLast is not being found and the container associated with sItFirst goes beyond its last element, then an ::std::out_of_range exception is thrown.

Method (2) inserts one new data entry, while it depends on the key value the data entry generates where the new data entry is being inserted.

Method (3) inserts one new data entry, while trying to use sCIterHint as an indicator where that insertion has to take place, for an accelerated operation. If the position displayed by sCIterHint is insufficient to be used, then this method ignores sCIterHint and falls back to method (2).

Return value:

The return value of methods (2) and (3) is the position associating the location where the data set has been inserted.

Note:

For method (1), in case the input iterators are referring to the target container itself, then the method acts as if it would take a copy of all elements between sItFirst and sItLast, to then insert those as if these were data elements from a different source. This means that, the method is capable of self-referencing, without destroying elements or ending up in an invalid state. However, if sItLast is before sItFirst or one of those input iterators is referring to a container other than the target container, then the behaviour is undefined.

emplace ()

```
1)
iterator emplace (value_type &&rrData)
2)
template<class ..._t_va_args>
iterator emplace (_t_va_args && ... rrArgs)
```

Description:

These methods insert a new data entry by constructing it in place.

Input parameters:

rrData	- specifies input for the move constructor to move an entry in place
rrArgs	- specifies constructor parameters to be forwarded to construct an entry in place

Remarks:

Both methods construct a new entry in place based on the constructor arguments, which increases the container size by one.

Return value:

Both methods return an iterator specifying the location where the new data entry has been constructed.

emplace_hint ()

1)
```
iterator emplace_hint (const_iterator sCIterHint, value_type &&rData)
```
2)
```
template<class ..._t_va_args>
iterator emplace_hint (const_iterator sCIterHint, _t_va_args && ... rrArgs)
```

Description:

These methods insert a new data entry by constructing it in place, while trying to exploit a hint indicator for accelerated access.

Input parameters:

rrData	- specifies input for the move constructor to move an entry in place
rrArgs	- specifies constructor parameters to be forwarded to construct an entry in place
sCIterHint	- indicates position near the place where data entry is to be constructed

Remarks:

Both methods construct a new entry in place based on the constructor arguments, which increases the container size by one, while trying to use sCIterHint as an indicator where that insertion has to take place, for an accelerated operation. If the position displayed by sCIterHint is insufficient to be used, then both methods ignore sCIterHint and fall back to their emplace () equivalents.

Return value:

Both methods return an iterator specifying the location where the new data entry has been constructed.

erase ()

1)
```
iterator erase (const_iterator sCIterPos)
```
2)
```
size_type erase (const key_type &rKey)
```
3)
```
iterator erase (const_iterator sCIterFirst, const_iterator sCIterLast)
```

Description:

These methods remove one or more elements.

Input parameters:

sCIterPos	- specifies position of data entry to be removed
rKey	- specifies key of data entries to be removed
sCIterFirst	- specifies initial position of data entries to be removed
sCIterLast	- specifies data entry following the final data entry to be removed

Remarks:

To remove one or more entries, these methods can be used. If any method is called with an empty target container or method (3) is called with sCIterLast pointing at an illegal location such as before sCIterFirst or beyond the end of the target container, then an ::std::out_of_range exception is thrown. One behavioural exception is; if sCIterFirst equals sCIterLast. In that case, the call of method (3) is ineffective and the return value is sCIterFirst again. Also, if, for method (2), the specifies key is not present in the container, then the call is ineffective and the return value is zero.

Return value:

The return value of method (1) and (3) is an iterator pointing to the location where data was being removed from.

Method (2) returns the number of data entries that were removed.

swap ()

```
1)
void swap (CBTreeAssociative &rContainer)
2)
void swap (CBTreeKeySort &rKeySort)
```

Description:

This method swaps the content of a source and a target key sort container.

Input parameters:

rContainer	- specifies reference to abstract source container, content is to be swapped with

rKeySort - specifies reference to source container, content is to be swapped with

Remarks:

This method swaps the content of two key sort containers by their entire status as well as the pointers to the respective data layer instances. The only exceptions are the iterator registration list and the time stamp state. However, the time stamp state of the source and target container get updated, so that any iterator calling the either container is forced to re-evaluate its location.

find ()

1)
```
iterator find (const key_type &rKey)
```
2)
```
const_iterator find (const key_type &rKey) const
```

Description:

These methods find the location of a key.

Input parameters:

rKey - specifies reference of key which associated entries are to be located

Remarks:

These methods find the location of one data entry which has the associated key specified by the input parameter rKey. In case more than one entry sharing the same key is present, then it isn't guaranteed to be the initial position of that data entry sub-set in question. To find the initial position of a key-set lower_bound () has to be called. This method is useful to find out if data entries with a specific key exists, since it has the chance of an early abort when tracing into the container.

Return value:

If any data entry that has an associated key has been found, then the returned iterator points at said entry, otherwise it points at a value equivalent to the return value of cend ().

lower_bound ()

1)
```
iterator lower_bound (key_type &rKey)
```
2)
```
const_iterator lower_bound (const key_type &rKey)
```

Description:

These methods return the initial location of an entry set sharing the same key.

Input parameters:

rKey	- specifies key associated with data entry set

Remarks:

These methods are employed if the initial location of a data entry set sharing the same key, specified by the input parameter rKey, has to be found.

Return value:

The return value is an iterator, which if the call was successful, is pointing at the initial location of the data entries, that have an associated key value equal to input parameter rKey, otherwise it points at the location of a data entry being associated with a key deemed to have the next greater key value.

upper_bound ()

1)
```
iterator upper_bound (key_type &rKey)
```
2)
```
const_iterator upper_bound (const key_type &rKey)
```

Description:

These methods return the location of an entry followed by set of data entries sharing the same key.

Input parameters:

rKey	- specifies key associated with data entry set

Remarks:

These methods are employed if the location of a data entry following a set of data entries sharing the same key, specified by the input parameter rKey, has to be found.

Return value:

The return value is an iterator, which if the call was successful, is pointing at the location of a data entry followed by the data entries, that have an associated key value equal to input parameter rKey, otherwise it points at the location equivalent to cend ().

equal_range ()

1)
```
::std::pair<iterator, iterator> equal_range (const key_type &rKey)
```
2)
```
::std::pair<const_iterator, const_iterator> equal_range (const key_type &rKey) const
```

Description:

These methods return the range of a specific key value.

Input parameters:

rKey	- specifies key associated with data entry set

Remarks:

These methods can be employed to obtain the range of data entries holding a specific key value. If a data entry containing the specified key value doesn’t exist, then both iterators returned refer to the data entry holding what is deemed to be the next higher key value.

Return value:

The returned pair value contains in its first element what lower_bound () would return and in the second element what upper_bound () would return with the input key value.

get_prev_key ()

```
void get_prev_key (const key_type &rKey, key_type &rPrevKey, bool &bBounce) const
```

Description:

This method finds the next smaller key value of a specified key value.

Input parameter:

rKey	- specifies reference to key value

Output parameters:

nPrevKey	- specifies reference to previous key value
bBounce	- specifies reference to bounce value

Remarks:

This method looks for a key value deemed to be next smaller key value of specified by input parameter rKey. If the method was successful and a smaller key has been found, then bBounce is returned as false and rPrevKey contains the next smaller key, otherwise bBounce is true and rPrevKey is undefined.

Note:

This method also works even if a data set associated with input parameter rKey is not present.

get_next_key ()

```
void get_next_key (const key_type &rKey, key_type &rNextKey, bool &bBounce) const
```

Description:

This method finds the next greater key value of a specified key value.

Input parameter:

rKey	- specifies reference to key value

Output parameters:

rNextKey	- specifies reference to next key value
bBounce	- specifies reference to bounce value

Remarks:

This method looks for a key value deemed to be next greater key value of specified by input parameter rKey. If the method was successful and a greater key has been found, then bBounce is returned as false and rNextKey contains the next greater key, otherwise bBounce is true and rNextKey is undefined.

Note:

This method also works even if a data set associated with input parameter rKey is not present.

count ()

```
size_type count (const key_type &rKey) const
```

Description:

This method determines the number of entries sharing the same key.

Input parameters:

rKey	- specifies the key in question

Remarks:

To determine how many entries are sharing the same key, this method is used. It does so by tracing into the container, while looking for any key that is equal to the value given by parameter key. Once any data entry containing the key in question has been found, the method walks backward until a data entry with a different key value has been found or the beginning of the container has been reached. The last step walks forward until a data entry with a different key value has been found or the end of the tree has been reached, while counting the number of entries sharing the same key. This means that, if the container instance has one or more large data sets sharing the same key, then employing this method may result in performance issues.

Return value:

If the method was successful, then a linear position is returned, otherwise the return value has all bits asserted ((size_type) ~0). If no data items using the key value rKey are present, then the return value is zero.

serialize ()

```
size_type serialize (const size_type nStart, const size_type nLen, value_type *pData) const
```

Description:

This method reads a sequence of data entries and writes those back into an output array.

Input parameters:

nStart	- specifies linear position of initial entry
nLen	- specifies number of data items to be output
pData	- specifies a pointer to an array large enough to hold all output data

Remarks:

This method finds the position associated with linear address nStart. From that point on the method walks along entries in an ascending order and progressively writes the data items to the array pData is pointing at, until at least nLen data items have been output.

If this method hits the final data item before nLen entries have been output, then the method aborts early and the array pData is pointing at may not be fully used. The method is ineffective if nLen is set to zero.

Return value:

The value returned is the number of data items that have successfully retrieved from the container and written to the output array. If everything went successful, then the return value is equal to nLen. In case the return value is lower than that, then the end of the container must have been encountered prematurely during the data entry walk.

operator=

1)
```
CBTreeAssociative &operator= (const CBTreeAssociative &rContainer)
```
2)
```
CBTreeAssociative &operator= (CBTreeAssociative &&rRhsContainer)
```

Description:

This is an overloaded key sort class assignment operator.

Input parameter:

rContainer	- specifies the reference of a source container to take a copy of
rRhsContainer	- specifies the reference of a container instance to be moved

Remarks:

This operator destroys all data in the target container and copies all data from the source container rContainer to the target container. Also copied across will be parameters such as:

- data layer properties
- nodes size
- cache description

The assignment operator (1) tries to balance the tree structure as evenly as possible, which means the tree structure of rContainer and the target instance are not guaranteed to be the same. However, this has no influence on how the calling process sees the data. This means that, rContainer and the target instance will appear the same to the calling application.

The move operator (2) swaps any member of the source and target container, except for the subscript wrapper object, the iterator registration list and the time stamp state. However, the time stamp state of the source and target container get updated, so that any iterator calling either container is forced to re-evaluate.

Return value:

The return value is a reference to the target container.

STL Equivalent Containers

The b-tree framework offers a number of classes, which are aiming to be eqiuivalent with the existing STL data containers. See below:

b-tree framework class	interchanges with
CBTreeMap	::std::map
CBTreeMultiMap	::std::multimap
CBTreeSet	::std::set
CBTreeMultiSet	::std::multiset
CBTreeArray *	::std::vector

* The CBTreeArray API section can be found here: CBTreeArray API

Although the classes listed above are mostly method interface compatible, they have some significant differences that must be addressed. Firstly, the template parameter set has a very divergent approach in terms of what will the container classes use to make data available to a calling application. This means that, the STL classes allow to set what return types will be used, while the b-tree framework classes are using already set return types and instead allow for an alternate data layer to be set. Secondly, due to the b-tree framework's alpha state, it is lacking a number of methods. The most obvious missing methods are those which ought to have a const-qualifier set, in addition to the methods emplace () as well as emplace_hint (). The explanation why the b-tree framework classes lacking any const-qualified method, is because any b-tree instance has a root node cache, which isn't guaranteed to be up-to-date and thus has the potential to be written to on any access, even when calling size () or empty (). The emplace methods haven't been implemented yet, since the absence of them can be worked around by calling insert ().

