Min-Max Heap

Generated by Doxygen 1.8.9.1

Sun Dec 20 2015 21:45:55

Contents

1 Namespace Documentation

1.1 _mmheap Namespace Reference

Functions

- size_t parent (size_t i)
- size_t has_parent (size_t i)
- size_t left (size_t i)
- size_t right (size_t i)
- size_t gparent (size_t i)
- bool has_gparent (size_t i)
- bool child (size_t i, size_t c)
- uint64_t log_2 (uint64_t i)
- bool min_level (size_t i)
- template< typename DataType >
std::pair< bool, size_t > min_child (DataType ∗heap_array, size_t i, size_t right_index)
- template< typename DataType >
std::pair< bool, size_t > min_gchild (DataType ∗heap_array, size_t i, size_t right_index)
- template< typename DataType >
std::pair< bool, size_t > min_child_or_gchild (DataType ∗heap_array, size_t i, size_t right_index)
- template< typename DataType >
std::pair< bool, size_t > max_child (DataType ∗heap_array, size_t i, size_t right_index)
- template< typename DataType >
std::pair< bool, size_t > max_gchild (DataType ∗heap_array, size_t i, size_t right_index)
- template< typename DataType >
std::pair< bool, size_t > max_child_or_gchild (DataType ∗heap_array, size_t i, size_t right_index)
- template< typename DataType >
void sift_down_min (DataType ∗heap_array, size_t sift_index, size_t right_index)

- template<typename DataType >
  void sift_down_max (DataType ∗heap_array, size_t sift_index, size_t right_index)
- template<typename DataType >
  void sift_down (DataType ∗heap_array, size_t sift_index, size_t right_index)
- template<typename DataType >
  void bubble_up_min (DataType ∗heap_array, size_t bubble_index)
- template<typename DataType >
  void bubble_up_max (DataType ∗heap_array, size_t bubble_index)
- template<typename DataType >
  void bubble_up (DataType ∗heap_array, size_t bubble_index)

1.1.1 Detailed Description

The `_mmheap` namespace contains functions that are only intended for internal use by the "public-facing" functions in the `mmheap` namespace. None of the functions in `_mmheap::` should be necessary externally.

1.1.2 Function Documentation

1.1.2.1 template<typename DataType > void _mmheap::bubble_up (DataType ∗ *heap_array,* size_t *bubble_index*)

perform min-max heap bubble-up on an element (at `bubble_index`)

Parameters

heap_array	the heap
bubble_index	the index of the element that should be bubbled up

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Definition at line 419 of file mmheap.h.

1.1.2.2 template<typename DataType > void _mmheap::bubble_up_max (DataType ∗ *heap_array,* size_t *bubble_index*)

perform min-max heap bubble-up on an element (at `bubble_index`) that is on a max-level

Parameters

heap_array	the heap
bubble_index	the index of the element that should be bubbled up

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Definition at line 397 of file mmheap.h.

1.1.2.3 template<typename DataType > void _mmheap::bubble_up_min (DataType ∗ *heap_array,* size_t *bubble_index*)

perform min-max heap bubble-up on an element (at `bubble_index`) that is on a min-level

Parameters

heap_array	the heap
bubble_index	the index of the element that should be bubbled up

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Definition at line 375 of file mmheap.h.

1.1.2.4 bool _mmheap::child (size_t *i,* size_t *c*) `[inline]`

Definition at line 63 of file mmheap.h.

1.1.2.5 size_t _mmheap::gparent (size_t *i*) `[inline]`

Definition at line 61 of file mmheap.h.

1.1.2.6 bool _mmheap::has_gparent (size_t *i*) `[inline]`

Definition at line 62 of file mmheap.h.

1.1.2.7 size_t _mmheap::has_parent (size_t *i*) `[inline]`

Definition at line 58 of file mmheap.h.

1.1.2.8 size_t _mmheap::left (size_t *i*) `[inline]`

Definition at line 59 of file mmheap.h.

1.1.2.9 uint64_t _mmheap::log_2 (uint64_t *i*)

Definition at line 71 of file mmheap.h.

1.1.2.10 template<typename DataType > std::pair<bool, size_t> _mmheap::max_child (DataType ∗ *heap_array,* size_t *i,* size_t *right_index*)

get a pair considing of an indication of whether `i` has any children, and if so, the index of the child containing the maximum value.

Parameters

heap_array	the heap
i	the index (parent) for which to find the max-child
right-index	the index of the right-most element that is part of the heap

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Returns

> a pair where the first element is `true` if `i` has children (`false` otherwise), and the second element is the index of the child whose value is largest (only if the first element is `true`)

Definition at line 203 of file mmheap.h.

1.1.2.11 template<typename DataType > std::pair<bool, size_t> _mmheap::max_child_or_gchild (DataType ∗ *heap_array,* size_t *i,* size_t *right_index*)

get a pair considing of an indication of whether `i` has any children, and if so, the index of the child or grandchild containing the maximum value.

Parameters

heap_array	the heap
i	the index (parent) for which to find the max-(grand)child
right-index	the index of the right-most element that is part of the heap

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Returns

> a pair where the first element is `true` if `i` has children (`false` otherwise), and the second element is the index of the child or grandchild whose value is largest (only if the first element is `true`)

Definition at line 266 of file mmheap.h.

