# b-tree sequence

Generated by Doxygen 1.8.6

Mon Jul 7 2014 22:13:18

# Contents

# Chapter 1

# Class Index

## 1.1 Class List

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

# Chapter 2

# File Index

## 2.1 File List

Here is a list of all documented files with brief descriptions:

# Chapter 3

# Class Documentation

## 3.1 btree_seq< T, L, M, A > Class Template Reference

The fast sequence container, which behaves like std::vector takes O(log(N)) to insert/delete elements.

```
#include <btree_seq.h>
```

### Classes

- class iterator_base

  *Iterator template for const_iterator and iterator.*

### Public Types

- typedef A allocator_type

  *The last template parameter, allocator.*
- typedef A::value_type value_type

  *Value type, T (the first template parameter).*
- typedef A::reference reference

  *Reference type, T&.*
- typedef A::const_reference const_reference

  *Constant reference type, const T&.*
- typedef A::pointer pointer

  *Pointer type, T∗.*
- typedef A::const_pointer const_pointer

  *Constant pointer type, const T∗.*
- typedef A::size_type size_type

  *Unsigned integer type, size_t (unsigned int).*
- typedef A::difference_type difference_type

  *Signed integer type, ptr_diff_t (int).*
- typedef iterator_base< const T > const_iterator

  *Constant forward random-access iterator.*
- typedef iterator_base< T > iterator

  *Modifying forward random-access iterator.*
- typedef std::reverse_iterator
  < const_iterator > const_reverse_iterator

  *Constant reverse random-access iterator.*

- typedef std::reverse_iterator
  < iterator > reverse_iterator

  *Modifying reverse random-access iterator.*

## Public Member Functions

- btree_seq (const allocator_type &alloc=allocator_type())

  *Empty container constructor.*
- btree_seq (const btree_seq< T, L, M, A > &that)

  *Copy constructor.*
- btree_seq (size_type n, const value_type &val, const allocator_type &alloc=allocator_type())

  *Fill constructor.*
- template<typename Iterator >
  btree_seq (Iterator first, Iterator last, const allocator_type &alloc=allocator_type())

  *Range constructor.*
- ∼btree_seq ()

  *Destructor.*

### Iterators

- const_iterator begin () const

  *Return constant iterator to beginning.*
- const_iterator end () const

  *Return constant iterator to end.*
- const_iterator cbegin () const

  *Return constant iterator to beginning.*
- const_iterator cend () const

  *Return constant iterator to end.*
- iterator begin ()

  *Return iterator to beginning.*
- iterator end ()

  *Return constant iterator to end.*
- const_reverse_iterator rbegin () const

  *Return constant reverse iterator to reverse beginning.*
- const_reverse_iterator rend () const

  *Return constant reverse iterator to reverse end.*
- const_reverse_iterator crbegin () const

  *Return constant reverse iterator to reverse beginning.*
- const_reverse_iterator crend () const

  *Return constant reverse iterator to reverse end.*
- reverse_iterator rbegin ()

  *Return reverse iterator to reverse beginning.*
- reverse_iterator rend ()

  *Return reverse iterator to reverse end.*
- iterator iterator_at (size_type pos)

  *Return iterator to a given index.*
- const_iterator citerator_at (size_type pos) const

  *Return constant iterator to a given index.*

### Access

*Access of the contents*

- reference operator[] (size_type pos)

  *Access to element.*
- const_reference operator[] (size_type pos) const

  *Constant access to element.*

- size_type size () const

  *Number of elements in container.*
- reference at (size_type pos)

  *Access to element with range check.*
- const_reference at (size_type pos) const

  *Constant access to element with range check.*
- reference front ()

  *Returns a reference to the first element.*
- const_reference front () const

  *Returns a constnt reference to the first element.*
- reference back ()

  *Returns a reference to the last element.*
- const_reference back () const

  *Returns a constant reference to the last element.*
- bool empty () const

  *Returns true if the container contains no elements.*
- template< typename V >
  size_type visit (size_type start, size_type end, V &v)

  *Sequential search/modify operation on the range.*

### Modifying certain elements of the sequence

- void insert (size_type pos, const value_type &val)

  *Native function for inserting a single element.*
- template< class InputIterator >
  void insert (size_type pos, InputIterator first, InputIterator last)

  *Native function for inserting a range of elements.*
- iterator insert (const_iterator pos, const value_type &val)