CBTreeMap, CBTreeMultiMap, CBTreeSet and CBTreeMultiSet API

As discussed above, the four container classes CBTreeMap, CBTreeMultiMap, CBTreeSet and CBTreeMultiSet are aiming to be equivalent with their respective STL counter parts. Since these four classes have very similar interfaces, it makes sense for them to be derived from the same source, which in their case is the Abstract Data Classes. Hence, a number of API methods, of the four classes in question, have already been displayed in the sections CBTree API as well as CBTreeKeySort API. See the list below:

Method	Class
cbegin ()	Specific Default Container to b-tree Methods (CBTreeBaseDefaults)
cend ()	Specific Default Container to b-tree Methods (CBTreeBaseDefaults)
crbegin ()	Specific Default Container to b-tree Methods (CBTreeBaseDefaults)
crend ()	Specific Default Container to b-tree Methods (CBTreeBaseDefaults)
empty ()	Specific Default Container to b-tree Methods (CBTreeBaseDefaults)
size ()	Specific Default Container to b-tree Methods (CBTreeBaseDefaults)
swap ()	CBTreeKeySort
clear ()	Specific Default Container to b-tree Methods (CBTreeBaseDefaults)
count ()	CBTreeKeySort

These methods won't be discussed here again. To see the description of those methods, click the respective link above. Furthermore, those four classes are derived from the Abstract Data Classes and they are Application Classes, therefore already providing their own methods of comp () and extract_key (). The comp () method provided is linked as an arithmetic comparison in case ::std::is_arithmetic<_t_keytype>::type is resolved as ::std::true_type, otherwise it is resolved as a method that binary compares key values. Unlike the STL containers, the comparison function is not part of the template parameters and in case something other than an equivalent of ::std::less is required, a new version of comp () is needed. This new version has to be introduced by creating a new container class that inherits from the respective STL equivalent application class in question. The extract_key () method on the other hand ought to be sufficient for most applications, since it extracts an already existing key value from a data set in question. This means, as the STL container classes don't have any complex data set to key value conversion and the STL equivalent application classes are reflecting that, it is unlikely that the given extract_key () method needs replacing.

max_size ()

```
size_type max_size () const
```

Description:

This method returns the maximum number of addressable data entries.

Remarks:

To obtain the theoretical maximum number of addressable data entries, this method must be called.

The actual maximum size is likely to be much lower, since it depends on a number of different template parameters, such as Data Layer Type Definition (data_layer_type) and Node Iterator Type (_t_nodeiter) as well as Sub-Node Iterator Type (_t_subnodeiter). While the data layer may run out of resources, the node- and sub-node iterator type template parameters are limiting the maximum size by showing a restricted address space. An additional factor is, that a theoretical top address ((size_type) ~0x0) is reserved for internal use.

Return value:

The return value is the theoretical maximum size, which is generated as follows: size_type (((size_type) ~0x0) - 1).

erase ()

```
1)
iterator erase (const_iterator sCIterPos)
2)
size_type erase (const key_type &rKey)
3)
iterator erase (const_iterator sCIterFirst, const_iterator sCIterLast)
```

Description:

These methods remove one or more data entries from a data container.

Input parameters:

sCIterPos	- specifies iterator referring to data entry to be removed
rKey	- references key value equivalent to any data entry to be removed
sCIterFirst	- specifies iterator referring to initial data entry to be removed
sCIterLast	- specifies iterator referring the data entry following the last item to be removed

Remarks:

To remove one or more entries, these methods are to be used. If any method is called with an empty target container or method (3) is called with sCIterLast pointing at an illegal location such as before sCIterFirst or beyond the end of the target container, then an ::std::out_of_range exception is thrown. One behavioural exception is; if sCIterFirst equals sCIterLast. In that case, the call of method (3) is ineffective and the return value is an iterator version of sCIterFirst. Also, if, for method (2), the specifies key is not present in the container, then the call is ineffective and the return value is zero.

Return value:

The return value of method (1) and (3) is an iterator pointing to the location where data was being removed from.

Method (2) returns the number of data entries that were removed.

key_comp ()

`key_compare key_comp () const`

Description:

This method returns an object capable of comparing two key values.

Remarks:

If it is required to compare two key value, then this method can be employed to create an object able to do so. The returned object has an overloaded call operator, that has two input parameters of type `const _t_keytype` and returns a value of type `bool`. As oppose to the internal comparison method comp () the return type of the overloaded call operator is not an integer and follows a strict weak ordering. This means that, if the first parameter of said call operator is deemed to be less than the second parameter, then the result is `true`, otherwise `false`.

Return value:

The return value is an object containing an overloaded call operator, which is capable to compare key values. The return type of said operator is a boolean, which, if the call operator's first input parameter is less than the second parameter, is returned as `true`, otherwise `false`. Two keys are deemed equivalent, if the input parameters can be given in any order and the result is always `false`.

Note:

Although the overloaded default call operator uses the method comp () to produce the desired result, the chosen return type is `bool` to remain compatible with the existing STL classes.

value_comp ()

value_compare value_comp () const

Description:

This method returns an object capable of comparing the keys of two data items.

Remarks:

If it is required to compare the keys of two data items, then this method can be employed to create an object able to do so. The returned object has an overloaded call operator, that has two input parameters of type `const _t_valuetype` and returns a value of type `bool`. As oppose to the internal comparison method comp () the return type of the overloaded call operator is not an integer and follows a strict weak ordering. This means that, if the first parameter of said call operator is deemed to be less than the second parameter, then the result is `true`, otherwise `false`.

Return value:

The return value is an object containing an overloaded call operator, which is capable to compare the keys of two data items. The return type of said operator is a boolean, which, if the call operator's first input parameter is less than the second parameter, is returned as `true`, otherwise `false`. Two keys of their data entries are deemed equivalent, if the input parameters can be given in any order and the result is always `false`.

Note:

Although the overloaded default call operator uses the method comp () to produce the desired result, the chosen return type is `bool` to remain compatible with the existing STL classes.

find ()

1)
iterator find (const key_type &rKey)

2)
const_iterator find (const key_type &rKey) const

Description:

These methods find the location of an entry associated with a key value.

Input parameters:

rKey	- specifies reference of key which associated entries are to be located

Remarks:

These methods find the location of one data entry which has the associated key specified by the input parameter rKey. In case more than one entry share the same key, then it isn't guaranteed to be the initial position of that key's data entry sub-set. To find the initial position of a key-set lower_bound () has to be used. This method is useful to find out if data entries with a specific key exist, since it has the chance of an early abort when tracing into the b-tree structure.

Return value:

If a data entry that has an associated key is being found, then the returned iterator points at said entry, otherwise it points at a value equivalent to the return value of this->end ().

lower_bound ()

1)
```
iterator lower_bound (const key_type &rKey)
```
2)
```
const_iterator lower_bound (const key_type &rKey) const
```

Description:

These methods return the initial location of a set of data entries sharing the same key.

Input parameters:

rKey	- specifies key associated with data entry set

Remarks:

These methods are employed if the initial location of a set of data entries sharing the same key, specified by the input parameter rKey, has to be found.

Return value:

The return value is an iterator, which if the call was successful, is pointing at the initial location of the data entries, that have an associated key value equal to input parameter rKey, otherwise it points at the location of a data entry being associated with a key deemed to have the next greater key value.

upper_bound ()

1)
```
iterator upper_bound (const key_type &rKey)
```
2)
```
const_iterator upper_bound (const key_type &rKey) const
```

Description:

These methods return the location of an entry followed by a set of data entries sharing the same key.

Input parameters:

rKey	- specifies key associated with data entry set

Remarks:

These methods are employed if the location of a data entry following a set of data entries sharing the same key, specified by the input parameter rKey, has to be found.

Return value:

The return value is an iterator, which if the call was successful, is pointing at the location of a data entry followed by the data entries, that have an associated key value equal to input parameter rKey, otherwise it points at the location equivalent to this->end ().

equal_range ()

1)
```
::std::pair<iterator, iterator> equal_range (const key_type &rKey)
```
2)
```
::std::pair<const_iterator, const_iterator> equal_range (const key_type &rKey) const
```

Description:

These methods return the range of a specific key value.

Input parameters:

rKey	- specifies key associated with data entry set

Remarks:

These methods can be employed to obtain the range of data entries holding a specific key value. If a data entry containing the specified key value doesn’t exist, then both iterators returned refer to the data entry holding what is deemed to be the next higher key value.

Return value:

The returned pair value contains in its first element what lower_bound () would return and in the second element what upper_bound () would return with the input key value.

CBTreeMap and CBTreeMultiMap

extract_key ()

```
key_type *extract_key (key_type *pKey, const value_type &rData) const
```

Description:

This method extracts the key value from a data item.

Input parameter:

rData	- references data item where the key value is extracted from

Input- / Output parameter:

pKey	- specifies pointer where the extracted key value is written to

Remarks:

This method provides the functionality to extract a key value from a data item or convert a data item to its associated key value, exactly the way the data container does it before comparing keys, when ordering data entries.

Return value:

The return value is equal to the input parameter pKey, which allows this method to be used as an input parameter for other methods.

CBTreeMap

The C. class is a container type sorting data entries in an ascending order. Each data entry consists of a key value and a map value. While the key value is used as an operand for the sorting criteria, the map value is payload data associated with its key value. This container is only allowing for unique key values. This means that, if a data entry containing key A already exists, then inserting an additional data entry using the same key value A won't be successful.

Type Definitions

This container class is derived from CBTreeAssociativeBase, which one of the Abstract Data Classes as it can be seen in the Class Hierarchy diagram and defines the following types:

Type	**Definition**	**Description**
key_type	_t_keytype	This defines what type is used as an operand for the sorting criteria when determining the order of data entries stored by the container.
map_type	_t_maptype	This defines what type is used for the mapped data being associated with a key value of key_type.
value_type	typename ::std::pair<key_type, map_type>	This type defines what a data entry is and what is displayed to

Type	Definition	Description
		the application by the container when accessing it.
size_type	typename data_layer_properties_type::size_type	This defines what type is employed to address individual data entries when accessing a container as a linear array container type.
reference	value_type &	This type defines what a read / write reference of a data entry is.
const_reference	const value_type &	This type defines what a read-only reference of a data entry is.
pointer	value_type *	This type defines what a read / write pointer of a data entry is.
const_pointer	const value_type *	This type defines what a read-only pointer of a data entry is.
difference_type	typename ::std::make_signed<size_type>::type	This defines what types is used to display the distance between two data entries is.
data_layer_properties_type	_t_datalayerproperties	This type aids the definition of the types below.
sub_node_iter_type	typename data_layer_properties_type::sub_node_iter_type	This defines what type is used to address individual data entries within a node.
CBTreeAssociativeBase_t	CBTreeAssociativeBase<value_type, key_type, data_layer_properties_type>	This type aids the definition of the types below.
iterator	typename CBTreeAssociativeBase_t::iterator	This defines what is used as a read / write iterator type.
const_iterator	typename CBTreeAssociativeBase_t::const_iterator	This defines what is used as a read -only iterator type.
reverse_iterator	typename CBTreeAssociativeBase_t::reverse_iterator	This defines what is used as a read / write reverse iterator type.
const_reverse_iterator	typename CBTreeAssociativeBase_t::const_reverse_iterator	This defines what is used as a read -only reverse iterator type.

Type	Definition	Description
key_compare	struct key_compare_s<key_type>	This defines what type is in use to compare two key values of key_type using its operator ().
value_compare	struct value_compare_s<value_type>	This defines what type is in use to compare two data entries of value_type using its operator ().

Constructor

```
1)
CBTreeMap<_t_keytype, _t_maptype, _t_datalayerproperties>
(
        const data_layer_properties_type &rDataLayerProperties,
        const sub_node_iter_type nNodeSize

)
2)
CBTreeMap<_t_keytype, _t_maptype, _t_datalayerproperties>
(
        const CBTreeMap<_t_keytype, _t_maptype, _t_datalayerproperties> &rContainer,
        const bool bAssign = true
)
3)
CBTreeMap<_t_keytype, _t_maptype, _t_datalayerproperties>
(
        CBTreeMap_t &&rRhsContainer
)
```

Description:

These constructors initialise an object of type CBTreeMap.

Input parameter:

rDataLayerProperties	- references a data layer properties object, which must be compatible to the template parameter _t_datalayer
nNodeSize	- specifies node size parameter being used for this instance
rContainer	- references CBTreeMap instance of which to take a copy of
bAssign	- specifies, when taking a copy, if content will be copied across
rRhsContainer	- specifies right hand side reference to be moved

Remarks:

Constructor (1) initialises the CBTreeMap instance by calling the CBTreeKeySort Constructor. Since, the parameter set of this constructor and its CBTreeKeySort API counterpart are a perfect match, the parameters are only forwarded and the actual initialisation happens there.

As for the copy constructor (2), to set the parameter bAssign to false is useful if only the properties, rather than the content, have to be copied across. This is helpful prior to creating a sub-set of entries for instance. In case the compiler needs the code for a copy constructor, then the default parameter of bAssign = true applies.

Assignment Operator

1)
CBTreeMap<...> & operator= (const CBTreeMap<_t_keytype, _t_maptype, _t_datalayerproperties> &rContainer)

2)
CBTreeMap<...> & operator= (CBTreeMap<_t_keytype, _t_maptype, _t_datalayerproperties> &&rRhsContainer)

Description:

This operator copies the contents from a source CBTreeMap instance to a target instance.

Input parameter:

rContainer	- references instance to source CBTreeMap
rRhsContainer	- specifies right hand side reference to instance to be moved

Remarks:

This operator destroys all contents of the target CBTreeMap instance and then inserts all contents from the source CBTreeMap instance. In case `this` (target) and `&rContainer` (source) are pointing at the same instance, the call to this method is ineffective and returns immediately.

