Min-Max Heap

Generated by Doxygen 1.8.9.1

Sun Dec 20 2015 18:23:24

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)
- std::pair< bool, size_t > min_child (int *heap_array, size_t i, size_t right_index)
- std::pair< bool, size_t > min_gchild (int *heap_array, size_t i, size_t right_index)
- std::pair< bool, size_t > min_child_or_gchild (int *heap_array, size_t i, size_t right_index)
- std::pair< bool, size_t > max_child (int *heap_array, size_t i, size_t right_index)
- std::pair< bool, size_t > max_gchild (int *heap_array, size_t i, size_t right_index)
- std::pair< bool, size_t > max_child_or_gchild (int *heap_array, size_t i, size_t right_index)
- void sift_down_min (int *heap_array, size_t sift_index, size_t right_index)
- void sift_down_max (int *heap_array, size_t sift_index, size_t right_index)
- void sift_down (int *heap_array, size_t sift_index, size_t right_index)
- void bubble_up_min (int *heap_array, size_t bubble_index)
- void bubble_up_max (int *heap_array, int bubble_index)
- void bubble_up (int *heap_array, int 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 void _mmheap::bubble_up (int * *heap_array*, int *bubble_index*)

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

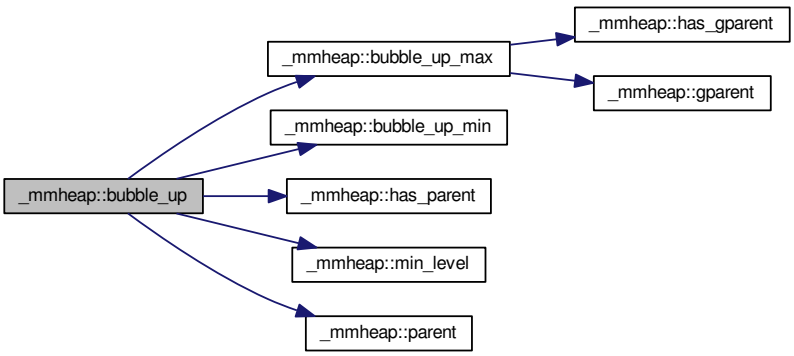
Parameters

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

Definition at line 372 of file mmheap.h.

References bubble_up_max(), bubble_up_min(), has_parent(), min_level(), and parent().

Here is the call graph for this function:



1.1.2.2 void _mmheap::bubble_up_max (int * *heap_array*, int *bubble_index*)

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

Parameters

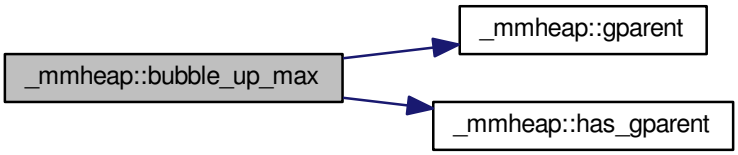
heap_array	the heap
bubble_index	the index of the element that should be bubbled up

Definition at line 354 of file mmheap.h.

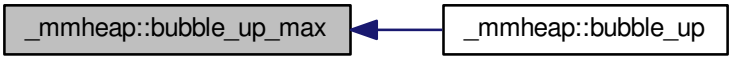
References gparent(), and has_gparent().

Referenced by bubble_up().

Here is the call graph for this function:



Here is the caller graph for this function:



1.1.2.3 void _mmheap::bubble_up_min (int * *heap_array*, size_t *bubble_index*)

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

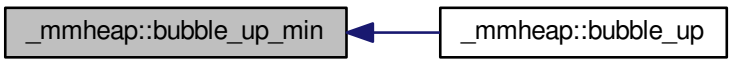
Parameters

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

Definition at line 336 of file mmheap.h.

Referenced by bubble_up().

Here is the caller graph for this function:



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

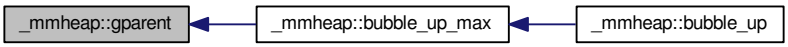
Definition at line 63 of file mmheap.h.

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

Definition at line 61 of file mmheap.h.

Referenced by bubble_up_max().