  *Compatible function for inserting the single element.*
- iterator insert (const_iterator pos, size_type n, const value_type &val)

  *Compatible function for inserting n copies of an element.*
- template< class InputIterator >
  iterator insert (const_iterator pos, InputIterator first, InputIterator last)

  *Compatible function for inserting a range of elements.*
- void fill (size_type pos, size_type repetition, const value_type &val)

  *Native function for inserting n copies of an element.*
- void resize (size_type n, const value_type &val=value_type())

  *Resize container so that it contains n elements.*
- void erase (size_type first, size_type last)

  *Native function for erasing elements.*
- iterator erase (const_iterator pos)

  *Compatible function for erasing one element.*
- iterator erase (const_iterator first, const_iterator last)

  *Compatible function for erasing elements.*
- void push_back (const value_type &val)

  *Adds element to the end of the sequence.*
- void push_front (const value_type &val)

  *Adds element to the beginning of the sequence.*
- void pop_back ()

  *Removes the last element.*
- void pop_front ()

  *Removes the first element.*

### Modifying the whole contents of the sequence

- btree_seq & operator= (const btree_seq< T, L, M, A > &that)

  *Assign content.*
- void swap (btree_seq< T, L, M, A > &that)

*Swaps contents of two containers.*
- void clear ()

    *Erases all contents of the container.*
- void assign (size_type n, const value_type &val)

    *Replaces the whole contents with n copies of val.*
- template<class InputIterator >
  void assign (InputIterator first, InputIterator last)

    *Replaces the whole contents with a range.*
- void concatenate_right (btree_seq< T, L, M, A > &that)

    *Fast concatenate two sequences (that sequence to the right).*
- void concatenate_left (btree_seq< T, L, M, A > &that)

    *Fast concatenate two sequences (that sequence to the left).*
- void split_right (btree_seq< T, L, M, A > &that, size_type pos)

    *Fast split, leaving right piece in that container.*
- void split_left (btree_seq< T, L, M, A > &that, size_type pos)

    *Fast split, leaving left piece in that container.*

**Others**

*Functions not used in release version: debug, profile and compatibility.*

- void __check_consistency ()

    *Checks consistency of the container (debug).*
- template<class output_stream >
  void __output (output_stream &o, const char ∗comm="")

    *Output the contents of container into the stream (debug).*
- size_type __children_in_branch ()

    *Returns L (second parameter of the template) (profile, RTTI).*
- size_type __elements_in_leaf ()

    *Returns M (third parameter of the template) (profile, RTTI).*
- size_type __branch_size ()

    *Returns size of inner node of the tree (profile, RTTI).*
- size_type __leaf_size ()

    *Returns size of leaf node of the tree (profile, RTTI).*
- void reserve (size_type n)

    *Function does nothing (compatibility with std::vector).*
- allocator_type get_allocator () const

    *Returns allocator.*

### 3.1.1 Detailed Description

**template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>>class btree_seq< T, L, M, A >**

The fast sequence container, which behaves like std::vector takes O(log(N)) to insert/delete elements.

This container implements most of std::vector's members. It inserts/deletes elements much faster than any standart container. However, random access to the element takes O(log(N)) time as well. For sequential access to elements, iterators and `visit` exist, which require practically constant time per element. The implementation is based on btrees.

**Template Parameters**

| | |
|---:|---|
| *T* | the type of the element |
| *L* | maximal number of children per branch, default 30 minimum 4. You can change it for better performance. |

| | |
|---|---|
| *M* | maximal number of elements per leaf, default 60 minimum 4. You can change it for better performance. |
| *A* | allocator. |

## 3.1.2 Constructor & Destructor Documentation

### 3.1.2.1 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> btree_seq< T, L, M, A >::btree_seq ( const allocator_type & *alloc* = allocator_type() ) [inline], [explicit]

Empty container constructor.

Constructs an empty container with no elements. Complexity: constant.

**Parameters**

| | |
|---|---|
| *alloc* | allocator |

### 3.1.2.2 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> btree_seq< T, L, M, A >::btree_seq ( const btree_seq< T, L, M, A > & *that* ) [inline]

Copy constructor.

Copies all elements from another container. Complexity: O(N∗log(N)), N=that.size().

**Parameters**

| | |
|---|---|
| *that* | another container to be copied |

### 3.1.2.3 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> btree_seq< T, L, M, A >::btree_seq ( size_type *n*, const value_type & *val*, const allocator_type & *alloc* = allocator_type() ) [inline], [explicit]

Fill constructor.