1.1.2.12 template<typename DataType > std::pair<bool, size_t> _mmheap::max_gchild (DataType ∗ *heap_array,* size_t *i,* size_t *right_index*)

get a pair considing of an indication of whether `i` has any grandchildren, and if so, the index of the grandchild containing the maximum value.

Parameters

heap_array	the heap
i	the index (parent) for which to find the max-grandchild
right-index	the index of the right-most element that is part of the heap

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Returns

> a pair where the first element is `true` if `i` has grandchildren (`false` otherwise), and the second element is the index of the grandchild whose value is largest (only if the first element is `true`)

Definition at line 230 of file mmheap.h.

1.1.2.13 template<typename DataType > std::pair<bool, size_t> _mmheap::min_child (DataType ∗ *heap_array,* size_t *i,* size_t *right_index*)

get a pair considing of an indication of whether `i` has any children, and if so, the index of the child containing the minimum value.

Parameters

heap_array	the heap
i	the index (parent) for which to find the min-child
right-index	the index of the right-most element that is part of the heap

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Returns

> a pair where the first element is `true` if `i` has children (`false` otherwise), and the second element is the index of the child whose value is smallest (only if the first element is `true`)

Definition at line 116 of file mmheap.h.

1.1.2.14 template<typename DataType > std::pair<bool, size_t> _mmheap::min_child_or_gchild (DataType ∗ *heap_array,* size_t *i,* size_t *right_index*)

get a pair considing of an indication of whether `i` has any children, and if so, the index of the child or grandchild containing the minimum value.

Parameters

heap_array	the heap
i	the index (parent) for which to find the min-(grand)child
right-index	the index of the right-most element that is part of the heap

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Returns

> a pair where the first element is `true` if `i` has children (`false` otherwise), and the second element is the index of the child or grandchild whose value is smallest (only if the first element is `true`)

Definition at line 179 of file mmheap.h.

1.1.2.15 template<typename DataType > std::pair<bool, size_t> _mmheap::min_gchild (DataType ∗ *heap_array,* size_t *i,* size_t *right_index*)

get a pair considing of an indication of whether `i` has any grandchildren, and if so, the index of the grandchild containing the minimum value.

Parameters

heap_array	the heap
i	the index (parent) for which to find the min-grandchild
right-index	the index of the right-most element that is part of the heap

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Returns

a pair where the first element is `true` if `i` has grandchildren (`false` otherwise), and the second element is the index of the grandchild whose value is smallest (only if the first element is `true`)

Definition at line 143 of file mmheap.h.

1.1.2.16 bool _mmheap::min_level (size_t *i*) `[inline]`

returns `true` if `i` is on a Min-Level

Parameters

i	index into the heap

Returns

`true` if `i` is on a min-level

Definition at line 97 of file mmheap.h.

1.1.2.17 size_t _mmheap::parent (size_t *i*) `[inline]`

Definition at line 57 of file mmheap.h.

1.1.2.18 size_t _mmheap::right (size_t *i*) `[inline]`

Definition at line 60 of file mmheap.h.

1.1.2.19 template<typename DataType > void _mmheap::sift_down (DataType * *heap_array,* size_t *sift_index,* size_t *right_index*)

perform min-max heap sift-down on an element (at `sift_index`)

Parameters

heap_array	the heap
sift_index	the index of the element that should be sifted down
right_index	the index of the right-most element that is part of the heap

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Definition at line 356 of file mmheap.h.

1.1.2.20 template<typename DataType > void _mmheap::sift_down_max (DataType * *heap_array,* size_t *sift_index,* size_t *right_index*)

perform min-max heap sift-down on an element (at `sift_index`) that is on a max-level

Parameters

heap_array	the heap
sift_index	the index of the element that should be sifted down
right_index	the index of the right-most element that is part of the heap

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Definition at line 321 of file mmheap.h.

1.1.2.21 template<typename DataType > void _mmheap::sift_down_min (DataType ∗ *heap_array,* size_t *sift_index,* size_t *right_index*)

perform min-max heap sift-down on an element (at `sift_index`) that is on a min-level

Parameters

heap_array	the heap
sift_index	the index of the element that should be sifted down
right_index	the index of the right-most element that is part of the heap

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Definition at line 286 of file mmheap.h.

1.2 mmheap Namespace Reference

Functions

- template<typename DataType >
  void make_heap (DataType ∗heap_array, size_t size)

  make an arbitrary array into a heap (in-place)
- template<typename DataType >
  void heap_insert (const DataType &value, DataType ∗heap_array, size_t &count, size_t max_size)
- template<typename DataType >
  DataType heap_max (DataType ∗heap_array, size_t count)
- template<typename DataType >
  DataType heap_min (DataType ∗heap_array, size_t count)
- template<typename DataType >
  std::pair< bool, DataType > heap_insert_circular (const DataType &value, DataType ∗heap_array, size_t &count, size_t max_size)

  add to heap, rotating the maximum value out if the heap is full
- template<typename DataType >
  DataType heap_replace_at_index (const DataType &new_value, size_t index, DataType ∗heap_array, size_t count)
- template<typename DataType >
  DataType heap_remove_at_index (size_t index, DataType ∗heap_array, size_t &count)
- template<typename DataType >
  DataType heap_remove_min (DataType ∗heap_array, size_t &count)

- template<typename DataType >
  DataType heap_remove_max (DataType ∗heap_array, size_t &count)

1.2.1 Detailed Description

The `mmheap` namespace defines functions that are useful for building and maintaining a Min-Max heap. All necessary ("public-facing") functionality is in this namespace.