Return value:

The return value is a reference to the target CBTreeMap instance.

insert ()

```
1)
template <class _t_iterator>
void insert (_t_iterator sItFirst, _t_iterator sItLast)
2)
iterator insert (const value_type &rData)
3)
iterator insert (const_iterator sCIterHint, const value_type &rData)
```

Description:

These methods insert new data entries into the data container.

Input parameters:

sItFirst	- specifies iterator associated with initial element to be inserted
sItLast	- specifies iterator associated with subsequent of the last element to be inserted
rData	- specifies reference to data element to be inserted
sCIterHint	- indicates position near the place where data entry is to be inserted

Remarks:

These methods will insert one or more data entries into the target container, according to their parameters. In case sItFirst equals sItLast, the call is ineffective neither the stored data nor the state of the target container is being altered. Also, if a data entry displaying the same key value as any of the values being input already exists, then they are omitted. While method (1) does that silently, both method (2) and (3) are returning with an error state.

Method version (1) iterates element by element from sItFirst onwards until sItLast has been found. In case sItLast is not being found and the container associated with sItFirst goes beyond its last element, then an ::std::out_of_range exception is thrown.

Method (2) inserts one new data entry, while it depends on the key value the data entry generates where the new data entry is being inserted.

Method (3) inserts one new data entry, while trying to use sCIterHint as an indicator where that insertion has to take place, for an accelerated operation. If the position displayed by sCIterHint is insufficient to be used, then this method ignores sCIterHint and falls back to method (2).

Return value:

The return value of method (2) and (3) is an iterator, which is associated with the location where the newly added data entry has been inserted. If the method was not successful, because a data entry with the same key value already existed, then the return value is equal to what the method end () returns, once this method's call completed.

Note:

For method (1), in case the input iterators are referring to the target container itself, then the method acts as if it would take a copy of all elements between sItFirst and sItLast, to then insert those as if these were data elements from a different source. This means that, the method is capable of self-referencing, without destroying elements or ending up in an invalid state. However, if sItLast is before sItFirst or one of those input iterators is referring to a container other than the target container, then the behaviour is undefined.

emplace ()

```
1)
iterator emplace (value_type &&rrData)
2)
template<class ..._t_va_args>
iterator emplace (_t_va_args && ... rrArgs)
```

Description:

These methods insert a new data entry by constructing it in place.

Input parameters:

rrData	- specifies input for the move constructor to move an entry in place
rrArgs	- specifies constructor parameters to be forwarded to construct an entry in place

Remarks:

Both methods construct a new entry in place based on the constructor arguments, which increases the container size by one, unless an entry with the same key value already exists.

Return value:

Both methods return an iterator specifying the location where the new data entry has been constructed, if the operation was successful. Otherwise, a data entry with the same key already existed and a value equivalent to end () is returned.

emplace_hint ()

1)
iterator emplace_hint (const_iterator sCIterHint, value_type &&rData)

2)
template<class ..._t_va_args>
iterator emplace_hint (const_iterator sCIterHint, _t_va_args && ... rrArgs)

Description:

These methods insert a new data entry by constructing it in place, while trying to exploit a hint indicator for accelerated access.

Input parameters:

rrData	- specifies input for the move constructor to move an entry in place
rrArgs	- specifies constructor parameters to be forwarded to construct an entry in place
sCIterHint	- indicates position near the place where data entry is to be constructed

Remarks:

Both methods construct a new entry in place based on the constructor arguments, which increases the container size by one, while trying to use sCIterHint as an indicator where that insertion has to take place, for an accelerated operation. If the position displayed by sCIterHint is insufficient to be used, then both methods ignore sCIterHint and fall back to their emplace () equivalents. If a data entry with the same key value already existed, then neither a new data entry is constructed nor inserted, while the container remains in the same state.

Return value:

Both methods return an iterator specifying the location where the new data entry has been constructed, if the operation was successful. Otherwise, a data entry with the same key already existed and a value equivalent to end () is returned.

CBTreeMultiMap

The C. class is a container type sorting data entries in an ascending order. Each data entry consists of a key value and a map value. While the key value is used as an operand for the sorting criteria, the map value is payload data associated with its key value. This container is allowing for more than one data entry sharing the same key value. This means that, if a data entry containing key

A already exists, then inserting an additional data entry using the same key value A will result in said new data entry being appended to the same key value set of data entries already present in the data container.

Type Definitions

This container class is derived from CBTreeAssociativeBase, which one of the Abstract Data Classes as it can be seen in the Class Hierarchy diagram and defines the following types:

Type	Definition	Description
key_type	_t_keytype	This defines what type is used as an operand for the sorting criteria when determining the order of data entries stored by the container.
map_type	_t_maptype	This defines what type is used for the mapped data being associated with a key value of key_type.
value_type	typename ::std::pair<key_type, map_type>	This type defines what a data entry is and what is displayed to the application by the container when accessing it.
size_type	typename data_layer_properties_type::size_type	This defines what type is employed to address individual data entries when accessing a container as a linear array container type.
reference	value_type &	This type defines what a read / write reference of a data entry is.
const_reference	const value_type &	This type defines what a read-only reference of a data entry is.
pointer	value_type *	This type defines what a read / write pointer of a data entry is.
const_pointer	const value_type *	This type defines what a read-only pointer of a data entry is.

Type	Definition	Description
difference_type	typename ::std::make_signed<size_type>::type	This defines what types is used to display the distance between two data entries is.
data_layer_properties_type	_t_datalayerproperties	This type aids the definition of the types below.
sub_node_iter_type	typename data_layer_properties_type::sub_node_iter_type	This defines what type is used to address individual data entries within a node.
CBTreeAssociativeBase_t	CBTreeAssociativeBase<value_type, key_type, data_layer_properties_type>	This type aids the definition of the types below.
iterator	typename CBTreeAssociativeBase_t::iterator	This defines what is used as a read / write iterator type.
const_iterator	typename CBTreeAssociativeBase_t::const_iterator	This defines what is used as a read -only iterator type.
reverse_iterator	typename CBTreeAssociativeBase_t::reverse_iterator	This defines what is used as a read / write reverse iterator type.
const_reverse_iterator	typename CBTreeAssociativeBase_t::const_reverse_iterator	This defines what is used as a read -only reverse iterator type.
key_compare	struct key_compare_s<key_type>	This defines what type is in use to compare two key values of key_type using its operator ().
value_compare	struct value_compare_s<value_type>	This defines what type is in use to compare two data entries of value_type using its operator ().

Constructor

```
1)
CBTreeMultiMap<_t_keytype, _t_maptype, _t_datalayerproperties>
(
        const _t_datalayerproperties &rDataLayerProperties,
        const sub_node_iter_type nNodeSize
)
```

```
2)
CBTreeMultiMap<_t_keytype, _t_maptype, _t_datalayerproperties>
(
	const CBTreeMultiMap<_t_keytype, _t_maptype, _t_datalayerproperties> &rContainer,
	const bool bAssign = true
)
```

```
3)
CBTreeMultiMap<_t_keytype, _t_maptype, _t_datalayerproperties>
(
	CBTreeMultiMap_t &&rRhsContainer
)
```

Description:

These constructors initialise an object of type CBTreeMultiMap.

Input parameter:

rDataLayerProperties	- references a data layer properties object, which must be compatible to the template parameter _t_datalayer
nNodeSize	- specifies node size parameter being used for this instance
rContainer	- references CBTreeMultiMap instance of which to take a copy from
bAssign	- specifies, when taking a copy, if content will be copied across
rRhsContainer	- specifies right hand side reference of instance to be moved

Remarks:

Constructor (1) initialises the CBTreeMultiMap instance by calling the CBTreeKeySort Constructor. Since, the parameter set of this constructor and its CBTreeKeySort API counterpart are a perfect match, the parameters are only forwarded and the actual initialisation happens there.

As for the copy constructor (2), to set the parameter bAssign to false is useful if only the properties, rather than the content, have to be copied across. This is helpful prior to creating a sub-set of entries for instance. In case the compiler needs the code for a copy constructor, then the default parameter of bAssign = true applies.

Assignment Operator

```
1)
CBTreeMultiMap<...> & operator= (const CBTreeMultiMap<_t_keytype, _t_maptype,
_t_datalayerproperties> &rContainer)
```

```
2)
CBTreeMultiMap<...> & operator= (CBTreeMultiMap<_t_keytype, _t_maptype,
_t_datalayerproperties> &&rRhsContainer)
```

Description:

This operator copies the contents from a source CBTreeMultiMap instance to a target instance.

Input parameter:

rContainer	- references instance to source CBTreeMultiMap
rRhsContainer	- specifies right hand side reference of instance to be moved

Remarks:

This operator destroys all contents of the target CBTreeMultiMap instance and then inserts all contents from the source CBTreeMultiMap instance. In case `this` (target) and `&rContainer` (source) are pointing at the same instance, the call to this method is ineffective and returns immediately.

Return value:

The return value is a reference to the target CBTreeMultiMap instance.

insert ()

```
1)
template <class _t_iterator>
void insert (_t_iterator sItFirst, _t_iterator sItLast)
```

```
2)
iterator insert (const value_type &rData)
```

```
3)
iterator insert (const_iterator sCIterHint, const value_type &rData)
```

Description:

These methods insert new data elements into a target container.

Input parameters:

sItFirst	- specifies iterator associated with initial element to be inserted
sItLast	- specifies iterator associated with subsequent of the last element to be inserted
rData	- specifies reference to data element to be inserted

sCIterHint - indicates position near the place where data entry is to be inserted

Remarks:

These methods will insert one or more data items to the target container, according to their parameters. In case sItFirst equals sItLast, the call is ineffective neither the stored data nor the state of the target container is being altered.

Method (1) iterates element by element from sItFirst onwards until sItLast has been found. In case sItLast is not being found and the container associated with sItFirst goes beyond its last element, then an ::std::out_of_range exception is thrown.

Method (2) inserts one new data entry, while it depends on the key value the data entry generates where the new data entry is being inserted.

Method (3) inserts one new data entry, while trying to use sCIterHint as an indicator where that insertion has to take place, for an accelerated operation. If the position displayed by sCIterHint is insufficient to be used, then this method ignores sCIterHint and falls back to method (2).

Return value:

The return value of methods (2) and (3) is the position associating the location where the data set has been inserted.

Note:

For method (1), in case the input iterators are referring to the target container itself, then the method acts as if it would take a copy of all elements between sItFirst and sItLast, to then insert those as if these were data elements from a different source. This means that, the method is capable of self-referencing, without destroying elements or ending up in an invalid state. However, if sItLast is before sItFirst or one of those input iterators is referring to a container other than the target container, then the behaviour is undefined.

emplace ()

```
1)
iterator emplace (value_type &&rrData)
```

```
2)
template<class ..._t_va_args>
iterator emplace (_t_va_args && ... rrArgs)
```

Description:

These methods insert a new data entry by constructing it in place.

Input parameters:

rrData	- specifies input for the move constructor to move an entry in place
rrArgs	- specifies constructor parameters to be forwarded to construct an entry in place

Remarks:

Both methods construct a new entry in place based on the constructor arguments, which increases the container size by one.

Return value:

Both methods return an iterator specifying the location where the new data entry has been constructed.

emplace_hint ()

```
1)
iterator emplace_hint (const_iterator sCIterHint, value_type &&rData)
2)
template<class ..._t_va_args>
iterator emplace_hint (const_iterator sCIterHint, _t_va_args && ... rrArgs)
```

Description:

These methods insert a new data entry by constructing it in place, while trying to exploit a hint indicator for accelerated access.

Input parameters:

rrData	- specifies input for the move constructor to move an entry in place
rrArgs	- specifies constructor parameters to be forwarded to construct an entry in place
sCIterHint	- indicates position near the place where data entry is to be constructed

Remarks:

Both methods construct a new entry in place based on the constructor arguments, which increases the container size by one, while trying to use sCIterHint as an indicator where that insertion has to take place, for an accelerated operation. If the position displayed by sCIterHint is insufficient to be used, then both methods ignore sCIterHint and fall back to their emplace () equivalents.

Return value:

Both methods return an iterator specifying the location where the new data entry has been constructed.

CBTreeSet

The C. class is a container type sorting data entries in an ascending order. Each data entry is a key value, which is used as an operand for the sorting criteria, This container is only allowing for unique key values. This means that, if a data entry containing key A already exists, then inserting an additional data entry using the same key value A won't be successful.