Here is the caller graph for this function:

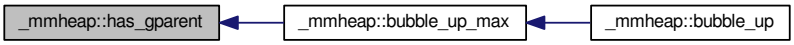


1.1.2.6 bool _mmheap::has_gparent (size_t *i*) [inline]

Definition at line 62 of file mmheap.h.

Referenced by bubble_up_max().

Here is the caller graph for this function:



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

Definition at line 58 of file mmheap.h.

Referenced by bubble_up().

Here is the caller graph for this function:

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 std::pair<bool, size_t> _mmheap::max_child (int * *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

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 187 of file mmheap.h.

1.1.2.11 std::pair<bool, size_t> _mmheap::max_child_or_gchild (int * *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

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 243 of file mmheap.h.

1.1.2.12 std::pair<bool, size_t> _mmheap::max_gchild (int * *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

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 211 of file mmheap.h.

1.1.2.13 std::pair<bool, size_t> _mmheap::min_child (int * *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

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 112 of file mmheap.h.

1.1.2.14 std::pair<bool, size_t> _mmheap::min_child_or_gchild (int * *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

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 167 of file mmheap.h.

1.1.2.15 std::pair<bool, size_t> _mmheap::min_gchild (int * *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

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 135 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.

Referenced by bubble_up().

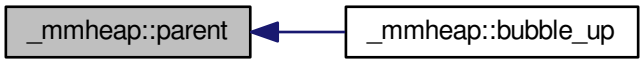
Here is the caller graph for this function:

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

Definition at line 57 of file mmheap.h.

Referenced by bubble_up().

Here is the caller graph for this function:



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

Definition at line 60 of file mmheap.h.

1.1.2.19 void _mmheap::sift_down (int * *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

Definition at line 321 of file mmheap.h.

1.1.2.20 void _mmheap::sift_down_max (int * *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

Definition at line 290 of file mmheap.h.

1.1.2.21 void _mmheap::sift_down_min (int * *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

Definition at line 259 of file mmheap.h.

1.2 mmheap Namespace Reference

Functions

- void make_heap (int *heap_array, size_t size)

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

  add to heap, rotating the maximum value out if the heap is full
- int heap_replace_at_index (int new_value, size_t index, int *heap_array, size_t count)
- int heap_remove_at_index (size_t index, int *heap_array, size_t &count)
- int heap_remove_min (int *heap_array, size_t &count)
- int heap_remove_max (int *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 void mmheap::heap_insert (int *value,* int * *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

Exceptions

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

Definition at line 425 of file mmheap.h.

1.2.2.2 std::pair<bool, int> mmheap::heap_insert_circular (int *value*, int * *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

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 483 of file mmheap.h.

1.2.2.3 int mmheap::heap_max (int * *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

Returns

the maximum value in the heap

Exceptions

std::runtime_error	if the heap is empty

Definition at line 444 of file mmheap.h.

1.2.2.4 int mmheap::heap_min (int * *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

Returns

the minimum value in the heap

Exceptions

std::runtime_error	if the heap is empty

Definition at line 461 of file mmheap.h.

1.2.2.5 int mmheap::heap_remove_at_index (size_t *index*, int * *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)

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 566 of file mmheap.h.

1.2.2.6 int mmheap::heap_remove_max (int * *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)

Returns

the maximum value in the heap

Exceptions

std::runtime_error	if the heap is empty

Definition at line 609 of file mmheap.h.

1.2.2.7 int mmheap::heap_remove_min (int * *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)

Returns

the minimum value in the heap

Exceptions

std::runtime_error	if the heap is empty

Definition at line 587 of file mmheap.h.

1.2.2.8 int mmheap::heap_replace_at_index (int *new_value*, size_t *index*, int * *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

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 521 of file mmheap.h.

1.2.2.9 void mmheap::make_heap (int * *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

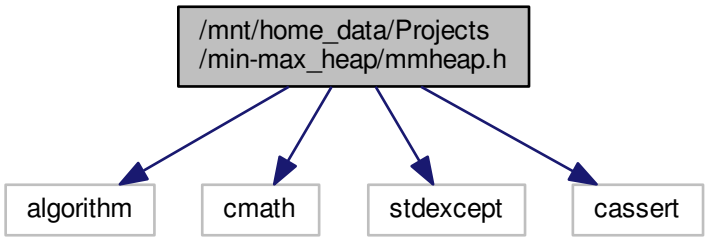
Definition at line 408 of file mmheap.h.