1.2.2 Function Documentation

1.2.2.1 template<typename DataType > void mmheap::heap_insert (const DataType & *value,* DataType ∗ *heap_array,* size_t & *count,* size_t *max_size*)

insert a new value to the heap (and update the `count`)

Parameters

	value	the new value to insert
	heap_array	the heap
`in,out`	*count*	the current number of items in the heap (will update)
	max_size	the physical storage allocation size of the heap

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Exceptions

std::runtime_error	if the heap is full prior to the insert operation

Definition at line 482 of file mmheap.h.

1.2.2.2 template<typename DataType > std::pair<bool, DataType> mmheap::heap_insert_circular (const DataType & *value,* DataType ∗ *heap_array,* size_t & *count,* size_t *max_size*)

add to heap, rotating the maximum value out if the heap is full

Add to the min-max heap in such a way that the maximum value is removed at the same time if the heap has reached its storage capacity.

Parameters

	value	new value to add
	heap_array	the heap
`in,out`	*count*	number of values currently in the heap (will update)
	max_size	maximum physical size allocated for the heap

Template Parameters

DataType	the type of data stored in the heap - must be DefaultConstructable, LessThan↩Comparable, Swappable, CopyConstructable, and CopyAssignable

Returns

> a pair consising of a flag and a value; the first element is a flag indicating that overflow occurred, and the second element is the value that rotated out of the heap (formerly the maximum) when the new value was added (set only if an overflow occurred)

Definition at line 549 of file mmheap.h.

1.2.2.3 template<typename DataType > DataType mmheap::heap_max (DataType ∗ *heap_array,* size_t *count*)

get the maximum value in the heap

Parameters

heap_array	the heap
count	the current number of values contained in the heap

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Returns

> the maximum value in the heap

Exceptions

std::runtime_error	if the heap is empty

Definition at line 504 of file mmheap.h.

1.2.2.4 template<typename DataType > DataType mmheap::heap_min (DataType ∗ *heap_array,* size_t *count*)

get the minimum value in the heap

Parameters

heap_array	the heap
count	the current number of values contained in the heap

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Returns

> the minimum value in the heap

Exceptions

std::runtime_error	if the heap is empty

Definition at line 524 of file mmheap.h.

1.2.2.5 template<typename DataType > DataType mmheap::heap_remove_at_index (size_t *index,* DataType ∗ *heap_array,* size_t & *count*)

remove and return value at a given index

Parameters

	index	index to remove
	heap_array	the heap
`in,out`	*count*	current number of values in the heap (will update)

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Returns

the value being removed

Exceptions

std::runtime_error	if the heap is empty
std::range_error	if the index is out of range

Definition at line 638 of file mmheap.h.

1.2.2.6 template<typename DataType > DataType mmheap::heap_remove_max (DataType ∗ *heap_array,* size_t & *count*)

remove and return the maximum value in the heap

Parameters

heap_array	the array
count	the current number of values in the heap (will update)

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Returns

the maximum value in the heap

Exceptions

std::runtime_error	if the heap is empty

Definition at line 687 of file mmheap.h.

1.2.2.7 template<typename DataType > DataType mmheap::heap_remove_min (DataType ∗ *heap_array,* size_t & *count*)

remove and return the minimum value in the heap

Parameters

heap_array	the array
count	the current number of values in the heap (will update)

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Returns

the minimum value in the heap

Exceptions

std::runtime_error	if the heap is empty

Definition at line 662 of file mmheap.h.

1.2.2.8 template<typename DataType > DataType mmheap::heap_replace_at_index (const DataType & *new_value,* size_t *index,* DataType ∗ *heap_array,* size_t *count*)

replace and return the value at a given index with a new value

Parameters

new_value	new value to insert
index	index of the value to replace
heap_array	the heap
count	number of values currently stored in the heap

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

Returns

the old value being replaced

Exceptions

std::runtime_error	if the heap is empty
std::range_error	if the index is out of range

Definition at line 590 of file mmheap.h.

1.2.2.9 template<typename DataType > void mmheap::make_heap (DataType ∗ *heap_array,* size_t *size*)

make an arbitrary array into a heap (in-place)

Applies Floyd's algorithm (adapted to a min-max heap) to produce a heap from an arbitrary array in linear time.

Parameters

heap_array	the array that will become a heap
size	the number of elements in the array

Template Parameters

DataType	the type of data stored in the heap - must be LessThanComparable, Swappable, CopyConstructable, and CopyAssignable

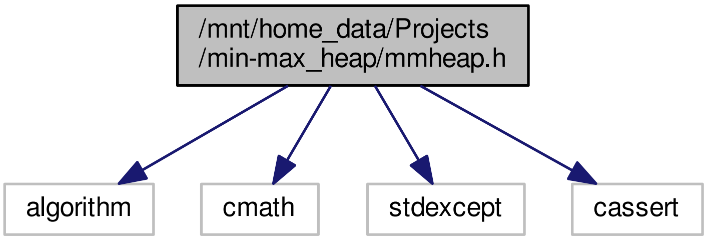
Definition at line 459 of file mmheap.h.