Type Definitions

This container class is derived from CBTreeAssociativeBase, which one of the Abstract Data Classes as it can be seen in the Class Hierarchy diagram and defines the following types:

Type	Definition	Description
key_type	_t_keytype	This defines what type is used as an operand for the sorting criteria when determining the order of data entries stored by the container.
value_type	_t_keytype	This type defines what a data entry is and what is displayed to the application by the container when accessing it.
size_type	typename data_layer_properties_type::size_type	This defines what type is employed to address individual data entries when accessing a

Type	Definition	Description
		container as a linear array container type.
reference	value_type &	This type defines what a read / write reference of a data entry is.
const_reference	const value_type &	This type defines what a read-only reference of a data entry is.
pointer	value_type *	This type defines what a read / write pointer of a data entry is.
const_pointer	const value_type *	This type defines what a read-only pointer of a data entry is.
difference_type	typename ::std::make_signed<size_type>::type	This defines what types is used to display the distance between two data entries is.
CBTreeAssociativeBase_t	CBTreeAssociativeBase<value_type, key_type, data_layer_properties_type>	This type aids the definition of the types below.
iterator	typename CBTreeAssociativeBase_t::iterator	This defines what is used as a read / write iterator type.
const_iterator	typename CBTreeAssociativeBase_t::const_iterator	This defines what is used as a read -only iterator type.
reverse_iterator	typename CBTreeAssociativeBase_t::reverse_iterator	This defines what is used as a read / write reverse iterator type.
const_reverse_iterator	typename CBTreeAssociativeBase_t::const_reverse_iterator	This defines what is used as a read -only reverse iterator type.
key_compare	struct key_compare_s<key_type>	This defines what type is in use to compare two key values of key_type using its operator ().
value_compare	struct value_compare_s<value_type>	This defines what type is in use to compare two data entries of value_type using its operator ().

Constructor

```
1)
CBTreeSet<_t_keytype, _t_datalayerproperties>
(
	const data_layer_properties_type &rDataLayerProperties,
	const sub_node_iter_type nNodeSize
)
2)
CBTreeSet<_t_keytype, _t_datalayerproperties>
(
	const CBTreeSet<_t_keytype, _t_datalayerproperties> &rContainer,
	const bool bAssign = true
)
3)
CBTreeSet<_t_keytype, _t_datalayerproperties>
(
	CBTreeSet_t &&rRhsContainer
)
```

Description:

These constructors initialise an object of type CBTreeSet.

Input parameter:

rDataLayerProperties	- references a data layer properties object, which must be compatible to the template parameter _t_datalayer
nNodeSize	- specifies node size parameter being used for this instance
rContainer	- references CBTreeSet instance of which to take a copy from
bAssign	- specifies, when taking a copy, if content will be copied across
rRhsContainer	- specifies right hand side reference of instance to be moved

Remarks:

Constructor (1) initialises the CBTreeSet instance by calling the CBTreeKeySort Constructor. Since, the parameter set of this constructor and its CBTreeKeySort API counterpart are a perfect match, the parameters are only forwarded and the actual initialisation happens there.

As for the copy constructor (2), to set the parameter bAssign to false is useful if only the properties, rather than the content, have to be copied across. This is helpful prior to creating a sub-set of entries for instance. In case the compiler needs the code for a copy constructor, then the default parameter of bAssign = true applies.

Assignment Operator

1)
CBTreeSet<...> & operator= (const CBTreeSet<_t_keytype, _t_datalayerproperties> &rContainer)

2)
CBTreeSet<...> & operator= (CBTreeSet<_t_keytype, _t_datalayerproperties> &&rRhsContainer)

Description:

This operator copies the contents from a source CBTreeSet instance to a target instance.

Input parameter:

rContainer	- references instance to source CBTreeSet
rRhsContainer	- specifies right hand side reference of instance to be moved

Remarks:

This operator destroys all contents of the target CBTreeSet instance and then inserts all contents from the source CBTreeSet instance. In case `this` (target) and `&rContainer` (source) are pointing at the same instance, the call to this method is ineffective and returns immediately.

Return value:

The return value is a reference to the target CBTreeSet instance.

insert ()

1)
template <class _t_iterator>
void insert (_t_iterator sItFirst, _t_iterator sItLast)

2)
iterator insert (const value_type &rData)

3)
iterator insert (const_iterator sCIterHint, const value_type &rData)

Description:

These methods insert new data entries into the data container.

Input parameters:

sItFirst	- specifies iterator associated with initial element to be inserted
sItLast	- specifies iterator associated with subsequent of the last element to be inserted
rData	- specifies reference to data element to be inserted
sCIterHint	- indicates position near the place where data entry is to be inserted

Remarks:

These methods will insert one or more data entries into the target container, according to their parameters. In case sItFirst equals sItLast, the call is ineffective neither the stored data nor the state of the target container is being altered. Also, if a data entry displaying the same key value as any of the values being input already exists, then they are omitted. While method (1) does that silently, both method (2) and (3) are returning with an error state.

Method version (1) iterates element by element from sItFirst onwards until sItLast has been found. In case sItLast is not being found and the container associated with sItFirst goes beyond its last element, then an ::std::out_of_range exception is thrown.

Method (2) inserts one new data entry, while it depends on the key value the data entry generates where the new data entry is being inserted.

Method (3) inserts one new data entry, while trying to use sCIterHint as an indicator where that insertion has to take place, for an accelerated operation. If the position displayed by sCIterHint is insufficient to be used, then this method ignores sCIterHint and falls back to method (2).

Return value:

The return value of method (2) and (3) is an iterator, which is associated with the location where the newly added data entry has been inserted. If the method was not successful, because a data entry with the same key value already existed, then the return value is equal to what the method end () returns, once this method's call completed.

Note:

For method (1), in case the input iterators are referring to the target container itself, then the method acts as if it would take a copy of all elements between sItFirst and sItLast, to then insert those as if these were data elements from a different source. This means that, the method is capable of self-referencing, without destroying elements or ending up in an invalid state. However, if sItLast is before sItFirst or one of those input iterators is referring to a container other than the target container, then the behaviour is undefined.

emplace ()

```
1)
iterator emplace (value_type &&rrData)
2)
template<class ..._t_va_args>
iterator emplace (_t_va_args && ... rrArgs)
```

Description:

These methods insert a new data entry by constructing it in place.

Input parameters:

rrData	- specifies input for the move constructor to move an entry in place
rrArgs	- specifies constructor parameters to be forwarded to construct an entry in place

Remarks:

Both methods construct a new entry in place based on the constructor arguments, which increases the container size by one, unless an entry with the same key value already exists.

Return value:

Both methods return an iterator specifying the location where the new data entry has been constructed, if the operation was successful. Otherwise, a data entry with the same key already existed and a value equivalent to end () is returned.

emplace_hint ()

```
1)
iterator emplace_hint (const_iterator sCIterHint, value_type &&rData)
2)
template<class ..._t_va_args>
iterator emplace_hint (const_iterator sCIterHint, _t_va_args && ... rrArgs)
```

Description:

These methods insert a new data entry by constructing it in place, while trying to exploit a hint indicator for accelerated access.

Input parameters:

rrData	- specifies input for the move constructor to move an entry in place
rrArgs	- specifies constructor parameters to be forwarded to construct an entry in place
sCIterHint	- indicates position near the place where data entry is to be constructed

Remarks:

Both methods construct a new entry in place based on the constructor arguments, which increases the container size by one, while trying to use sCIterHint as an indicator where that insertion has to take place, for an accelerated operation. If the position displayed by sCIterHint is insufficient to be used, then both methods ignore sCIterHint and fall back to their emplace () equivalents. If a data entry with the same key value already existed, then neither a new data entry is constructed nor inserted, while the container remains in the same state.

Return value:

Both methods return an iterator specifying the location where the new data entry has been constructed, if the operation was successful. Otherwise, a data entry with the same key already existed and a value equivalent to end () is returned.

CBTreeMultiSet

The C. class is a container type sorting data entries in an ascending order. Each data entry is a key value, which is used as an operand for the sorting criteria. This container is allowing for more than one data entry sharing the same key value. This means that, if a data entry containing key A already exists, then inserting an additional data entry using the same key value A will result in said new data entry being appended to the same key value set of data entries already present in the data container.

Type Definitions

This container class is derived from CBTreeAssociativeBase, which one of the Abstract Data Classes as it can be seen in the Class Hierarchy diagram and defines the following types:

Type	Definition	Description
key_type	_t_keytype	This defines what type is used as an operand for the sorting criteria when determining the order of data entries stored by the container.
value_type	_t_keytype	This type defines what a data entry is and what is displayed to the application by the container when accessing it.
size_type	typename data_layer_properties_type::size_type	This defines what type is employed to address individual data entries when accessing a container as a linear array container type.
reference	value_type &	This type defines what a read / write reference of a data entry is.
const_reference	const value_type &	This type defines what a read-only reference of a data entry is.
pointer	value_type *	This type defines what a read / write pointer of a data entry is.
const_pointer	const value_type *	This type defines what a read-only pointer of a data entry is.
difference_type	typename ::std::make_signed<size_type>::type	This defines what types is used to display the distance between two data entries is.
CBTreeAssociativeBase_t	CBTreeAssociativeBase<value_type, key_type, data_layer_properties_type>	This type aids the definition of the types below.
iterator	typename CBTreeAssociativeBase_t::iterator	This defines what is used as a read / write iterator type.
const_iterator	typename CBTreeAssociativeBase_t::const_iterator	This defines what is used as a read -only iterator type.
reverse_iterator	typename CBTreeAssociativeBase_t::reverse_iterator	This defines what is used as a read / write reverse iterator type.
const_reverse_iterator	typename CBTreeAssociativeBase_t::const_reve	This defines what is used as a read

Type	Definition	Description
	rse_iterator	-only reverse iterator type.
key_compare	struct key_compare_s<key_type>	This defines what type is in use to compare two key values of key_type using its operator ().
value_compare	struct value_compare_s<value_type>	This defines what type is in use to compare two data entries of value_type using its operator ().

Constructor

```
1)
CBTreeMultiSet<_t_keytype, _t_datalayerproperties>
(
	const data_layer_properties_type &rDataLayerProperties,
	const sub_node_iter_type nNodeSize
)
2)
CBTreeMultiSet<_t_keytype, _t_datalayerproperties>
(
	const CBTreeMultiSet<_t_keytype, _t_datalayerproperties> &rContainer,
	const bool bAssign = true
)
3)
CBTreeMultiSet<_t_keytype, _t_datalayerproperties>
(
	CBTreeMultiSet<_t_keytype, _t_datalayerproperties> &&rRhsContainer
)
```

Description:

These constructors initialise an object of type CBTreeMultiSet.

Input parameter:

rDataLayerProperties	- references a data layer properties object, which must be compatible to the template parameter _t_datalayer
nNodeSize	- specifies node size parameter being used for this instance
rContainer	- references CBTreeMultiSet instance of which to take a copy from
bAssign	- specifies, when taking a copy, if content will be copied across
rRhsContainer	- specifies right hand side reference of instance to be moved

Remarks:

Constructors (1) initialises the CBTreeMultiSet instance by calling the CBTreeKeySort Constructor. Since, the parameter set of this constructor and its CBTreeKeySort API counterpart are a perfect match, the parameters are only forwarded and the actual initialisation happens there.

As for the copy constructor (2), to set the parameter bAssign to false is useful if only the properties, rather than the content, have to be copied across. This is helpful prior to creating a sub-set of entries for instance. In case the compiler needs the code for a copy constructor, then the default parameter of bAssign = true applies.

Assignment Operator

```
1)
CBTreeMultiSet<...> & operator= (const CBTreeMultiSet<_t_keytype, _t_datalayerproperties>
&rContainer)
2)
CBTreeMultiSet<...> & operator= (CBTreeMultiSet<_t_keytype, _t_datalayerproperties>
&&rRhsContainer)
```

Description:

This operator copies the contents from a source CBTreeMultiSet instance to a target instance.

Input parameter:

rContainer	- references instance to source CBTreeMultiSet
rRhsContainer	- specifies right hand side reference of instance to be moved

Remarks:

This operator destroys all contents of the target CBTreeMultiSet instance and then inserts all contents from the source CBTreeMultiSet instance. In case `this` (target) and `&rContainer` (source) are pointing at the same instance, the call to this method is ineffective and returns immediately.

Return value:

The return value is a reference to the target CBTreeMultiSet instance.

insert ()

```
1)
template <class _t_iterator>
void insert (_t_iterator sItFirst, _t_iterator sItLast)
2)
iterator insert (const value_type &rData)
3)
iterator insert (const_iterator sCIterHint, const value_type &rData)
```

Description:

These methods insert new data elements into a target container.

Input parameters:

sItFirst	- specifies iterator associated with initial element to be inserted
sItLast	- specifies iterator associated with subsequent of the last element to be inserted
rData	- specifies reference to data element to be inserted
sCIterHint	- indicates position near the place where data entry is to be inserted

Remarks:

These methods will insert one or more data items to the target container, according to their parameters. In case sItFirst equals sItLast, the call is ineffective neither the stored data nor the state of the target container is being altered.

Method (1) iterates element by element from sItFirst onwards until sItLast has been found. In case sItLast is not being found and the container associated with sItFirst goes beyond its last element, then an ::std::out_of_range exception is thrown.