Constructs a container with n elements, each of them is copy of val. Complexity: O(n∗log(n)).

**Parameters**

| | |
|---|---|
| *n* | number of elements |
| *val* | fill value |
| *alloc* | allocator |

### 3.1.2.4 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> template<typename Iterator > btree_seq< T, L, M, A >::btree_seq ( Iterator *first*, Iterator *last*, const allocator_type & *alloc* = allocator_type() ) [inline]

Range constructor.

Constructs a container filled with elements from range [first,last). Complexity: O(N∗log(N)), N=dist(last,first).

**Parameters**

| | |
|---|---|
| *first* | first position in a range |
| *last* | last position in a range |

| | |
|---|---|
| *alloc* | allocator |

#### 3.1.2.5 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> btree_seq< T, L, M, A >::~btree_seq ( ) [inline]

Destructor.

Deletes the contents and frees memory. Complexity: O(N∗log(N)).

### 3.1.3 Member Function Documentation

#### 3.1.3.1 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> void btree_seq< T, L, M, A >::__check_consistency ( )

Checks consistency of the container (debug).

Use this function when modifying this code or if you are unsure, if your program corrupts memory of the container.

#### 3.1.3.2 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> void btree_seq< T, L, M, A >::assign ( size_type *n*, const value_type & *val* )

Replaces the whole contents with n copies of val.

Erases contents of the container, then fills it with n copies of val. Complexity: O((n+M)∗log(n+M)), M - existing elements, n - new ones

**Parameters**

| | |
|---|---|
| *n* | new size of container |
| *val* | element to clone |

#### 3.1.3.3 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> template<class InputIterator > void btree_seq< T, L, M, A >::assign ( InputIterator *first*, InputIterator *last* ) [inline]

Replaces the whole contents with a range.

Erases contents of the container, then fills it with a copy of range [first,last). Complexity: O((n+M)∗log(n+M)), M - existing elements, n - new ones

**Parameters**

| | |
|---|---|
| *first* | first element to be inserted |
| *last* | element behind the last element to be inserted |

#### 3.1.3.4 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> reference btree_seq< T, L, M, A >::at ( size_type *pos* ) [inline]

Access to element with range check.

Returns a reference to the element at position pos. Complexity: O(log(N)).

**Parameters**

| | |
|---|---|
| *pos* | index of the element |

#### 3.1.3.5 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> const_reference btree_seq< T, L, M, A >::at ( size_type *pos* ) const [inline]

Constant access to element with range check.

Returns a constant reference to the element at position pos. Complexity: O(log(N)).

**Parameters**

| | |
|---|---|
| *pos* | index of the element |

#### 3.1.3.6 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> reference btree_seq< T, L, M, A >::back ( ) [inline]

Returns a reference to the last element.

Complexity: O(log(N)).

#### 3.1.3.7 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> const_reference btree_seq< T, L, M, A >::back ( ) const [inline]

Returns a constant reference to the last element.

Complexity: O(log(N)).

#### 3.1.3.8 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> const_iterator btree_seq< T, L, M, A >::begin ( ) const [inline]

Return constant iterator to beginning.

Complexity: constant.

#### 3.1.3.9 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> iterator btree_seq< T, L, M, A >::begin ( ) [inline]

Return iterator to beginning.

Complexity: constant.

#### 3.1.3.10 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> const_iterator btree_seq< T, L, M, A >::cbegin ( ) const [inline]

Return constant iterator to beginning.

Complexity: constant.

#### 3.1.3.11 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> const_iterator btree_seq< T, L, M, A >::cend ( ) const [inline]

Return constant iterator to end.

Complexity: constant.

#### 3.1.3.12 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> const_iterator btree_seq< T, L, M, A >::citerator_at ( size_type *pos* ) const [inline]

Return constant iterator to a given index.

Complexity: constant.

#### 3.1.3.13 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> void btree_seq< T, L, M, A >::clear ( ) [inline]

Erases all contents of the container.

Complexity: O(N∗log(N))

#### 3.1.3.14 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> void btree_seq< T, L, M, A >::concatenate_left ( btree_seq< T, L, M, A > & *that* )

Fast concatenate two sequences (that sequence to the left).