2 File Documentation

2.1 /mnt/home_data/Projects/min-max_heap/mmheap.h File Reference

```
#include <algorithm>
#include <cmath>
#include <stdexcept>
#include <cassert>
```

Include dependency graph for mmheap.h:



Namespaces

- _mmheap
- mmheap

Functions

- size_t _mmheap::parent (size_t i)
- size_t _mmheap::has_parent (size_t i)
- size_t _mmheap::left (size_t i)
- size_t _mmheap::right (size_t i)
- size_t _mmheap::gparent (size_t i)
- bool _mmheap::has_gparent (size_t i)
- bool _mmheap::child (size_t i, size_t c)
- uint64_t _mmheap::log_2 (uint64_t i)
- bool _mmheap::min_level (size_t i)
- template< typename DataType >
  std::pair< bool, size_t > _mmheap::min_child (DataType ∗heap_array, size_t i, size_t right_index)

- template<typename DataType >
  std::pair< bool, size_t > _mmheap::min_gchild (DataType ∗heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > _mmheap::min_child_or_gchild (DataType ∗heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > _mmheap::max_child (DataType ∗heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > _mmheap::max_gchild (DataType ∗heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > _mmheap::max_child_or_gchild (DataType ∗heap_array, size_t i, size_t right_index)
- template<typename DataType >
  void _mmheap::sift_down_min (DataType ∗heap_array, size_t sift_index, size_t right_index)
- template<typename DataType >
  void _mmheap::sift_down_max (DataType ∗heap_array, size_t sift_index, size_t right_index)
- template<typename DataType >
  void _mmheap::sift_down (DataType ∗heap_array, size_t sift_index, size_t right_index)
- template<typename DataType >
  void _mmheap::bubble_up_min (DataType ∗heap_array, size_t bubble_index)
- template<typename DataType >
  void _mmheap::bubble_up_max (DataType ∗heap_array, size_t bubble_index)
- template<typename DataType >
  void _mmheap::bubble_up (DataType ∗heap_array, size_t bubble_index)
- template<typename DataType >
  void mmheap::make_heap (DataType ∗heap_array, size_t size)

  make an arbitrary array into a heap (in-place)
- template<typename DataType >
  void mmheap::heap_insert (const DataType &value, DataType ∗heap_array, size_t &count, size_t max_size)
- template<typename DataType >
  DataType mmheap::heap_max (DataType ∗heap_array, size_t count)
- template<typename DataType >
  DataType mmheap::heap_min (DataType ∗heap_array, size_t count)
- template<typename DataType >
  std::pair< bool, DataType > mmheap::heap_insert_circular (const DataType &value, DataType ∗heap_array, size_t &count, size_t max_size)

  add to heap, rotating the maximum value out if the heap is full
- template<typename DataType >
  DataType mmheap::heap_replace_at_index (const DataType &new_value, size_t index, DataType ∗heap_array, size_t count)
- template<typename DataType >
  DataType mmheap::heap_remove_at_index (size_t index, DataType ∗heap_array, size_t &count)
- template<typename DataType >
  DataType mmheap::heap_remove_min (DataType ∗heap_array, size_t &count)
- template<typename DataType >
  DataType mmheap::heap_remove_max (DataType ∗heap_array, size_t &count)

2.1.1 Detailed Description

Defines functions for maintaining a Min-Max Heap, as described by Adkinson: M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte. 1986. Min-max heaps and generalized priority queues. Commun. ACM 29, 10 (October 1986), 996-1000. DOI=`http://dx.doi.org/10.1145/6617.6621`

This file defines two namespaces:

- The mmheap namespace defines functions that are useful for building and maintaining a Min-Max heap. All necessary ("public-facing") functionality is in this namespace.
- The The _mmheap namespace contains functions that are only intended for internal use by the "public-facing" functions in the mmheap namespace. None of the functions in _mmheap:: should be necessary externally.

Author

Jason L Causey Released under the MIT License: http://opensource.org/licenses/MIT

Copyright

Definition in file mmheap.h.