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

  make an arbitrary array into a heap (in-place)
- void mmheap::heap_insert (int value, int *heap_array, size_t &count, size_t max_size)
- int mmheap::heap_max (int *heap_array, size_t count)
- int mmheap::heap_min (int *heap_array, size_t count)

- std::pair< bool, int > mmheap::heap_insert_circular (int value, int *heap_array, size_t &count, size_t max_size)

  add to heap, rotating the maximum value out if the heap is full
- int mmheap::heap_replace_at_index (int new_value, size_t index, int *heap_array, size_t count)
- int mmheap::heap_remove_at_index (size_t index, int *heap_array, size_t &count)
- int mmheap::heap_remove_min (int *heap_array, size_t &count)
- int mmheap::heap_remove_max (int *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

```
00001 #ifndef MMHEAP_H
00002 #define MMHEAP_H
00003 /**
00004  * @file mmheap.h
00005  *
00006  * Defines functions for maintaining a Min-Max Heap,
00007  * as described by Adkinson:
00008  *     M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte. 1986.
```

```
00009  *     Min-max heaps and generalized priority queues.
00010  *     Commun. ACM 29, 10 (October 1986), 996-1000.
00011  *     DOI=http://dx.doi.org/10.1145/6617.6621
00012  *
00013  * @details
00014  *   This file defines two namespaces:
00015  *     * The `mmheap` namespace defines functions that are useful for building and
00016  *       maintaining a Min-Max heap.  All necessary ("public-facing") functionality
00017  *       is in this namespace.
00018  *     * The The `_mmheap` namespace contains functions that are only intended for
00019  *       internal use by the "public-facing" functions in the `mmheap` namespace.
00020  *       None of the functions in `_mmheap::` should be necessary externally.
00021  *
00022  * @author    Jason L Causey
00023  * @license   Released under the MIT License: http://opensource.org/licenses/MIT
00024  * @copyright Copyright (c) 2015 Jason L Causey, Arkansas State University
00025  *
00026  *   Permission is hereby granted, free of charge, to any person obtaining a copy
00027  *   of this software and associated documentation files (the "Software"), to deal
00028  *   in the Software without restriction, including without limitation the rights
00029  *   to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
00030  *   copies of the Software, and to permit persons to whom the Software is
00031  *   furnished to do so, subject to the following conditions:
00032  *
00033  *   The above copyright notice and this permission notice shall be included in
00034  *   all copies or substantial portions of the Software.
00035  *
00036  *   THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
00037  *   IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
00038  *   FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL THE
00039  *   AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
00040  *   LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
00041  *   OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
00042  *   THE SOFTWARE.
00043  */
00044
00045 #include <algorithm>
00046 #include <cmath>
00047 #include <stdexcept>
00048 #include <cassert>
00049
00050 /**
00051  * The `_mmheap` namespace contains functions that are only intended for internal
00052  * use by the "public-facing" functions in the `mmheap` namespace.  None of the
00053  * functions in `_mmheap::` should be necessary externally.
00054  */
00055 namespace _mmheap{
00056
00057     inline size_t  parent(size_t i)          { return (i - 1) / 2;
              }
00058     inline size_t  has_parent(size_t i)      { return i > 0;
                }
00059     inline size_t  left  (size_t i)          { return 2*i + 1;
            }
00060     inline size_t  right (size_t i)          { return 2*i + 2;
             }
00061     inline size_t  gparent(size_t i)         { return parent(parent(i));
               }
00062     inline bool    has_gparent(size_t i)     { return i > 2;
    }
00063     inline bool    child(size_t i, size_t c) { return c == left(i) || c == right(i);    }
00064
00065     /*
00066      * fast log-base-2 based on code from:
00067      *     http://stackoverflow.com/a/11398748
00068      * @param  i value to compute the log_2 for (must be > 0)
00069      * @return log-base-2 of `i`
00070      */
00071     uint64_t log_2(uint64_t i) {
00072         static const uint64_t tab64[64] = {
00073             63,  0, 58,  1, 59, 47, 53,  2,
00074             60, 39, 48, 27, 54, 33, 42,  3,
00075             61, 51, 37, 40, 49, 18, 28, 20,