Method (2) inserts one new data entry, while it depends on the key value the data entry generates where the new data entry is being inserted.

Method (3) inserts one new data entry, while trying to use sCIterHint as an indicator where that insertion has to take place, for an accelerated operation. If the position displayed by sCIterHint is insufficient to be used, then this method ignores sCIterHint and falls back to method (2).

Return value:

The return value of methods (2) and (3) is the position associating the location where the data set has been inserted.

Note:

For method (1), in case the input iterators are referring to the target container itself, then the method acts as if it would take a copy of all elements between sItFirst and sItLast, to then insert those as if these were data elements from a different source. This means that, the method is capable of self-referencing, without destroying elements or ending up in an invalid state. However, if sItLast is before sItFirst or one of those input iterators is referring to a container other than the target container, then the behaviour is undefined.

emplace ()

```
1)
iterator emplace (value_type &&rrData)
2)
template<class ..._t_va_args>
iterator emplace (_t_va_args && ... rrArgs)
```

Description:

These methods insert a new data entry by constructing it in place.

Input parameters:

rrData	- specifies input for the move constructor to move an entry in place
rrArgs	- specifies constructor parameters to be forwarded to construct an entry in place

Remarks:

Both methods construct a new entry in place based on the constructor arguments, which increases the container size by one.

Return value:

Both methods return an iterator specifying the location where the new data entry has been constructed.

emplace_hint ()

1)
```
iterator emplace_hint (const_iterator sCIterHint, value_type &&rData)
```

2)
```
template<class ..._t_va_args>
iterator emplace_hint (const_iterator sCIterHint, _t_va_args && ... rrArgs)
```

Description:

These methods insert a new data entry by constructing it in place, while trying to exploit a hint indicator for accelerated access.

Input parameters:

rrData	- specifies input for the move constructor to move an entry in place
rrArgs	- specifies constructor parameters to be forwarded to construct an entry in place
sCIterHint	- indicates position near the place where data entry is to be constructed

Remarks:

Both methods construct a new entry in place based on the constructor arguments, which increases the container size by one, while trying to use sCIterHint as an indicator where that insertion has to take place, for an accelerated operation. If the position displayed by sCIterHint is insufficient to be used, then both methods ignore sCIterHint and fall back to their emplace () equivalents.

Return value:

Both methods return an iterator specifying the location where the new data entry has been constructed.

Iterators

This section shows what iterator types are available and what methods exist to have those interact with an application. Also, a sub-section is dedicated to a data read / write return type used by non-const iterator types, which can be found here: CBTreeIteratorSubScriptWrapper

I. are capable of fast forwarding and rewinding to any location by using its overloaded arithmetic operators. It is legal to make the iterator end up pointing to a location outside of its associated container's range for as long as an access is not taking place while being in that state. This is possible due to the lazy evaluation approach this b-tree framework iterator types are exercising. This means that, I. have an external pointer, which is modified by the application or initialised by a container, and also it has an internal pointer being associated with the location within the b-tree structure. Once an access is taking place, the external pointer needs to be in range, otherwise an ::std::out_of_range exception is thrown. If the external pointer is in range, then the external and internal pointer are checked to be equal. In case they are equal, then the iterator was evaluated before, so that a re-evaluation is not needed, otherwise the internal pointer is re-evaluated to the external pointer's location and the iterator tries to find the quickest way to do so. When an iterator is initialised, then the internal pointer and the time stamp counter are set to an invalid state to make sure an iterator is evaluated on the first access.

To ensure that an iterator and its associated container are always synchronised, both have a time stamp, which has two components:

- a container access counter
- the time when the container was last modified

The access counter is incremented every time the container is modified in terms of size (i.e. when data is inserted or removed). If that is the case, then also the last access time is updated. When an iterator requires to access its associated data container and time stamps are not equal, then the iterator has to be fully re-evaluated, since after a modification it is not guaranteed that the iterator is still pointing at the correct location or that the location still exists. Also it is made sure that the iterator is still associated with the container in question, because there is the potential that an iterator can be re-initialised with a different container.

CBTreeConstIterator

The C. is utilised to read-only access data sets it is pointing at and has one template parameter called _ti_container, which is the type of data container to be used when performing operations. This container type must provide a number of type definitions, as described in section

Type Definitions, for this iterator type to work. Furthermore, C. is inheriting from ::std::iterator with its template parameters set up as follows:

_Category	- ::std::random_access_iterator_tag
_Ty	- typename _ti_container::value_type
_Diff	- typename _ti_container::difference_type
_Pointer	- typename _ti_container::pointer
_Reference	- typename _ti_container::reference

Type Definitions

All iterators only have one template parameter, which is the type of container the iterator is going to be used for. This template parameter is called _ti_container and has to provide a number of definitions as shown in the table below:

Type	Definition	Description
container_t	_ti_container	This definition is the data container type that will be using the iterator type in question.
value_type	typename container_t::value_type	This type defines what a data entry is and what is displayed to the application by the container when accessing it.
size_type	typename container_t::size_type	This defines what type is employed to address individual data entries when accessing a container as a linear array container type.
difference_type	typename container_t::difference_type	This defines what types is used to display the distance between two data entries is.
reference	typename container_t::reference	This type defines what a read / write reference of a data entry is.
pointer	typename container_t::pointer	This type defines what a read / write pointer of a data entry is.

Constructor

1)
CBTreeConstIterator<_ti_container> ()

2)
CBTreeConstIterator<_ti_container> (_ti_container *pContainer, sizetype_t nPos,
bool bRegister = true)

3)
CBTreeConstIterator<_ti_container> (_ti_container *pContainer, sizetype_t nPos,
nodeiter_t nNode, subnodeiter_t nSub, btree_time_stamp_t sTimeStamp, bool bRegister = true)

4)
CBTreeConstIterator<_ti_container> (const CBTreeConstIterator &rIter, bool bRegister = true)

5)
CBTreeConstIterator<_ti_container> (CBTreeConstIterator && rIter)

6)
CBTreeConstIterator<_ti_container> (const CBTreeIterator<_ti_container> &rIter, bool bRegister =
true)

Description:

These constructors initialise a b-tree framework read-only iterator.

Input parameters:

pContainer	- specifies pointer to container this iterator will be associated with
nPos	- specifies absolute linear position this iterator is to be initialised with
nNode	- specifies node of node / entry pair associated with absolute linear position
nSub	- specifies entry of node / entry pair associated with absolute linear position
sTimeStamp	- specifies initial time stamp this iterator will be set up with
rIter	- references iterator instance to take a copy of or to move construct
bRegister	- specifies if this iterator will be registered with its associated b-tree instance

Remarks:

These constructors initialise an iterator instance to be used with a b-tree container. The only exception is the default constructor (1), which only sets the iterator to a disassociated state. So, this iterator has to be initialised, using its Assignment Operators, before any valid access can take place. Constructor (2) only sets up the external pointer to nPos and due to the lazy evaluation approach, the b-tree iterators exercise, the internal pointer is not evaluated on construction. This is useful if further arithmetic operations are performed with the newly created instance and an immediate access is not required.

To fully initialise an iterator, the next variant (3) has to be used. The external and internal pointer are set to nPos, while the nNode and nSub have to display the correct node / entry pair associated with the internal pointer. This way of constructing an iterator is helpful if the evaluation took place externally and an immediate access is due.

The copy constructors (4, 6), do as the name already says. They take a copy of the state of rIter, may that be evaluated or not, and initialises this iterator with the taken copy.

The move constructor (5), swaps the state of the newly construct but not evaluated iterator with the state the input iterator rIter. This allows the compiler to generate faster code in order to quickly move temporary instances.

Since an iterator can be associated with a container, it has to be ensured that a container exists, once an iterator is using said container for an access. An iterator itself cannot tell if the container it is associated with is still present. Hence, a container instance has to flag to the iterators, which are registered with it, that it is about to be destroyed. If the parameter bRegister is set to true, then the newly created iterator is registering itself with the container instance being pointed at by pContainer. Setting bRegister to false skips the registration and is useful if an iterator only exists temporarily.

Note:

For the registration to work probably the code must be compiled with the build flag BTREE_ITERATOR_REGISTRATION. Building the code without that flag is helpful to create faster code, since the registration can be time consuming in case a large number of iterators is registered with a few number of containers. However, the application or any class using the iterators has to make sure any iterator isn't accessing a de-initialised container instance.

detach ()

```
void detach ()
```

Description:

This method detaches an iterator from a container instance.

Remarks:

In case the target iterator's b-tree instance is about to be destroyed, the container instance's destructor calls this method to reset the member variable m_pContainer to NULL and invalidates the time stamp. This ensures that the iterator is not trying to access a b-tree instance, which no longer exists.

Note:

This method isn't supposed to be called by an application and ought to be a protected member. It has been made public though, since the container type (defined by _ti_container) needs access to this method, when the container instance is being destroyed. With C++11 it isn't possible to friend template parameters, which means, that the type, displayed by the template parameter _ti_container, can only access public members.

swap ()

```
void swap (CBTreeConstIterator &rRight)
```

Description:

This method swaps the state of two iterators.

Input parameters:

rRight	- references iterator which state is to be swapped with the target iterator's state

Remarks:

This method swaps the state of two iterator instances. If any of the involved iterators is registered with a b-tree instance, then those registrations are swapped too, unless both iterators are registered with the same b-tree instance or not registered at all. The call of this method is ineffective, in case rRight is the same instance as “*this”.

Assignment Operators

```
1)
CBTreeConstIterator& operator= (const CBTreeConstIterator &rIter)
2)
CBTreeConstIterator& operator= (const CBTreeIterator<_ti_container> &rIter)
```

Description:

These operators copy the state of a source iterator to this instance.

Input parameters:

rIter	- reference to source iterator instance

Remarks:

To take a copy of an iterator instance, these operators have to be used. If the source iterator rIter is associated with a different b-tree container than the target iterator, then the assignment operator unregisters the target iterator from its associated b-tree container. In any event, the operator then copies the state of the external iterator across. If the target iterator was previously unregistered, then it is registered with the new b-tree instance.

Return value:

The return value is a reference to the updated target iterator.

Move Assignment Operator

```
CBTreeConstIterator& operator= (CBTreeConstIterator &&rIter)
```

Description:

This operator swaps the state of a source iterator with the state of this instance.

Input parameters:

rIter	- reference to source iterator instance

Remarks:

In order to move a temporary iterator instance, this operator can be exploited by the compiler. If the source iterator rIter is associated with a different b-tree container than the target iterator, then the assignment operator unregisters the target iterator from its associated b-tree container. In any event, the operator then swaps the state of the external iterator with the state of this iterator instance. If the target iterator was previously unregistered, then it is registered with the new b-tree instance.

Return value:

The return value is a reference to the updated target iterator.

Comparison Operators

1)
```
bool operator== (const CBTreeConstIterator &rIter)
```
2)
```
bool operator!= (const CBTreeConstIterator &rIter)
```

Description:

These operators tell if two iterators display the same offset or not.

Input parameters:

rIter	- reference to external iterator instance

Remarks:

These operators compare the offset of this and an external iterator and return whether they are equal or not. It doesn't matter if those iterators are associated with different container instances. However, if one or both iterators are not registered with a container instance (ie. m_pContainer is set to NULL), then an ::std::runtimeerror exception is thrown. If both this and the external iterator are the same instance, then the result is always displayed as equal, even if said instance is not fully initialised

Return value:

As for operator (1), if the result is deemed to be equal, then true is returned, otherwise false. Operator (2) returns the exact opposite of what operator (1) returns.

Indirection Operator

```
value_type & operator* ()
```

Description:

This operator provides pseudo read-only access to the data item the iterator is pointing at.

Remarks:

To access a data item within the container the iterator is associated with, the indirection operator has to be used. If the iterator is pointing to a location outside of its associated container, then an

::std::out_of_range exception is thrown. The returned reference is non-const, but if it is being written to, then the write is ineffective and only a buffer being part of the iterator is modified. This modification is discarded once the next access takes place, even if the iterator has not been moved in the meantime.

If the iterator is not associated with a data container, then an ::std::runtime_error exception is thrown.

Return value:

The return value is a reference to the data element the iterator is pointing at.

De-reference Operator

```
value_type * operator-> ()
```

Description:

This operator provides a pointer to the data element the iterator is pointing at.

Remarks:

The de-reference operator provides a pointer the data element the iterator is pointing at. Since this a const iterator providing read-only access, any writes would modify an internal copy of the data element in question, as oppose to altering the actual data in the b-tree structure. If that copy is modified, then this modification is discarded once the next access takes place, even if the iterator is not been moved bin the meantime. In case the iterator is pointing to a location that exceeds the container, then an ::std::out_of_range exception is thrown.

If the iterator is not associated with a data container, then an ::std::runtime_error exception is thrown.

Return value:

The return value is a pointer to the data element the iterator is pointing at.