2.2 /mnt/home_data/Projects/min-max_heap/mmheap.h

```
#ifndef MMHEAP_H
#define MMHEAP_H
/**
 * @file mmheap.h
 *
 * Defines functions for maintaining a Min-Max Heap,
 * as described by Adkinson:
 *     M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte. 1986.
 *     Min-max heaps and generalized priority queues.
 *     Commun. ACM 29, 10 (October 1986), 996-1000.
 *     DOI=http://dx.doi.org/10.1145/6617.6621
 *
 * @details
 *   This file defines two namespaces:
 *     * The `mmheap` namespace defines functions that are useful for building and
 *       maintaining a Min-Max heap.  All necessary ("public-facing") functionality
 *       is in this namespace.
 *     * The The `_mmheap` namespace contains functions that are only intended for
 *       internal use by the "public-facing" functions in the `mmheap` namespace.
 *       None of the functions in `_mmheap::` should be necessary externally.
 *
 * @author    Jason L Causey
 * @license   Released under the MIT License: http://opensource.org/licenses/MIT
 * @copyright Copyright (c) 2015 Jason L Causey, Arkansas State University
 *
 *   Permission is hereby granted, free of charge, to any person obtaining a copy
 *   of this software and associated documentation files (the "Software"), to deal
 *   in the Software without restriction, including without limitation the rights
 *   to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 *   copies of the Software, and to permit persons to whom the Software is
 *   furnished to do so, subject to the following conditions:
 *
 *   The above copyright notice and this permission notice shall be included in
 *   all copies or substantial portions of the Software.
 *
 *   THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 *   IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 *   FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL THE
 *   AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 *   LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 *   OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 *   THE SOFTWARE.
 */

#include <algorithm>
#include <cmath>
#include <stdexcept>
#include <cassert>

/**
 * The `_mmheap` namespace contains functions that are only intended for internal
 * use by the "public-facing" functions in the `mmheap` namespace.  None of the
 * functions in `_mmheap::` should be necessary externally.
 */
namespace _mmheap{

    inline size_t  parent(size_t i)          { return (i - 1) / 2;
          }
    inline size_t  has_parent(size_t i)      { return i > 0;
              }
    inline size_t  left  (size_t i)          { return 2*i + 1;
        }
    inline size_t  right (size_t i)          { return 2*i + 2;
         }
    inline size_t  gparent(size_t i)         { return parent(parent(i));
           }
    inline bool    has_gparent(size_t i)     { return i > 2;
  }
    inline bool    child(size_t i, size_t c) { return c == left(i) || c == right(i);    }

    /*
     * fast log-base-2 based on code from:
     *     http://stackoverflow.com/a/11398748
     * @param  i value to compute the log_2 for (must be > 0)
     * @return log-base-2 of `i`
     */
    uint64_t log_2(uint64_t i) {
        static const uint64_t tab64[64] = {
            63,  0, 58,  1, 59, 47, 53,  2,
            60, 39, 48, 27, 54, 33, 42,  3,
            61, 51, 37, 40, 49, 18, 28, 20,
            55, 30, 34, 11, 43, 14, 22,  4,
            62, 57, 46, 52, 38, 26, 32, 41,
            50, 36, 17, 19, 29, 10, 13, 21,
            56, 45, 25, 31, 35, 16,  9, 12,
            44, 24, 15,  8, 23,  7,  6,  5
        };
        i |= i >> 1;
        i |= i >> 2;
        i |= i >> 4;
        i |= i >> 8;
        i |= i >> 16;
        i |= i >> 32;
        return tab64[((uint64_t)((i - (i >> 1))*0x07EDD5E59A4E28C2)) >> 58];
    }

    /**
     * returns `true` if `i` is on a Min-Level
     *
     * @param   i index into the heap
     * @return  `true` if `i` is on a min-level
     */
    inline bool min_level(size_t i) {
        return i > 0 ? log_2(++i) % 2 == 0 : true;
    }

    /**
     * get a pair considing of an indication of whether `i` has any children, and
     * if so, the index of the child containing the minimum value.
     *
     * @param   heap_array  the heap
     * @param   i           the index (parent) for which to find the min-child
     * @param   right-index the index of the right-most element that is part of the heap
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     * @return  a pair where the first element is `true` if `i` has children (`false`
     *          otherwise), and the second element is the index of the child whose value
     *          is smallest (only if the first element is `true`)
     */
    template <typename DataType>
    std::pair<bool, size_t> min_child(DataType*
heap_array, size_t i, size_t right_index){
        std::pair<bool, size_t> result{false, 0};
        if(left(i) <= right_index){
            auto m = left(i);
            if(right(i) <= right_index && heap_array[right(i)] < heap_array[m]){
                m = right(i);
            }
            result = {true, m};
        }
        return result;
    }

    /**
     * get a pair considing of an indication of whether `i` has any grandchildren, and
     * if so, the index of the grandchild containing the minimum value.
     *
     * @param   heap_array  the heap
     * @param   i           the index (parent) for which to find the min-grandchild
     * @param   right-index the index of the right-most element that is part of the heap
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     * @return  a pair where the first element is `true` if `i` has grandchildren
     *          (`false` otherwise), and the second element is the index of the
     *          grandchild whose value is smallest (only if the first element is `true`)
     */
    template <typename DataType>
    std::pair<bool, size_t> min_gchild(DataType*
heap_array, size_t i, size_t right_index){
        std::pair<bool, size_t> result{false, 0};
        auto l = left(i);
        auto r = right(i);
        if(left(l) <= right_index){
            auto m = left(l);
            if(right(l) <= right_index && heap_array[right(l)] < heap_array[m]){
                m = right(l);
            }
            if(left(r) <= right_index && heap_array[left(r)] < heap_array[m]){
                m = left(r);
            }
            if(right(r) <= right_index && heap_array[right(r)] < heap_array[m]){
                m = right(r);
            }
            result = {true, m};
        }
        return result;
    }

    /**
     * get a pair considing of an indication of whether `i` has any children, and
     * if so, the index of the child or grandchild containing the minimum value.
     *
     * @param   heap_array  the heap
     * @param   i           the index (parent) for which to find the min-(grand)child
     * @param   right-index the index of the right-most element that is part of the heap