00076             55, 30, 34, 11, 43, 14, 22,  4,
00077             62, 57, 46, 52, 38, 26, 32, 41,
00078             50, 36, 17, 19, 29, 10, 13, 21,
00079             56, 45, 25, 31, 35, 16,  9, 12,
00080             44, 24, 15,  8, 23,  7,  6,  5
00081         };
00082         i |= i >> 1;
00083         i |= i >> 2;
```

```
00084         i |= i >> 4;
00085         i |= i >> 8;
00086         i |= i >> 16;
00087         i |= i >> 32;
00088         return tab64[((uint64_t)((i - (i >> 1))*0x07EDD5E59A4E28C2)) >> 58];
00089     }
00090
00091     /**
00092      * returns `true` if `i` is on a Min-Level
00093      *
00094      * @param   i index into the heap
00095      * @return  `true` if `i` is on a min-level
00096      */
00097     inline bool min_level(size_t i) {
00098         return i > 0 ? log_2(++i) % 2 == 0 : true;
00099     }
00100
00101     /**
00102      * get a pair considing of an indication of whether `i` has any children, and
00103      * if so, the index of the child containing the minimum value.
00104      *
00105      * @param   heap_array  the heap
00106      * @param   i           the index (parent) for which to find the min-child
00107      * @param   right-index the index of the right-most element that is part of the heap
00108      * @return  a pair where the first element is `true` if `i` has children (`false`
00109      *          otherwise), and the second element is the index of the child whose value
00110      *          is smallest (only if the first element is `true`)
00111      */
00112     std::pair<bool, size_t> min_child(int*
      heap_array, size_t i, size_t right_index){
00113         std::pair<bool, size_t> result{false, 0};
00114         if(left(i) <= right_index){
00115             auto m = left(i);
00116             if(right(i) <= right_index && heap_array[right(i)] < heap_array[m]){
00117                 m = right(i);
00118             }
00119             result = {true, m};
00120         }
00121         return result;
00122     }
00123
00124     /**
00125      * get a pair considing of an indication of whether `i` has any grandchildren, and
00126      * if so, the index of the grandchild containing the minimum value.
00127      *
00128      * @param   heap_array  the heap
00129      * @param   i           the index (parent) for which to find the min-grandchild
00130      * @param   right-index the index of the right-most element that is part of the heap
00131      * @return  a pair where the first element is `true` if `i` has grandchildren
00132      *          (`false` otherwise), and the second element is the index of the
00133      *          grandchild whose value is smallest (only if the first element is `true`)
00134      */
00135     std::pair<bool, size_t> min_gchild(int*
      heap_array, size_t i, size_t right_index){
00136         std::pair<bool, size_t> result{false, 0};
00137         auto l = left(i);
00138         auto r = right(i);
00139         if(left(l) <= right_index){
00140             auto m = left(l);
00141             if(right(l) <= right_index && heap_array[right(l)] < heap_array[m]){
00142                 m = right(l);
00143             }
00144             if(left(r) <= right_index && heap_array[left(r)] < heap_array[m]){
00145                 m = left(r);
00146             }
00147             if(right(r) <= right_index && heap_array[right(r)] < heap_array[m]){
00148                 m = right(r);
00149             }
00150             result = {true, m};
00151         }
00152         return result;
00153     }
00154
00155     /**
00156      * get a pair considing of an indication of whether `i` has any children, and
00157      * if so, the index of the child or grandchild containing the minimum value.
00158      *
00159      * @param   heap_array  the heap
00160      * @param   i           the index (parent) for which to find the min-(grand)child
00161      * @param   right-index the index of the right-most element that is part of the heap
00162      * @return  a pair where the first element is `true` if `i` has children
```

```
00163      *          (`false` otherwise), and the second element is the index of the
00164      *          child or grandchild whose value is smallest (only if the first
00165      *          element is `true`)
00166      */
00167     std::pair<bool, size_t> min_child_or_gchild(int*
      heap_array, size_t i, size_t right_index){
00168         auto m = min_child(heap_array, i, right_index);
00169         if(m.first){
00170             auto  gm = min_gchild(heap_array, i, right_index);