Unary Operators

1)
CBTreeConstIterator & operator++ ()

2)
CBTreeConstIterator   operator++ (int)

3)
CBTreeConstIterator & operator-- ()

4)
CBTreeConstIterator   operator-- (int)

Description:

These operators move the location an iterator is pointing at by one position forward or backward.

Input parameters:

<name less>   - if present, then it specifies that the operator is postfix

Remarks:

To increment or decrement the location an iterator is pointing at by one, these operators are being used. Since these operators don't access the container associated with the iterator, the resulting location may exceed the container range, without an exception being thrown.

Return value:

For prefix operators (1, 3), the return value is a reference to the updated iterator instance. As oppose to postfix operators (2, 4), where the return value is a copy of the iterator instance prior to its modification.

Compound Operators

1)
CBTreeConstIterator & operator+= (const int nOperand)

2)
CBTreeConstIterator & operator+= (const sizetype_t nOperand)

3)
CBTreeConstIterator & operator+= (const CBTreeConstIterator &rIter)

4)
CBTreeConstIterator & operator-= (const int nOperand)

5)

```
CBTreeConstIterator & operator-= (const sizetype_t nOperand)
```
6)
```
CBTreeConstIterator & operator-= (const CBTreeConstIterator &rIter)
```

Description:

These operators move an iterator forward or backward by an arbitrary distance.

Input parameter:

nOperand	- specifies distance the location the iterator is pointing at will be moved
rIter	- reference to external iterator displaying distance the target iterator will be moved

Remarks:

If the location an iterator is pointing at has to be moved by a variable distance, then these operators may be used. The operators (1, 4) allow for negative inputs and therefore are able to invert their operation in case the input is indeed negative. This means that, if an iterator is incremented by a negative number, then the result is equivalent to said iterator being decremented by that number's absolute value and vice versa. In case the magnitude of nOperand is deemed to be zero, then calling these operators is ineffective. Since these operators are not accessing the container associated with the target iterator, the resulting iterator may exceed the container's range, without an exception being thrown. However, if the target iterator is not associated with a container instance, then an ::std::runtime_error exception is thrown.

In case operator (3, 6) are being used, rIter doesn't need to be associated with the same container as the target iterator is. Basically, the external pointer of rIter is extracted and then used as if operator (2, 5) has been called.

Return value:

The return value is always a reference to the updated iterator instance.

Binary Operators

1)
```
const CBTreeConstIterator operator+ (const int nOperand) const
```
2)
```
const CBTreeConstIterator operator+ (const sizetype_t nOperand) const
```
3)
```
const CBTreeConstIterator operator+ (const CBTreeConstIterator &rIter) const
```
4)

```
const CBTreeConstIterator operator- (const int nOperand) const
5)
const CBTreeConstIterator operator- (const sizetype_t nOperand) const
6)
const sizetype_t operator- (const CBTreeConstIterator &rIter) const
```

Description:

The operators (1-5) calculate the location a newly created iterator will be pointing at and return the new iterator instance. Operator (6) determines the distance between two iterators.

Input parameter:

nOperand	- specifies distance the location the iterator is pointing at will be moved
rIter	- reference to external iterator displaying distance the target iterator will be moved

Remarks:

If the location an iterator is pointing at has to be moved by a variable distance, then these operators may be used. Operators (1, 4) that allow for negative inputs are able to invert their operation in case the input is indeed negative. This means that, if an iterator is incremented by a negative number, then the result is equivalent to said iterator being decremented by that number's absolute value and vice versa. In case the magnitude of nOperand is deemed to be zero, then calling these operators is ineffective, except for operator (6), where the result would be an equivalent of the target iterator's external pointer. Since these operators are not accessing the container associated with the target iterator, the resulting iterator may exceed the container's range, without an exception being thrown. However, if the target iterator is not associated with a container instance, then an ::std::runtime_error exception is thrown.

In case operator (3) is being used, rIter doesn't need to be associated with the same container as the target iterator is. Basically, the external pointer of rIter is extracted and then used as if operator (2) has been called.

Return value:

The return value is a newly created iterator instance pointing at the result of its respective operation, except for operator (6), where the result displayed the distance between two iterators.

Note:

Programmers are advised not to use operator (6) directly and instead use ::std::distance to determine the difference between to iterators. Operator (6) solely exists, so that ::std::distance is able to make use of it.

Equality Operators

```
1)
bool operator< (const CBTreeConstIterator &rIter)
2)
bool operator<= (const CBTreeConstIterator &rIter)
3)
bool operator> (const CBTreeConstIterator &rIter)
4)
bool operator>= (const CBTreeConstIterator &rIter)
```

Description:

To determine the equality between this and an external iterator, these operators are to be used.

Input parameters:

rIter	- references the external iterator to compare against

Remarks:

These operators determine if the external iterator is greater or smaller than this iterator. Only the external pointer of both iterators is employed to determine the result. This means that, both this and the external iterator may be outside their respective container's range, without an exception being thrown, since none of the containers is being accessed. Also, this and the external iterator don't need to be associated with the same container. However, if this or the external iterator is not associated with a container instance, then an ::std::runtime_error exception is thrown.

Return value:

Operator (1) returns true if this iterator is deemed to be smaller than the external iterator, otherwise false.

Operator (2) returns true if this iterator is deemed to be smaller than or equal to the external iterator, otherwise false.

Operator (3) returns true if this iterator is deemed to be greater than the external iterator, otherwise false.

Operator (4) returns true if this iterator is deemed to be greater than or equal to the external iterator, otherwise false.

Sub-Script Operators

1)
```
value_type & operator[] (const int nPos)
```
2)
```
value_type & operator[] (const sizetype_t nPos)
```

Description:

These operators provide pseudo read-only access to a data item relative to the position the iterator is pointing at.

Input parameters:

nPos	- specifies relative position the iterator is currently pointing at

Remarks:

To access a data item within the container the iterator is associated with, the sub-script operators may be used. The current location the iterator is pointing at is combined with the input parameter nPos to determine what data item will be accessed. If the result is pointing to a location outside of its associated container, then an ::std::out_of_range exception is thrown. If the location the iterator is pointing at, is outside of its associated container, but in combination with nPos results in a valid location not exceeding said container again, then the access is valid and an exception is not thrown. The returned reference is non-const, but if it is being written to, then the write is ineffective and only a buffer being part of the iterator is modified. This modification is discarded once the next access takes place, even if the iterator is not moved in the meantime and nPos remains the same.

The input parameter nPos of operator (1) may be negative and due to the subtraction of nPos from the iterator's current position, results in a location prior to what the iterator is pointing at.

Return value:

The return value is a reference to the data element the iterator is pointing at and the input parameter nPos combined.

get_container ()

```
_ti_container* get_container ()
```

Description:

This method returns a pointer to the data container associated with the iterator.

Remarks:

To retrieve a pointer to the data container instance currently being associated with the iterator, this method has to be called. In case the iterator isn't associated with a data container, the return value is NULL.

Return value:

If the iterator currently is associated with a data container then a pointer that container is returned, otherwise the returned pointer is NULL.

Note:

This method isn't supposed to be called by an application and ought to be a protected member. It has been made public though, since the container type (defined by _ti_container) needs access to this method, when the container instance is being destroyed. With C++11 it isn't possible to friend template parameters, which means, that the type, displayed by the template parameter _ti_container, can only access public members.

is_btree_valid ()

```
bool is_btree_valid ()
```

Description:

This method determines if the iterator is associated with a data container.

Remarks:

To determine if an iterator is currently associated with a data container, this method has to be called.

Return value:

If the iterator is associated with a data container then the return value is true, otherwise false.

Note:

This method isn't supposed to be called by an application and ought to be a protected member. It has been made public though, since the container type (defined by _ti_container) needs access to this method, when the container instance is being destroyed. With C++11 it isn't possible to friend template parameters, which means, that the type, displayed by the template parameter _ti_container, can only access public members.

is_evaluated ()

```
bool is_evaluated ()
```

Description:

This method determines if the next access to the data element the iterator is pointing at, will result in an evaluation.

Remarks:

This method tests if the time stamp of the iterator and its associated data container are the same and if the external and internal pointer of the iterator are equal. If either of those tests fails, then the next data access requires an evaluation. This means that, on the next data access the iterator has to determine what the new node / sub-position pair inside the data container associated with the external pointer is.

Return value:

The return value is true, if the next access to the data container doesn't require the iterator to evaluate the new node / sub-position pair, otherwise false.

Note:

This method isn't supposed to be called by an application and ought to be a protected member. It has been made public though, since the container type (defined by _ti_container) needs access to this method, when the container instance is being destroyed. With C++11 it isn't possible to friend template parameters, which means, that the type, displayed by the template parameter _ti_container, can only access public members.

sync ()

`void sync ()`

Description:

This method synchronises the iterator with its associated data container.

Remarks:

The method synchronises the iterator with its associated data container, by evaluating the iterator in case the data container has been modified since the last access or by rewinding or fast forwarding if the external and the internal pointer of the iterator are mismatching. Once this method has returned, then the iterator may access the data item it is pointing at without any further processing to be done.

Note:

This method isn't supposed to be called by an application and ought to be a protected member. It has been made public though, since the container type (defined by _ti_container) needs access to this method, when the container instance is being destroyed. With C++11 it isn't possible to friend template parameters, which means, that the type, displayed by the template parameter _ti_container, can only access public members.

get_pos ()

`sizetype_t get_pos ()`

Description:

This method retrieves the external pointer of the iterator.

Remarks:

Iterators have an external pointer member variable and its state can be read by using this method.

Return value:

The return value is the current state of the external pointer of the iterator.

Note:

This method isn't supposed to be called by an application and ought to be a protected member. It has been made public though, since the container type (defined by _ti_container) needs access to this method, when the container instance is being destroyed. With C++11 it isn't possible to friend template parameters, which means, that the type, displayed by the template parameter _ti_container, can only access public members.

get_iterator_state ()

```
void * get_iterator_state ()
```

Description:

This method returns a pointer to an iterator's external state.

Remarks:

To retrieve the external state of an iterator for a container to be used, this method has to be called.

Return value:

The return value is a pointer to the external state of this iterator.

Note:

This method isn't supposed to be called by an application and ought to be a protected member. It has been made public though, since the container type (defined by _ti_container) needs access to this method, when the container instance is being destroyed. With C++11 it isn't possible to friend template parameters, which means, that the type, displayed by the template parameter _ti_container, can only access public members.

set_time_stamp ()

```
void set_time_stamp (_ti_container *pContainer)
```

Description:

This method sets the time stamp of the iterator.

Input parameter:

pContainer	- specifies pointer to the container of which the time stamp will be copied across

Remarks:

This method copies the time stamp of the specified data container to the time stamp of the iterator. If the specified data container is not the same as the data container the iterator is associated with, then an ::std::runtime_error exception is thrown. In case the iterator is not associated with a data container and the specified container is set to NULL, then the resulting behaviour is undefined.

Note:

This method isn't supposed to be called by an application and ought to be a protected member. It has been made public though, since the container type (defined by _ti_container) needs access to this method, when the container instance is being destroyed. With C++11 it isn't possible to friend template parameters, which means, that the type, displayed by the template parameter _ti_container, can only access public members.

Friend Operators

```
1)
template <class _ti_container>
friend typename ::CBTreeConstIterator<_ti_container> operator+ (
          typename _ti_container::sizetype_t nLhs,
          const typename ::CBTreeConstIterator<_ti_container> &nRhs)
2)
template <class _ti_container>
friend typename ::CBTreeConstIterator<_ti_container> operator+ (
          int nLhs,
          const typename ::CBTreeConstIterator<_ti_container> &nRhs)
```

Description:

These operators combine a value and an iterator to calculate the location to be pointed at by a new iterator.

Input parameters:

nLhs	- specifies left hand side operand to be combined
nRhs	- specifies right hand side operand to be combined

Remarks:

These operators are the commutative versions of the arithmetic binary addition operators (Binary Operators) allowing to offset an iterator by an integer or size_type type.

Operator (2), which is allowing for negative inputs, is able to invert its operation in case the input is indeed negative. This means that, if an iterator is incremented by a negative number, then the result is equivalent to said iterator being decremented by that number's absolute value. In case the magnitude of nLhs is deemed to be zero, then calling these operators is ineffective and only a copy of nRhs is returned. Since these operators are not accessing the container associated with the target iterator, the resulting iterator may exceed the container's range, without an exception being thrown.

Return value:

The return value is an iterator, which is associated with the same container as nRhs and displays the result of the operation via its external pointer, hence it is not evaluated once those operators have returned.

CBTreeIterator

The iterator type C. inherits from CBTreeConstIterator, which in addition to the type it is originating from, is capable to provide read / write access. Said access is possible via a class called CBTreeIteratorSubScriptWrapper, which is returned by one of the Indirection Operators and the Sub-Script Operators of this iterator type.