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     * @return  a pair where the first element is `true` if `i` has children
     *          (`false` otherwise), and the second element is the index of the
     *          child or grandchild whose value is smallest (only if the first
     *          element is `true`)
     */
    template <typename DataType>
    std::pair<bool, size_t> min_child_or_gchild(
DataType* heap_array, size_t i, size_t
right_index){
        auto m = min_child(heap_array, i, right_index);
        if(m.first){
            auto  gm = min_gchild(heap_array, i, right_index);
            m.second = gm.first && heap_array[gm.second] < heap_array[m.second] ? gm.second : m.second
;
```

```
        }
        return m;
    }

    /**
     * get a pair considing of an indication of whether `i` has any children, and
     * if so, the index of the child containing the maximum value.
     *
     * @param   heap_array  the heap
     * @param   i           the index (parent) for which to find the max-child
     * @param   right-index the index of the right-most element that is part of the heap
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     * @return  a pair where the first element is `true` if `i` has children (`false`
     *          otherwise), and the second element is the index of the child whose value
     *          is largest (only if the first element is `true`)
     */
    template <typename DataType>
    std::pair<bool, size_t> max_child(DataType* heap_array, size_t i, size_t right_index){
        std::pair<bool, size_t> result {false, 0};
        if(left(i) <= right_index){
            auto m = left(i);
            if(right(i) <= right_index && heap_array[m] < heap_array[right(i)]){
                m = right(i);
            }
            result = {true, m};
        }
        return result;
    }

    /**
     * get a pair considing of an indication of whether `i` has any grandchildren, and
     * if so, the index of the grandchild containing the maximum value.
     *
     * @param   heap_array  the heap
     * @param   i           the index (parent) for which to find the max-grandchild
     * @param   right-index the index of the right-most element that is part of the heap
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     * @return  a pair where the first element is `true` if `i` has grandchildren
     *          (`false` otherwise), and the second element is the index of the
     *          grandchild whose value is largest (only if the first element is `true`)
     */
    template <typename DataType>
    std::pair<bool, size_t> max_gchild(DataType* heap_array, size_t i, size_t right_index){
        std::pair<bool, size_t> result{false, 0};
        auto l = left(i);
        auto r = right(i);
        if(left(l) <= right_index){
            auto m = left(l);
            if(right(l) <= right_index && heap_array[m] < heap_array[right(l)]){
                m = right(l);
            }
            if(left(r) <= right_index && heap_array[m] < heap_array[left(r)]){
                m = left(r);
            }
            if(right(r) <= right_index && heap_array[m] < heap_array[right(r)]){
                m = right(r);
            }
            result = {true, m};
        }
        return result;
    }

    /**
     * get a pair considing of an indication of whether `i` has any children, and
     * if so, the index of the child or grandchild containing the maximum value.
     *
     * @param   heap_array  the heap
     * @param   i           the index (parent) for which to find the max-(grand)child
     * @param   right-index the index of the right-most element that is part of the heap
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     * @return  a pair where the first element is `true` if `i` has children
     *          (`false` otherwise), and the second element is the index of the
     *          child or grandchild whose value is largest (only if the first
     *          element is `true`)
     */
    template <typename DataType>
    std::pair<bool, size_t> max_child_or_gchild(DataType* heap_array, size_t i, size_t right_index){
        auto m = max_child(heap_array, i, right_index);
        if(m.first){
            auto gm  = max_gchild(heap_array, i, right_index);
            m.second = gm.first &&  heap_array[m.second] < heap_array[gm.second] ? gm.second : m.second;
        }
        return m;
    }

    /**
     * perform min-max heap sift-down on an element (at `sift_index`) that is on a min-level
     *
     * @param heap_array  the heap
     * @param sift_index  the index of the element that should be sifted down
     * @param right_index the index of the right-most element that is part of the heap
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     */
    template <typename DataType>
    void sift_down_min(DataType* heap_array, size_t sift_index, size_t right_index){
        bool sift_more = true;
        while(sift_more && left(sift_index) <= right_index){                            // if a[i] has children
            sift_more = false;
            auto mp = min_child_or_gchild(heap_array, sift_index, right_index);         // get min child or grandchild
            auto m  = mp.second;
            if(child(sift_index, m)){                                                   // if the min was a child
                if(heap_array[m] < heap_array[sift_index]){
                    std::swap(heap_array[m], heap_array[sift_index]);
                }
            }
            else{                                                                       // min was a grandchild
                if(heap_array[m] < heap_array[sift_index]){
                    std::swap(heap_array[m], heap_array[sift_index]);
                    if(heap_array[parent(m)] < heap_array[m]){
                        std::swap(heap_array[m], heap_array[parent(m)]);
                    }
                    sift_index = m;
                    sift_more  = true;
                }
            }
        }
    }

    /**
     * perform min-max heap sift-down on an element (at `sift_index`) that is on a max-level
     *
     * @param heap_array  the heap
     * @param sift_index  the index of the element that should be sifted down
     * @param right_index the index of the right-most element that is part of the heap
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     */
    template <typename DataType>
    void sift_down_max(DataType* heap_array, size_t sift_index, size_t right_index){
        bool sift_more = true;
        while(sift_more && left(sift_index) <= right_index){                            // if a[i] has children
            sift_more = false;
            auto mp = max_child_or_gchild(heap_array, sift_index, right_index);         // get max child or grandchild
            auto m  = mp.second;
            if(child(sift_index, m)){                                                   // if the max was a child
                if(heap_array[sift_index] < heap_array[m]){
                    std::swap(heap_array[m], heap_array[sift_index]);