00171             m.second = gm.first && heap_array[gm.second] < heap_array[m.second] ? gm.second : m.second
      ;
00172         }
00173         return m;
00174     }
00175
00176     /**
00177      * get a pair considing of an indication of whether `i` has any children, and
00178      * if so, the index of the child containing the maximum value.
00179      *
00180      * @param   heap_array  the heap
00181      * @param   i           the index (parent) for which to find the max-child
00182      * @param   right-index the index of the right-most element that is part of the heap
00183      * @return  a pair where the first element is `true` if `i` has children (`false`
00184      *          otherwise), and the second element is the index of the child whose value
00185      *          is largest (only if the first element is `true`)
00186      */
00187     std::pair<bool, size_t> max_child(int*
      heap_array, size_t i, size_t right_index){
00188         std::pair<bool, size_t> result {false, 0};
00189         if(left(i) <= right_index){
00190             auto m = left(i);
00191             if(right(i) <= right_index && heap_array[right(i)] > heap_array[m]){
00192                 m = right(i);
00193             }
00194             result = {true, m};
00195         }
00196         return result;
00197     }
00198
00199     /**
00200      * get a pair considing of an indication of whether `i` has any grandchildren, and
00201      * if so, the index of the grandchild containing the maximum value.
00202      *
00203      * @param   heap_array  the heap
00204      * @param   i           the index (parent) for which to find the max-grandchild
00205      * @param   right-index the index of the right-most element that is part of the heap
00206      * @return  a pair where the first element is `true` if `i` has grandchildren
00207      *          (`false` otherwise), and the second element is the index of the
00208      *          grandchild whose value is largest (only if the first element is `true`)
00209      */
00210
00211     std::pair<bool, size_t> max_gchild(int*
      heap_array, size_t i, size_t right_index){
00212         std::pair<bool, size_t> result{false, 0};
00213         auto l = left(i);
00214         auto r = right(i);
00215         if(left(l) <= right_index){
00216             auto m = left(l);
00217             if(right(l) <= right_index && heap_array[right(l)] > heap_array[m]){
00218                 m = right(l);
00219             }
00220             if(left(r) <= right_index && heap_array[left(r)] > heap_array[m]){
00221                 m = left(r);
00222             }
00223             if(right(r) <= right_index && heap_array[right(r)] > heap_array[m]){
00224                 m = right(r);
00225             }
00226             result = {true, m};
00227         }
00228         return result;
00229     }
00230
00231     /**
00232      * get a pair considing of an indication of whether `i` has any children, and
00233      * if so, the index of the child or grandchild containing the maximum value.
00234      *
00235      * @param   heap_array  the heap
00236      * @param   i           the index (parent) for which to find the max-(grand)child
00237      * @param   right-index the index of the right-most element that is part of the heap
00238      * @return  a pair where the first element is `true` if `i` has children
00239      *          (`false` otherwise), and the second element is the index of the
```

```
00240      *          child or grandchild whose value is largest (only if the first
00241      *          element is `true`)
00242      */
00243     std::pair<bool, size_t> max_child_or_gchild(int*
      heap_array, size_t i, size_t right_index){
00244         auto m = max_child(heap_array, i, right_index);
00245         if(m.first){
00246             auto gm  = max_gchild(heap_array, i, right_index);
00247             m.second = gm.first && heap_array[gm.second] > heap_array[m.second] ? gm.second : m.second
      ;
00248         }
00249         return m;
00250     }
00251
00252     /**
00253      * perform min-max heap sift-down on an element (at `sift_index`) that is on a min-level
00254      *
00255      * @param heap_array  the heap
00256      * @param sift_index  the index of the element that should be sifted down
00257      * @param right_index the index of the right-most element that is part of the heap
00258      */
00259     void sift_down_min(int* heap_array, size_t sift_index, size_t right_index){
00260         bool sift_more = true;
00261         while(sift_more && left(sift_index) <= right_index){                          // if a[i] has
      children