Concatenate two sequences (that sequence to the left), put result into this sequence and leave that sequence empty. Concatenation is done without copying all elements. Example: if sequence A contains {0,1,2} and sequeance B contains {3,4,5}, after a call 'A.concatenate_left(B)' A contains {3,4,5,0,1,2} and B is empty. Complexity: O(log(N+-M))

**Parameters**

| | |
|---|---|
| *that* | container to concatenate |

#### 3.1.3.15 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> void btree_seq< T, L, M, A >::concatenate_right ( btree_seq< T, L, M, A > & *that* )

Fast concatenate two sequences (that sequence to the right).

Concatenate two sequences (that sequence to the right), put result into this sequence and leave that sequence empty. Concatenation is done without copying all elements. Example: if sequence A contains {0,1,2} and sequeance B contains {3,4,5}, after a call 'A.concatenate_right(B)' A contains {0,1,2,3,4,5} and B is empty. Complexity: O(log(N+M))

**Parameters**

| | |
|---|---|
| *that* | container to concatenate |

#### 3.1.3.16 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> const_reverse_iterator btree_seq< T, L, M, A >::crbegin ( ) const [inline]

Return constant reverse iterator to reverse beginning.

Complexity: constant.

#### 3.1.3.17 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> const_reverse_iterator btree_seq< T, L, M, A >::crend ( ) const [inline]

Return constant reverse iterator to reverse end.

Complexity: constant.

#### 3.1.3.18 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> const_iterator btree_seq< T, L, M, A >::end ( ) const [inline]

Return constant iterator to end.

Complexity: constant.

#### 3.1.3.19 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> iterator btree_seq< T, L, M, A >::end ( ) [inline]

Return constant iterator to end.

Complexity: constant.

#### 3.1.3.20 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> void btree_seq< T, L, M, A >::erase ( size_type *first*, size_type *last* )

Native function for erasing elements.

Erase the range of [first,last) elements. Complexity: O(M∗log(N)), M-erased elements, N - all elements.

**Parameters**

| | |
|---:|---|
| *first* | index of the first element to erase |
| *last* | index of the last element to erase + 1 |

#### 3.1.3.21 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> iterator btree_seq< T, L, M, A >::erase ( const_iterator *pos* ) [inline]

Compatible function for erasing one element.

Erase the element at position pos. Complexity: O(log(N)). Hint: native 'erase(size_type, size_type)' might be faster.

**Parameters**

| | |
|---:|---|
| *pos* | index of the element to erase |

**Returns**

iterator pointing to the next position after erased element

#### 3.1.3.22 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> iterator btree_seq< T, L, M, A >::erase ( const_iterator *first*, const_iterator *last* ) [inline]

Compatible function for erasing elements.

Erase the range of [first,last) elements. Complexity: O(M∗log(N)), M-erased elements, N - all elements. Hint: native 'erase(size_type, size_type)' might be faster.

**Parameters**

| | |
|---:|---|
| *first* | index of the first element to erase |
| *last* | index of the last element to erase + 1 |

**Returns**

iterator pointing to the next position after erased elements

#### 3.1.3.23 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> void btree_seq< T, L, M, A >::fill ( size_type *pos*, size_type *repetition*, const value_type & *val* ) [inline]

Native function for inserting n copies of an element.

Inserts n copiesof an element at a given position. Complexity: O((n+M)∗log(n+M)), M - existing elements, n - new ones.

**Parameters**

| | |
|---:|---|
| *pos* | position to insert |
| *repetition* | number of copies to insert |
| *val* | value to insert |

#### 3.1.3.24 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> reference btree_seq< T, L, M, A >::front ( ) [inline]

Returns a reference to the first element.

Complexity: O(log(N)).

#### 3.1.3.25 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> const_reference btree_seq< T, L, M, A >::front ( ) const [inline]

Returns a constnt reference to the first element.

Complexity: O(log(N)).

#### 3.1.3.26 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> void btree_seq< T, L, M, A >::insert ( size_type *pos*, const value_type & *val* ) [inline]

Native function for inserting a single element.

Inserts the element at given position. Hint: inserting the range is faster then multiple insertions of one element. Complexity: O(log(N)).

**Parameters**

| | |
|---:|---|
| *pos* | position to insert |
| *val* | element to insert |

#### 3.1.3.27 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> template<class InputIterator > void btree_seq< T, L, M, A >::insert ( size_type *pos*, InputIterator *first*, InputIterator *last* )

Native function for inserting a range of elements.