                }
            }
            else{                                                                       // max was a grandchild
```

```
                if(heap_array[sift_index] < heap_array[m]){
                    std::swap(heap_array[m], heap_array[sift_index]);
                    if(heap_array[m] < heap_array[parent(m)]){
                        std::swap(heap_array[m], heap_array[parent(m)]);
                    }
                    sift_index = m;
                    sift_more  = true;
                }
            }
        }
    }

    /**
     * perform min-max heap sift-down on an element (at `sift_index`)
     *
     * @param heap_array  the heap
     * @param sift_index  the index of the element that should be sifted down
     * @param right_index the index of the right-most element that is part of the heap
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     */
    template <typename DataType>
    void sift_down(DataType* heap_array, size_t sift_index, size_t right_index){
        if(min_level(sift_index)){
            sift_down_min(heap_array, sift_index, right_index);
        }
        else{
            sift_down_max(heap_array, sift_index, right_index);
        }
    }

    /**
     * perform min-max heap bubble-up on an element (at `bubble_index`) that is on a min-level
     *
     * @param heap_array    the heap
     * @param bubble_index  the index of the element that should be bubbled up
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     */
    template <typename DataType>
    void bubble_up_min(DataType* heap_array, size_t bubble_index){
        bool finished = false;
        while(!finished && has_gparent(bubble_index)){
            finished = true;
            if(heap_array[bubble_index] < heap_array[gparent(bubble_index)]){
                std::swap(heap_array[bubble_index], heap_array[gparent(bubble_index)]);
                bubble_index = gparent(bubble_index);
                finished     = false;
            }
        }
    }

    /**
     * perform min-max heap bubble-up on an element (at `bubble_index`) that is on a max-level
     *
     * @param heap_array    the heap
     * @param bubble_index  the index of the element that should be bubbled up
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     */
    template <typename DataType>
    void bubble_up_max(DataType* heap_array, size_t bubble_index){
        bool finished = false;
        while(!finished && has_gparent(bubble_index)){
            finished = true;
            if(heap_array[gparent(bubble_index)] < heap_array[bubble_index]){
                std::swap(heap_array[bubble_index], heap_array[gparent(bubble_index)]);
                bubble_index = gparent(bubble_index);
                finished     = false;
            }
        }
    }

    /**
     * perform min-max heap bubble-up on an element (at `bubble_index`)
     *
     * @param heap_array    the heap
     * @param bubble_index  the index of the element that should be bubbled up
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     */
    template <typename DataType>
    void bubble_up(DataType* heap_array, size_t bubble_index){
        if(min_level(bubble_index)){
            if(has_parent(bubble_index) && heap_array[parent(bubble_index)] < heap_array[bubble_index]
){
                std::swap(heap_array[bubble_index], heap_array[parent(bubble_index)]);
                bubble_up_max(heap_array, parent(bubble_index));
            }
            else{
                bubble_up_min(heap_array, bubble_index);
            }
        }
        else{
            if(has_parent(bubble_index) && heap_array[bubble_index] < heap_array[parent(bubble_index)]
){
                std::swap(heap_array[bubble_index], heap_array[parent(bubble_index)]);
                bubble_up_min(heap_array, parent(bubble_index));
            }
            else{
                bubble_up_max(heap_array, bubble_index);
            }
        }
    }
}

/**
 * The `mmheap` namespace defines functions that are useful for building and
 * maintaining a Min-Max heap.  All necessary ("public-facing") functionality
 * is in this namespace.
 */
namespace mmheap{
    /**
     * @brief   make an arbitrary array into a heap (in-place)
     * @details Applies Floyd's algorithm (adapted to a min-max heap) to produce
     *          a heap from an arbitrary array in linear time.
     *
     * @param heap_array    the array that will become a heap
     * @param size          the number of elements in the array
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     */
    template <typename DataType>
    void make_heap(DataType* heap_array, size_t size){
        if(size > 1){
            bool finished = false;
            for(size_t current = _mmheap::parent(size-1); !finished; --current){
                _mmheap::sift_down(heap_array, current, size-1);
                finished = current == 0;
            }
        }
    }

    /**
     * insert a new value to the heap (and update the `count`)
     *
     * @param           value       the new value to insert
     * @param           heap_array  the heap
     * @param[in,out]   count       the current number of items in the heap (will update)
     * @param           max_size    the physical storage allocation size of the heap
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     * @throws std::runtime_error if the heap is full prior to the insert operation
     */
    template <typename DataType>
    void heap_insert(const DataType& value, DataType* heap_array, size_t& count,
size_t max_size){
        if(count < max_size){
            heap_array[count++] = value;
            _mmheap::bubble_up(heap_array, count-1);
        }
        else{
            throw std::runtime_error("Cannot insert into heap - allocated size is full.");
        }
    }

    /**
     * get the maximum value in the heap
     *
     * @param heap_array the heap
     * @param count      the current number of values contained in the heap
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     * @return the maximum value in the heap
     * @throws std::runtime_error if the heap is empty
     */
    template <typename DataType>
    DataType heap_max(DataType* heap_array, size_t count){
        if(count < 1){
            throw std::runtime_error("Cannot get max value in empty heap.");
        }
        auto m = _mmheap::max_child(heap_array, 0, count-1);
        return m.first ? heap_array[m.second] : heap_array[0];
    }

    /**
     * get the minimum value in the heap
     *
     * @param heap_array the heap
     * @param count      the current number of values contained in the heap
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     * @return the minimum value in the heap
     * @throws std::runtime_error if the heap is empty
     */
    template <typename DataType>
    DataType heap_min(DataType* heap_array, size_t count){
        if(count < 1){
            throw std::runtime_error("Cannot get min value in empty heap.");
        }
        return heap_array[0];