The iterator type C. also has only one template parameter called _ti_container, which is the type of data container to be used when performing operations. Since this iterator type is derived from CBTreeConstIterator type, the container type set by _ti_container has to provide the same Type Definitions for this iterator type to work. Also any member that has been defined for CBTreeConstIterator is present in this iterator type and won't be explained here again. If a description of those methods and operators are required, then please find the appropriate section above.

Type Definitions

All iterators only have one template parameter, which is the type of container the iterator is going to be used for. This template parameter is called _ti_container and has to provide a number of definitions as shown in the table below:

Type	Definition	Description
container_t	_ti_container	This definition is the data container type that will be using the iterator type in question.
value_type	typename container_t::value_type	This type defines what a data entry is and what is displayed to the application by the container when accessing it.
size_type	typename container_t::size_type	This defines what type is employed to address individual data entries when accessing a container as a linear array container type.
difference_type	typename container_t::difference_type	This defines what types is used to display the distance between two data entries is.
reference	typename container_t::reference	This type defines what a read / write reference of a data entry is.
pointer	typename container_t::pointer	This type defines what a read / write pointer of a data entry is.

Constructor

1)
```
CBTreeIterator<_ti_container> ()
```
2)
```
CBTreeIterator<_ti_container> (const _ti_container *pContainer, const sizetype_t nPos,
const bool bRegister = true)
```
3)
```
CBTreeIterator<_ti_container> (const _ti_container *pContainer, const size_type nPos,
const void *pExternState, const btree_time_stamp_t sTimeStamp, const bool bRegister = true)
```

4)
```
CBTreeIterator<_ti_container> (const CBTreeIterator &rIter, const bool bRegister = true)
```
5)
```
CBTreeIterator<_ti_container> (CBTreeIterator && rIter)
```
6)
```
CBTreeIterator<_ti_container> (const CBTreeReverseIterator<CBTreeIterator<_ti_container> >
&rRIter, bool bRegister = true)
```

Description:

These constructors initialise a b-tree framework read-write iterator.

Input parameters:

pContainer	- specifies pointer to container this iterator will be associated with
rIter	- references iterator instance to take a copy of
rRIter	- references reverse iterator instance to take a copy of
nPos	- specifies absolute linear position this iterator is to be initialised with
bRegister	- specifies if this iterator will be registered with its associated container instance

Remarks:

These constructors initialise an iterator instance to be used with a b-tree instance. The only exception is the default constructor (1), which only sets the iterator to a disassociated state. So, this iterator has to be initialised, using its Assignment Operator, before any valid access can take place. Constructor (2) only sets up the external pointer to nPos and due to the lazy evaluation approach, the b-tree iterators exercise, the internal pointer is not evaluated on construction. This is useful if further arithmetic operations are to be performed with the newly created instance and an immediate access is not required.

To fully initialise an iterator, the next variant (3) has to be used. The external and internal pointer are set to nPos, while the nNode and nSub have to display the correct node / entry pair associated with the internal pointer. This way of constructing an iterator is helpful if the evaluation took place externally and an immediate access is due.

The copy constructor (4), does as the name already says. It takes a copy of the state of rIter, may that be evaluated or not, and initialises this iterator with the taken copy.

The move constructor (5), swaps the state of the newly construct but not evaluated iterator with the state the input iterator rIter. This allows the compiler to generate faster code in order to quickly move temporary instances.

The constructor (6) exist so that the underlying STL implementation of ::std::reverse_iterator can exploit the existing operators of the CBTreeIterator type in order to perform its operations.

Since an iterator can be associated with a container, it has to be ensured that a container exists, once an iterator is using said container for an access. An iterator itself cannot tell if the container it is associated with is still present. Hence, a container instance has to flag to the iterators, which are registered with it, that it is about to be destroyed. If the parameter bRegister is set to true, then the newly created iterator is registering itself with the container instance being pointed at by pContainer. Setting bRegister to false skips the registration and is useful if an iterator only exists temporarily.

Note:

For the registration to work probably the code must be compiled with the build flag BTREE_ITERATOR_REGISTRATION. Building the code without that flag is helpful to create faster code, since the registration can be time consuming in case a large number of iterators is registered with a few number of containers. However, the application or any class using the iterators has to make sure any iterator isn't accessing a de-initialised container instance.

Indirection Operators

1)
```
const value_type & operator* () const
```
2)
```
CBTreeIteratorSubScriptWrapper<_ti_container> operator* ()
```

Description:

These operators provide access to the data item the iterator is pointing at.

Remarks:

To access a data item within the container the iterator is associated with, the indirection operator has to be used. If the iterator is pointing at a location outside of its associated container, then an ::std::out_of_range exception is thrown.

If the iterator is not associated with a data container, then an ::std::runtime_error exception is thrown.

Return value:

In case operator (1) is used, the return value provides read-only access. As oppose to operator (2), which is provides read / write access by returning an object of type CBTreeIteratorSubScriptWrapper. This object will provide read access via its overloaded assignment operator and write access via its overloaded cast operator.

Assignment Operator

```
CBTreeIterator & operator= (const CBTreeIterator &rIter)
```

Description:

This operator copies the state of a source iterator to the target instance.

Input parameter:

rIter	- reference to source iterator instance

Remarks:

This operator copies the state of a source iterator to the target instance. If the source iterator rIter is associated with a different b-tree container than the target iterator, then the assignment operator unregisters the target iterator from its associated b-tree container. In any event, the operator then copies the state of the source iterator across. If the target iterator was previously unregistered, then it is registered with the new b-tree instance.

Return value:

The return value is a reference to the updated iterator.

Move Assignment Operator

```
CBTreeIterator& operator= (CBTreeIterator &&rIter)
```

Description:

This operator swaps the state of a source iterator with the state of this instance.

Input parameters:

rIter	- reference to source iterator instance

Remarks:

In order to move a temporary iterator instance, this operator can be exploited by the compiler. If the source iterator rIter is associated with a different b-tree container than the target iterator, then the assignment operator unregisters the target iterator from its associated b-tree container. In any event, the operator then swaps the state of the external iterator with the state of this iterator instance. If the target iterator was previously unregistered, then it is registered with the new b-tree instance.

Return value:

The return value is a reference to the updated target iterator.

Sub-Script Operators

```
1)
CBTreeIteratorSubScriptWrapper<_ti_container> operator[] (const int nPos)
2)
CBTreeIteratorSubScriptWrapper<_ti_container> operator[] (const sizetype_t nPos)
```

Description:

These operators provide access to a data item relative to the position the iterator is pointing at.

Input parameters:

nPos	- specifies relative position the iterator is currently pointing at

Remarks:

To access a data item within the container the iterator is associated with, the sub-script operators may be used. The current location the iterator is pointing is combined with the input parameter nPos to determine what data item will be accessed. If the result is pointing to a location outside of its associated container, then an ::std::out_of_range exception is thrown. If the location the iterator is pointing at, is outside of its associated container, but the combination with nPos results in a location not exceeding said container again, then the access is valid and an exception is not thrown. The return value is an object of type CBTreeIteratorSubScriptWrapper, which provides read / write access.

The input parameter nPos of operator (1) may be negative and due to the subtraction of nPos from the iterator's current position, results in a location prior to what the iterator is pointing at.

Return value:

The return value is an object of type CBTreeIteratorSubScriptWrapper, which provides read / write access. Said object allows for read access via its overloaded assignment operator and write access via its overloaded cast operator.

Friend Operators

```
1)
template <class _ti_container>
friend typename ::CBTreeIterator<_ti_container> operator+ (
          typename _ti_container::sizetype_t nLhs,
          const typename ::CBTreeIterator<_ti_container> &nRhs)
2)
template <class _ti_container>
friend typename ::CBTreeIterator<_ti_container> operator+ (
          int nLhs,
          const typename ::CBTreeIterator<_ti_container> &nRhs)
```

Description:

These operators combine a value and an iterator to calculate the location to be pointed at by a new iterator.

Input parameters:

nLhs	- specifies left hand side operand to be combined
nRhs	- specifies right hand side operand to be combined

Remarks:

These operators are the commutative versions of the arithmetic binary addition operators, inherited from CBTreeConstIterator, allowing to offset an iterator by an integer or size_type type.

Operator (2), which is allowing for negative inputs, is able to invert its operation in case the input is indeed negative. This means that, if an iterator is incremented by a negative number, then the result is equivalent to said iterator being decremented by that number's absolute value. In case the magnitude of nLhs is deemed to be zero, then calling these operators is ineffective and only a copy of nRhs is returned. Since these operators are not accessing the container associated with the target iterator, the resulting iterator may exceed the container's range, without an exception being thrown.

Return value:

The return value is an iterator, which is associated with the same container as nRhs and displays the result of the operation via its external pointer, hence it is not evaluated once those operators have returned.

CBTreeConstReverseIterator

This iterator type operates in reverse order when addressing individual data entries. This means that, when addressing the initial, second, third etc. entry it respectively gets mapped to the final, second last, third last etc. data item. Reverse iterators are useful in situations were parts of a contain need to accessed in reverse order, while keeping the iterator type abstract.

The iterator type C. inherits from ::std::reverse_iterator with its template parameter set to CBTreeConstIterator and as a result only provides read-only access.

Type Definitions

All iterators only have one template parameter, which is the type of container the iterator is going to be used for. This template parameter is called _ti_container and has to provide a number of definitions as shown in the table below:

Type	Definition	Description
container_t	typename _t_iterator::container_t	This definition is the data container type that will be using the iterator type in question.
value_type	typename container_t::value_type	This type defines what a data entry is and what is displayed to the application by the container when accessing it.
size_type	typename container_t::size_type	This defines what type is employed to address individual data entries when accessing a container as a linear array container type.
difference_type	typename container_t::difference_type	This defines what types is used to display the distance between two data entries is.
reference	typename container_t::reference	This type defines what a read / write reference of a data entry is.
pointer	typename container_t::pointer	This type defines what a read / write pointer of a data entry is.

Constructor

```
1)
CBTreeConstReverseIterator<_t_iterator> ()
2)
CBTreeConstReverseIterator<_t_iterator> (const _t_iterator &rIter)
3)
CBTreeConstReverseIterator<_t_iterator> (const CBTreeConstReverseIterator<_t_iterator>
&rIter)
4)
CBTreeConstReverseIterator<_t_iterator> (CBTreeConstReverseIterator<_t_iterator> && rIter)
5)
CBTreeConstReverseIterator<_t_iterator> (
const CBTreeReverseIterator<CBTreeIterator<container_t> > &rIter)
```

Description:

These constructors initialise a b-tree framework read-only reverse-iterator.

Input parameter:

rIter	- references iterator instance to take a copy of

Remark:

These constructors initialise a read-only reverse-iterator instance to be used with a b-tree framework container instance. The only exception is the default constructor (1), which only sets the iterator to a disassociated state. So, this iterator has to be initialised, using its Assignment Operator, before any valid access can take place. Constructor (2) uses a reference of type template parameter _t_iterator, which this iterator type is based on, to initialise the current instance. Usually _t_iterator is set to CBTreeConstIterator, which means that, this iterator is initialised via the copy constructor of this iterator's base type.

The copy constructors (3, 5), do as the name already says. They take a copy of the state of rIter, may that be evaluated or not, and initialises this iterator with the taken copy.

The move constructor (5), swaps the state of the newly construct but not evaluated iterator with the state the input iterator rIter. This allows the compiler to generate faster code in order to quickly move temporary instances.

swap ()

```
void swap (CBTreeConstReverseIterator &rRight)
```

Description:

This method swaps the state of two iterators.

Input parameter:

rRight	- references iterator which state is to be swapped with the target iterator's state

Remarks:

This method swaps the state of two reverse iterator instances. If any of the involved iterators is registered with a container instance, then those registrations are swapped too, unless both iterators are registered with the same container instance. The call of this method is ineffective, in case rRight is the same instance as “*this”.

Assignment Operator

```
1)
CBTreeConstReverseIterator<_t_iterator> & operator= (
const reverse_iterator<_t_iterator> &rIter)
2)
CBTreeConstReverseIterator<_t_iterator> & operator= (
const CBTreeConstReverseIterator<_t_iterator> &rIter)
3)
CBTreeConstReverseIterator<_t_iterator> & operator= (
const CBTreeReverseIterator<CBTreeIterator <container_t> > &rIter)
```

Description:

These operators copy the state of a source iterator instance to this instance.

Input parameters:

rIter	- reference to source iterator instance

Remarks:

To take a copy of a reverse iterator instance, one of these operators has to be used. If the source iterator rIter is associated with a different container than the target iterator, then the assignment operator unregisters the target iterator from its associated container. In any event, the operator then copies the state of the source iterator across. If the target iterator was previously unregistered, then it is registered with the new container instance.

Return value:

The return value is a reference to the updated target iterator.

Move Assignment Operator

```
CBTreeConstReverseIterator& operator= (CBTreeConstReverseIterator &&rIter)
```

Description:

This operator swaps the state of a source iterator with the state of this instance.