Inserts the range [first,last) of elements into the given position. Complexity: O((N+M)∗log(N+M)), N-existing elements, M - new ones.

**Parameters**

| | |
|---:|---|
| *pos* | position to insert |
| *first* | first element to insert |
| *last* | element behind the last element to insert |

### 3.1.3.28 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> iterator btree_seq< T, L, M, A >::insert ( const_iterator *pos*, const value_type & *val* ) [inline]

Compatible function for inserting the single element.

Inserts a single element at a given position. Complexity: O((N+M)∗log(N+M)), N-existing elements, M - new ones. Hint: native 'insert(size_type, const value_type& t)' might be faster.

**Parameters**

| | |
|---|---|
| *pos* | position to insert |
| *val* | value to insert |

**Returns**

iterator pointing to the newly inserted element

### 3.1.3.29 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> iterator btree_seq< T, L, M, A >::insert ( const_iterator *pos*, size_type *n*, const value_type & *val* ) [inline]

Compatible function for inserting n copies of an element.

Inserts n copiesof an element at a given position. Complexity: O((n+M)∗log(n+M)), M - existing elements, n - new ones. Hint: native 'fill(size_type, size_type, const value_type& val)' might be faster.

**Parameters**

| | |
|---|---|
| *pos* | position to insert |
| *n* | number of copies to insert |
| *val* | value to insert |

**Returns**

iterator pointing to the first of newly inserted elements

### 3.1.3.30 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> template<class InputIterator > iterator btree_seq< T, L, M, A >::insert ( const_iterator *pos*, InputIterator *first*, InputIterator *last* ) [inline]

Compatible function for inserting a range of elements.

Inserts the range [first,last) of elements into the given position. Complexity: O((N+M)∗log(N+M)), N-existing elements, M - new ones. Hint: native 'insert(size_type, InputIterator, InputIterator)' might be faster.

**Parameters**

| | |
|---|---|
| *pos* | position to insert |
| *first* | first element to insert |
| *last* | element behind the last element to insert |

**Returns**

iterator pointing to the first of newly inserted elements

### 3.1.3.31 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> iterator btree_seq< T, L, M, A >::iterator_at ( size_type *pos* ) [inline]

Return iterator to a given index.

Complexity: constant.

### 3.1.3.32 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> btree_seq& btree_seq< T, L, M, A >::operator= ( const btree_seq< T, L, M, A > & *that* ) [inline]

Assign content.

Deletes old contents and replaces it with copy of contents of that. Complexity: O(N∗log(N))+O(M∗log(M)), N=this->size(), M=that.size().

**Parameters**

| | |
|---|---|
| *that* | container to be assigned |

### 3.1.3.33 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> reference btree_seq< T, L, M, A >::operator[] ( size_type *pos* ) [inline]

Access to element.

Returns a reference to the element at position pos. No range check is done. Complexity: O(log(N)).

**Parameters**

| | |
|---|---|
| *pos* | index of the element |

### 3.1.3.34 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> const_reference btree_seq< T, L, M, A >::operator[] ( size_type *pos* ) const [inline]

Constant access to element.

Returns a constant reference to the element at position pos. No range check is done. Complexity: O(log(N)).

**Parameters**

| | |
|---|---|
| *pos* | index of the element |

### 3.1.3.35 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> void btree_seq< T, L, M, A >::pop_back ( ) [inline]

Removes the last element.

Complexity: O(log(N))

### 3.1.3.36 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> void btree_seq< T, L, M, A >::pop_front ( ) [inline]

Removes the first element.

Complexity: O(log(N))

### 3.1.3.37 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> void btree_seq< T, L, M, A >::push_back ( const value_type & *val* ) [inline]

Adds element to the end of the sequence.

Complexity: O(log(N))

#### 3.1.3.38 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> void btree_seq< T, L, M, A >::push_front ( const value_type & *val* ) [inline]

Adds element to the beginning of the sequence.

Complexity: O(log(N))

#### 3.1.3.39 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> const_reverse_iterator btree_seq< T, L, M, A >::rbegin ( ) const [inline]

Return constant reverse iterator to reverse beginning.

Complexity: constant.

#### 3.1.3.40 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> reverse_iterator btree_seq< T, L, M, A >::rbegin ( ) [inline]

Return reverse iterator to reverse beginning.

Complexity: constant.

#### 3.1.3.41 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> const_reverse_iterator btree_seq< T, L, M, A >::rend ( ) const [inline]

Return constant reverse iterator to reverse end.