    }

    /**
     * @brief   add to heap, rotating the maximum value out if the heap is full
     * @details Add to the min-max heap in such a way that the maximum value is removed
     *          at the same time if the heap has reached its storage capacity.
     *
     * @param         value         new value to add
     * @param         heap_array    the heap
     * @param[in,out] count         number of values currently in the heap (will update)
     * @param         max_size      maximum physical size allocated for the heap
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      DefaultConstructable, LessThanComparable, Swappable,
     *                      CopyConstructable, and CopyAssignable
     * @return a pair consising of a flag and a value; the first element is a flag
     *         indicating that overflow occurred, and the second element is the value
     *         that rotated out of the heap (formerly the maximum) when the new value
     *         was added (set only if an overflow occurred)
     */
    template <typename DataType>
    std::pair<bool, DataType> heap_insert_circular(const DataType& value, DataType* heap_array, size_t& count, size_t max_size){
        auto max_value  = DataType{};
        bool overflowed = count == max_size ? true : false;
        if(!overflowed){
            heap_insert(value, heap_array, count, max_size);
        }
        else{                                                   // if the heap is full, replace the max value with the new add...
            auto m        = max_size > 1 ? _mmheap::max_child(heap_array, 0, max_size-1).second : 0;
            max_value     = heap_array[m];
            if(value < max_value){                                  // if the new value is larger than the one rotating out, just rotate the new value
                heap_array[m] = value;
                if(max_size > 1){                                   // if this is non-trivial
                    if(value < heap_array[0]){                      // check that the new value isn't the new min
                        std::swap(heap_array[0], heap_array[m]);    //  (if it is, make it so)
                    }
                    _mmheap::sift_down(heap_array, m, max_size-1);  // sift the new item down
                }
            }
            else{
                max_value = value;
            }
        }
        return std::pair<bool, DataType>{overflowed, max_value};
    }


    /**
     * replace and return the value at a given index with a new value
     *
     * @param new_value   new value to insert
     * @param index       index of the value to replace
     * @param heap_array  the heap
     * @param count       number of values currently stored in the heap
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     * @return  the old value being replaced
     * @throws  std::runtime_error if the heap is empty
     * @throws  std::range_error   if the index is out of range
     */
    template <typename DataType>
    DataType heap_replace_at_index(const DataType& new_value, size_t index, DataType* heap_array, size_t count){
        if(count == 0){
            throw std::runtime_error("Cannot replace value in empty heap.");
        }
        if(index > count){
            throw std::range_error("Index beyond end of heap.");
        }
        auto old_value      = heap_array[index];
        heap_array[index] = new_value;
        if(_mmheap::min_level(index)){
            if(new_value < old_value){
                _mmheap::bubble_up_min(heap_array, index);
            }
            else{
                if(_mmheap::has_parent(index) && heap_array[_mmheap::parent(index)] < new_value){
                    _mmheap::bubble_up(heap_array, index);
                }
                _mmheap::sift_down(heap_array, index, count-1);
            }
        }
        else{
            if(old_value < new_value){
                _mmheap::bubble_up_max(heap_array, index);
            }
            else{
                if(_mmheap::has_parent(index) && new_value < heap_array[_mmheap::parent(index)]){
                    _mmheap::bubble_up(heap_array, index);
                }
                _mmheap::sift_down(heap_array, index, count-1);
            }
        }
        return old_value;
    }

    /**
     * remove and return value at a given index
     *
     * @param         index      index to remove
     * @param         heap_array the heap
     * @param[in,out] count      current number of values in the heap (will update)
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     * @return  the value being removed
     * @throws  std::runtime_error if the heap is empty
     * @throws  std::range_error   if the index is out of range
     */
    template <typename DataType>
    DataType heap_remove_at_index(size_t index, DataType* heap_array, size_t& count){
        if(count == 0){
            throw std::runtime_error("Cannot remove value in empty heap.");
        }
        if(index > count){
            throw std::range_error("Index beyond end of heap.");
        }
        auto old_value = heap_replace_at_index(heap_array[count-1], index, heap_array, count);
        --count;
        return old_value;
    }

    /**
     * remove and return the minimum value in the heap
     *
     * @param heap_array the array
     * @param count      the current number of values in the heap (will update)
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     * @return the minimum value in the heap
     * @throws std::runtime_error if the heap is empty
     */
    template <typename DataType>
    DataType heap_remove_min(DataType* heap_array, size_t& count){
        if(count == 0){
            throw std::runtime_error("Cannot remove from empty heap.");
        }
        auto value = heap_array[0];
        std::swap(heap_array[0], heap_array[count-1]);
        --count;
        if(count > 0){
            _mmheap::sift_down(heap_array, 0, count-1);
        }
        return value;
    }

    /**
     * remove and return the maximum value in the heap
     *
     * @param heap_array the array
     * @param count      the current number of values in the heap (will update)
     * @tparam  DataType    the type of data stored in the heap - must be
     *                      LessThanComparable, Swappable, CopyConstructable,
     *                      and CopyAssignable
     * @return the maximum value in the heap
     * @throws std::runtime_error if the heap is empty
     */
    template <typename DataType>
    DataType heap_remove_max(DataType* heap_array, size_t& count){
        if(count == 0){
            throw std::runtime_error("Cannot remove from empty heap.");
        }
        auto value = heap_array[0];
        auto m     = _mmheap::max_child(heap_array, 0, count-1);
        if(m.first){
            value = heap_array[m.second];
        }
        else{
            m.second = 0;
        }
        heap_remove_at_index(m.second, heap_array, count);
        return value;
    }
}

#endif
```

Index