Input parameters:

rIter	- reference to source iterator instance

Remarks:

In order to move a temporary iterator instance, this operator can be exploited by the compiler. If the source iterator rIter is associated with a different b-tree container than the target iterator, then the assignment operator unregisters the target iterator from its associated b-tree container. In any event, the operator then swaps the state of the external iterator with the state of this iterator instance. If the target iterator was previously unregistered, then it is registered with the new b-tree instance.

Return value:

The return value is a reference to the updated target iterator.

Comparison Operators

1)
```
bool operator== (const CBTreeConstReverseIterator &rIter)
```
2)
```
bool operator!= (const CBTreeConstReverseIterator &rIter)
```

Description:

These operators return if two reverse-iterators display the same offset.

Input parameters:

rIter	- reference to external iterator instance

Remarks:

These operators compare the offset of this and an external iterator and return whether they are equal or not. It doesn't matter if those iterators are associated with different container instances. However, if one or both iterators are not associated with a container (i.e. m_pContainer is set to NULL), then the result is always deemed as not equal. If both this and the external iterator are the same instance, then the result is always displayed as equal, even if said instance is not fully initialised.

Return value:

As for operator (1), if the result is deemed to be equal, then true is returned, otherwise false. Operator (2) returns the exact opposite of what operator (1) returns.

Indirection Operator

```
value_type & operator* ()
```

Description:

This operator provides pseudo read-only access to the data item the iterator is pointing at.

Remarks:

To access a data item within the container the iterator is associated with, the indirection operator has to be used. If the iterator is pointing to a location outside of its associated container, then an

::std::out_of_range exception is thrown. The returned reference is non-const, but if it is being written to, then the write is ineffective and only a buffer being part of the iterator is modified. This modification is discarded once the next access takes place, even if the iterator has not been moved in the meantime.

If the iterator is not associated with a data container, then an ::std::runtime_error exception is thrown.

Return value:

The return value is a reference to the data element the iterator is pointing at.

De-reference Operator

```
value_type * operator-> ()
```

Description:

This operator provides a pointer to the data element the iterator is pointing at.

Remarks:

The de-reference operator provides a pointer the data element the iterator is pointing at. Since this a const iterator providing read-only access, any writes would modify an internal copy of the data element in question, as oppose to altering the actual data in the b-tree structure. If that copy is modified, then this modification is discarded once the next access takes place, even if the iterator has not been moved in the meantime. In case the iterator is pointing to a location that exceeds the container, then an ::std::out_of_range exception is thrown.

Return value:

The return value is a pointer to the data element the iterator is pointing at.

Sub-Script Operator

1)
value_type & operator[] (const int nPos)

2)
value_type & operator[] (const sizetype_t nPos)

Description:

These operators provide pseudo read-only access to a data item relative to the position the iterator is pointing at.

Input parameters:

nPos	- specifies relative position the iterator is currently pointing at

Remarks:

To access a data item within the container the iterator is associated with, the sub-script operators may be used. The current location the iterator is pointing at is combined with the input parameter nPos to determine what data item will be accessed. If the result is pointing to a location outside of its associated container, then an ::std::out_of_range exception is thrown. If the location the iterator is pointing at, is outside of its associated container, but in combination with nPos results in a valid location not exceeding said container again, then the access is valid and an exception is not thrown. The returned reference is non-const, but if it is being written to, then the write is ineffective and only a buffer being part of the iterator is modified. This modification is discarded once the next access takes place, even if the iterator is not moved in the meantime and nPos remains the same.

The input parameter nPos of operator (1) may be negative and due to the subtraction of nPos from the iterator's current position, results in a location prior to what the iterator is pointing at.

Return value:

The return value is a reference to the data element the iterator is pointing at and the input parameter nPos combined.

CBTreeReverseIterator

The iterator type C. inherits from CBTreeConstReverseIterator, which in addition to the type it is originating from, is capable to provide read / write access. Said access is possible via a class called CBTreeIteratorSubScriptWrapper, which is returned by one of the Indirection Operators and the Sub-Script Operators of this iterator type.

The iterator type C. also has only one template parameter called _t_iterator, which is the type of base iterator to be used when performing operations. This base iterator type is used as a template parameter to complete CBTreeConstReverseIterator this class is inheriting from and in general is the template parameter _ti_container of CBTreeConstReverseIterator is set to CBTreeIterator instead of CBTreeConstIterator.

Type Definitions

All iterators only have one template parameter, which is the type of container the iterator is going to be used for. This template parameter is called _ti_container and has to provide a number of definitions as shown in the table below:

Type	**Definition**	**Description**
container_t	typename _t_iterator::container_t	This definition is the data container type that will be using the iterator type in question.
value_type	typename container_t::value_type	This type defines what a data entry is and what is displayed to the application by the container when accessing it.
size_type	typename container_t::size_type	This defines what type is employed to address individual data entries when accessing a container as a linear array container type.
difference_type	typename container_t::difference_type	This defines what types is used to display the distance between two data entries is.
reference	typename container_t::reference	This type defines what a read / write reference of a data entry is.
pointer	typename container_t::pointer	This type defines what a read / write pointer of a data entry is.

Constructor

1)
```
CBTreeReverseIterator<_t_iterator> ()
```
2)
```
CBTreeReverseIterator<_t_iterator> (const _t_iterator &rIter)
```
3)
```
CBTreeReverseIterator<_t_iterator> (const CBTreeReverseIterator<_t_iterator> &rIter)
```
4)
```
CBTreeReverseIterator<_t_iterator> (CBTreeReverseIterator<_t_iterator> && rIter)
```

Description:

These constructors initialise a b-tree framework read-write reverse-iterator.

Input parameter:

rIter	- references iterator instance to take a copy off

Remark:

These constructors initialise an iterator instance to be used with a b-tree instance. The only exception is the default constructor (1), which only sets the iterator to a disassociated state. So, this iterator has to be initialised, using its Assignment Operator, before any valid access can take place. Constructor (2) uses a reference of type template parameter _t_iterator, which this iterator type is based on, to initialise the current instance. Usually _t_iterator is set to CBTreeIterator, which means that, this iterator is initialised via the copy constructor of this iterator's base type.

The copy constructor (3), does as the name already says. It takes a copy of the state of rIter, may that be evaluated or not, and initialises this iterator with the taken copy.

The move constructor (4), swaps the state of the newly construct but not evaluated iterator with the state the input iterator rIter. This allows the compiler to generate faster code in order to quickly move temporary instances.

Indirection Operators

1)
```
const value_type & operator* () const
```
2)
```
CBTreeIteratorSubScriptWrapper<_ti_container> operator* ()
```

Description:

These operators provide access to the data item the iterator is pointing at.

Remarks:

To access a data item within the container the iterator is associated with, the indirection operator has to be used. If the iterator is pointing to a location outside of its associated container, then an ::std::out_of_range exception is thrown.

If the iterator is not associated with a data container, then an ::std::runtime_error exception is thrown.

Return value:

In case operator (1) is used, the return value provides read-only access. As oppose to operator (2), which is provides read / write access by returning an object of type CBTreeIteratorSubScriptWrapper. The afore mentioned object will provide read access via its overloaded assignment operator and write access via its overloaded cast operator.

Assignment Operator

```
1)
CBTreeReverseIterator & operator= (const ::std::reverse_iterator<_t_iterator> &rIter)
2)
CBTreeReverseIterator & operator= (const CBTreeReverseIterator &rIter)
3)
CBTreeReverseIterator & operator=    (const value_type &rData)
```

Description:

Operators (1, 2) copy the state of a source iterator to the target instance. Operator (3) assigns a new value to the data entry it is referring to.

Input parameter:

rIter	- reference to source iterator instance
rData	- reference of value to be assigned

Remarks:

Operators (1, 2) copy the state of a source iterator to the target instance. If the source iterator rIter is associated with a different b-tree container than the target iterator, then the assignment operator unregisters the target iterator from its associated container. In any event, the operator then copies the state of the source iterator across. If the target iterator was previously unregistered, then it is registered with the new container instance.

Operator (3) copies the input value rData, to the location the iterator is referring to. If the iterator cannot resolve the access, since either the iterator is not associated with a container or the iterator refers to a location outside of the container, then an ::std::run_time_error exception is thrown.

Return value:

The return value is a reference to the updated iterator.

Move Assignment Operator

```
CBTreeReverseIterator& operator= (CBTreeReverseIterator &&rIter)
```

Description:

This operator swaps the state of a source iterator with the state of this instance.

Input parameters:

rIter	- reference to source iterator instance

Remarks:

In order to move a temporary iterator instance, this operator can be exploited by the compiler. If the source iterator rIter is associated with a different b-tree container than the target iterator, then the assignment operator unregisters the target iterator from its associated b-tree container. In any event, the operator then swaps the state of the external iterator with the state of this iterator instance. If the target iterator was previously unregistered, then it is registered with the new b-tree instance.

Return value:

The return value is a reference to the updated target iterator.

Sub-Script Operators

1)
`CBTreeIteratorSubScriptWrapper<_ti_container> operator[] (const int nPos)`

2)
`CBTreeIteratorSubScriptWrapper<_ti_container> operator[] (const sizetype_t nPos)`

Description:

These operators provide access to a data item relative to the position the iterator is pointing at.

Input parameters:

nPos	- specifies relative position the iterator is currently pointing at

Remarks:

To access a data item within the container the iterator is associated with, the sub-script operators may be used. The current location the iterator is pointing is combined with the input parameter nPos to determine what data item will be accessed. If the result is pointing to a location outside of its associated container, then an ::std::out_of_range exception is thrown. If the location the iterator is pointing at, is outside of its associated container, but the combination with nPos results in a location not exceeding said container, then the access is valid and an exception is not thrown. The return value is an object of type CBTreeIteratorSubScriptWrapper, which provides read / write access.

The input parameter nPos of operator (1) may be negative and due to the subtraction of nPos from the iterator's current position, results in a location prior to what the iterator is pointing at.

Return value:

The return value is an object of type CBTreeIteratorSubScriptWrapper, which provides read / write access. Said object allows for read access via its overloaded assignment operator and write access via its overloaded cast operator.

CBTreeIteratorSubScriptWrapper

As it can be seen in the sections above, there are two types of iterators. One of those types is read-only, such as CBTreeConstIterator and CBTreeConstReverseIterator, and the other type is read-write capable, like CBTreeIterator and CBTreeReverseIterator. While the read-only types indirection operator returns a reference to an internal buffer, and thus making it impossible to manipulate the container's content, the read-write types need an arbiter to class, which is telling the iterator's container if a read or a write operation has to be performed. A reference cannot be returned in case a latter type is in use, since it is not known what data layer type is in use and a would-be returned reference could potentially point at a cache inside the data layer. Manipulations of said cache via a reference are hard to detect and more to the point, a cache can be overwritten at any point in time. To work around this problem the afore mentioned arbiter class is needed. The C. class acts as that arbiter and is retuned by the iterator types CBTreeIterator and CBTreeReverseIterator indirection and sub-script operators. C. has a cast operator which handles read operations and an assignment operator which handles writes.

Constructor

```
CBTreeIteratorSubScriptWrapper<_ti_container> (const CBTreeIterator<_ti_container> &rInstance,
sizetype_t nPos)
```

Description:

This constructor initialises an iterator read-write access wrapper instance.

Input parameter:

rInstance	- references to read-write capable iterator
nPos	- specifies the realtive location to be accessed

Remarks:

This constructor initialises an iterator read-write access wrapper instance, which usually is used as a nameless instance by the calling application. A copy of the iterator reference and the releative location parameter nPos are take for later use, when it is decided if a read or write operation has to take place.

Assignment Operator

```
CBTreeIteratorSubScriptWrapper<_ti_container> & operator= (const value_type &rData)
```

Description:

This operator handles writes to a read-write capable iterator.

Input parameter:

rData	- references a data item to be written

Remarks:

When a read-write iterator has been written to, then this operator of the returned access wrapper class instance handles it. The location the data will be written to, is the location of the read-write iterator rInstance combined with the relative location nPos. If the resulting location in exceeding the container size the iterator is associated with, then an ::std::out_of_range exception is thrown. In case the iterator is not associated with a container instance at all, then an ::std::runtime_error exception is thrown.

Return value:

The return value is a reference to the wrapper class instance.

Cast Operator

```
operator const value_type & ()
```

Description:

This operator handles reads to a read-write capable iterator.

Remarks:

When a read-write iterator is being read from, then this operator of the returned access wrapper class instance handles it. The location the data is returned from, is the location of the read-write iterator rInstance combined with the relative location nPos. If the resulting location in exceeding the container size the iterator is associated with, then an ::std::out_of_range exception is thrown. In case the iterator is not associated with a container instance at all, then an ::std::runtime_error exception is thrown.

Return value:

The return value is a reference to an internal buffer, which contains a copy of the data item in question.