Complexity: constant.

#### 3.1.3.42 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> reverse_iterator btree_seq< T, L, M, A >::rend ( ) [inline]

Return reverse iterator to reverse end.

Complexity: constant.

#### 3.1.3.43 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> void btree_seq< T, L, M, A >::resize ( size_type *n,* const value_type & *val* = value_type() )

Resize container so that it contains n elements.

If n is greater than container size, copies of the val are added to the end. If n is less than container size, some elements at the end of container are deleted.

#### 3.1.3.44 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> void btree_seq< T, L, M, A >::split_left ( btree_seq< T, L, M, A > & *that,* size_type *pos* )

Fast split, leaving left piece in that container.

Split sequence into two parts: [pos,size) is left in this container, [0,pos) is moved to that container. That container is cleaned before operation. Example if A contained {0,1,2,3,4}, after A.split_left(B,3) A contains {3,4} and B contains {0,1,2}. The operation is done without moving all elements. Complexity: O(log(N)), if the second container is initially empty.

**Parameters**

| | |
|---|---|
| *that* | container for leftt part of split operation (old contents removed) |
| *pos* | place to split |

#### 3.1.3.45 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> void btree_seq< T, L, M, A >::split_right ( btree_seq< T, L, M, A > & *that*, size_type *pos* )

Fast split, leaving right piece in that container.

Split sequence into two parts: [0,pos) is left in this container, [pos,size) is moved to that container. That container is cleaned before operation. Example if A contained {0,1,2,3,4}, after A.split_right(B,3) A contains {0,1,2} and B contains {3,4}. The operation is done without moving all elements. Complexity: O(log(N)), if the second container is initially empty.

**Parameters**

| | |
|---|---|
| *that* | container for right part of split operation (old contents removed) |
| *pos* | place to split |

#### 3.1.3.46 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> void btree_seq< T, L, M, A >::swap ( btree_seq< T, L, M, A > & *that* )

Swaps contents of two containers.

Complexity: constant.

**Parameters**

| | |
|---|---|
| *that* | container to swap with |

#### 3.1.3.47 template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>> template<typename V > size_type btree_seq< T, L, M, A >::visit ( size_type *start*, size_type *end*, V & *v* )

Sequential search/modify operation on the range.

Implements visitor pattern. The function 'visit' calls v() on the elements in the given range sequentially, until the end of range is reached or v() returns true (whichever happens earlier). This function allows to implement patterns like find_if, for_each and so on, but it behaves significantly (roughly twice) faster than standard implementation and iterator access. Complexity: O(log(N)∗(end-start))

**Parameters**

| | |
|---|---|
| *start* | the first element on which visitor should be called |
| *end* | the element beyond the last element on which visitor should be called |
| *v* | the visitor class, which must have 'bool operator(element&)' |

**Returns**

the index of the first element when v() returned true, or end if v() never returned true

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

- btree_seq.h

## 3.2 btree_seq< T, L, M, A >::iterator_base< TT > Class Template Reference

Iterator template for const_iterator and iterator.

```
#include <btree_seq.h>
```

### Public Types

- typedef std::random_access_iterator_tag iterator_category

    *Random access iterator category.*
- typedef std::iterator < std::random_access_iterator_tag, TT >::value_type value_type

    *T or const T.*
- typedef std::iterator < std::random_access_iterator_tag, TT >::difference_type difference_type

    *ptrdiff_t (int).*
- typedef std::iterator < std::random_access_iterator_tag, TT >::reference reference

    *T& or const T&.*
- typedef std::iterator < std::random_access_iterator_tag, TT >::pointer pointer

    *T∗ or const T∗.*

### Public Member Functions

- iterator_base ()

    *Default constructor.*
- iterator_base (const btree_seq ∗t, size_type pos)

    *Pointing to specific place in the tree.*
- iterator_base (const iterator_base &that)

    *Copy constructor for the same type.*
- template<typename T2 >
  iterator_base (const iterator_base< T2 > &that)

    *Constructor for conversion from iterator to const_iterator.*
- iterator_base & operator= (const iterator_base &that)

    *Operator = for the same type.*
- template<typename T2 >
  iterator_base & operator= (const iterator_base< T2 > &that)

    *Conversion from iterator to const_iterator.*
- reference operator∗ () const

    *Dereferencing.*
- pointer operator-> () const

    *Dereferencing.*
- reference operator[ ] (difference_type n) const

    *Getting arbitary element.*
- template<typename T2 >
  bool operator== (const iterator_base< T2 > &that) const

    *Comparison.*
- template<typename T2 >
  bool operator!= (const iterator_base< T2 > &that) const

    *Comparison.*

- template<typename T2 >
  bool operator> (const iterator_base< T2 > &that) const

  *Comparison.*
- template<typename T2 >
  bool operator< (const iterator_base< T2 > &that) const

  *Comparison.*
- template<typename T2 >
  bool operator>= (const iterator_base< T2 > &that) const

  *Comparison.*
- template<typename T2 >
  bool operator<= (const iterator_base< T2 > &that) const

  *Comparison.*
- template<typename T2 >
  difference_type operator- (const iterator_base< T2 > &that) const

  *Comparison.*
- iterator_base & operator++ ()

  *Preincrement.*
- iterator_base & operator-- ()

  *Predecrement.*
- iterator_base operator++ (int)

  *Postincrement.*
- iterator_base operator-- (int)

  *Postdecrement.*
- iterator_base & operator+= (difference_type n)

  *Increase position by n.*
- iterator_base & operator-= (difference_type n)

  *Decrease position by n.*
- iterator_base operator+ (difference_type n) const

  *Increase position by n.*
- iterator_base operator- (difference_type n) const

  *Decrease position by n.*
- size_type get_position () const

  *Returns current position.*
- const btree_seq ∗ get_container () const

  *Returns current container.*
- pointer __get_null_pointer () const

  *Returns null pointer.*

### 3.2.1 Detailed Description

**template<typename T, int L = 30, int M = 60, typename A = std::allocator<T>>template<typename TT>class btree_seq< T, L, M, A >::iterator_base< TT >**

Iterator template for const_iterator and iterator.

This is a lazy implementation of iterator. In other words, it takes constant time to construct, increment, decrement, get_position and relocate operations. When being dereferenced, it checks its validity and does actual work if necessary. So dereference operations (operators '∗' '[]' '->') do most work and take from constant time to O(log(-N)) time, depending on operation mode. If iterator is used in sequential mode (solely incrementing/decrementing), dereference operations require practically constant time amortized. If iterator is used in random access mode, dereference operations take O(log(N)) time and can be as expensive as sequence->operator[]. Hint: using function 'visit' might be faster than iterator. If you are modifying only one container at a time, using 'visit' might be preferable.

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

- btree_seq.h

# Chapter 4

# File Documentation

## 4.1 btree_seq.h File Reference

```
#include <assert.h>
#include <iterator>
#include "btree_seq2.h"
```

### Classes

- class btree_seq< T, L, M, A >

  *The fast sequence container, which behaves like std::vector takes O(log(N)) to insert/delete elements.*
- class btree_seq< T, L, M, A >::iterator_base< TT >

  *Iterator template for const_iterator and iterator.*

### Functions

- template<typename T , int L, int M, typename A >
  void swap (btree_seq< T, L, M, A > &first, btree_seq< T, L, M, A > &second)

  *Swap contents of two containers.*
- template<typename T , int L, int M, typename A >
  bool operator< (const btree_seq< T, L, M, A > &x, const btree_seq< T, L, M, A > &y)

  *Lexicographical comparison.*
- template<typename T , int L, int M, typename A >
  bool operator> (const btree_seq< T, L, M, A > &x, const btree_seq< T, L, M, A > &y)

  *Lexicographical comparison.*
- template<typename T , int L, int M, typename A >
  bool operator<= (const btree_seq< T, L, M, A > &x, const btree_seq< T, L, M, A > &y)

  *Lexicographical comparison.*
- template<typename T , int L, int M, typename A >
  bool operator>= (const btree_seq< T, L, M, A > &x, const btree_seq< T, L, M, A > &y)

  *Lexicographical comparison.*
- template<typename T , int L, int M, typename A >
  bool operator== (const btree_seq< T, L, M, A > &x, const btree_seq< T, L, M, A > &y)

  *Equality of size and all elements.*
- template<typename T , int L, int M, typename A >
  bool operator!= (const btree_seq< T, L, M, A > &x, const btree_seq< T, L, M, A > &y)

  *Inequality of size or any elements.*

### 4.1.1 Detailed Description

Declaration of btree_seq container, sequence based on btree.

# Index