00262             sift_more = false;
00263             auto mp = min_child_or_gchild(heap_array, sift_index, right_index);      // get min
      child or grandchild
00264             auto m  = mp.second;
00265             if(child(sift_index, m)){                                                // if the min
      was a child
00266                 if(heap_array[m] < heap_array[sift_index]){
00267                     std::swap(heap_array[m], heap_array[sift_index]);
00268                 }
00269             }
00270             else{                                                                    // min was a
      grandchild
00271                 if(heap_array[m] < heap_array[sift_index]){
00272                     std::swap(heap_array[m], heap_array[sift_index]);
00273                     if(heap_array[m] > heap_array[parent(m)]){
00274                         std::swap(heap_array[m], heap_array[parent(m)]);
00275                     }
00276                     sift_index = m;
00277                     sift_more  = true;
00278                 }
00279             }
00280         }
00281     }
00282
00283     /**
00284      * perform min-max heap sift-down on an element (at `sift_index`) that is on a max-level
00285      *
00286      * @param heap_array  the heap
00287      * @param sift_index  the index of the element that should be sifted down
00288      * @param right_index the index of the right-most element that is part of the heap
00289      */
00290     void sift_down_max(int* heap_array, size_t sift_index, size_t right_index){
00291         bool sift_more = true;
00292         while(sift_more && left(sift_index) <= right_index){                          // if a[i] has
      children
00293             sift_more = false;
00294             auto mp = max_child_or_gchild(heap_array, sift_index, right_index);      // get max
      child or grandchild
00295             auto m  = mp.second;
00296             if(child(sift_index, m)){                                                // if the max
      was a child
00297                 if(heap_array[m] > heap_array[sift_index]){
00298                     std::swap(heap_array[m], heap_array[sift_index]);
00299                 }
00300             }
00301             else{                                                                    // max was a
      grandchild
00302                 if(heap_array[m] > heap_array[sift_index]){
00303                     std::swap(heap_array[m], heap_array[sift_index]);
00304                     if(heap_array[m] < heap_array[parent(m)]){
00305                         std::swap(heap_array[m], heap_array[parent(m)]);
00306                     }
00307                     sift_index = m;
00308                     sift_more  = true;
00309                 }
00310             }
```

```
00311         }
00312     }
00313
00314     /**
00315      * perform min-max heap sift-down on an element (at `sift_index`)
00316      *
00317      * @param heap_array  the heap
00318      * @param sift_index  the index of the element that should be sifted down
00319      * @param right_index the index of the right-most element that is part of the heap
00320      */
00321     void sift_down(int* heap_array, size_t sift_index, size_t right_index){
00322         if(min_level(sift_index)){
00323             sift_down_min(heap_array, sift_index, right_index);
00324         }
00325         else{
00326             sift_down_max(heap_array, sift_index, right_index);
00327         }
00328     }
00329
00330     /**
00331      * perform min-max heap bubble-up on an element (at `bubble_index`) that is on a min-level
00332      *
00333      * @param heap_array    the heap
00334      * @param bubble_index  the index of the element that should be bubbled up
00335      */
00336     void bubble_up_min(int* heap_array, size_t bubble_index){
00337         bool finished = false;
00338         while(!finished && has_gparent(bubble_index)){
00339             finished = true;
00340             if(heap_array[bubble_index] < heap_array[gparent(bubble_index)]){
00341                 std::swap(heap_array[bubble_index], heap_array[gparent(bubble_index)]);
00342                 bubble_index    = gparent(bubble_index);
00343                 finished = false;
00344             }
00345         }
00346     }
00347
00348     /**
00349      * perform min-max heap bubble-up on an element (at `bubble_index`) that is on a max-level
00350      *
00351      * @param heap_array    the heap
00352      * @param bubble_index  the index of the element that should be bubbled up
00353      */
00354     void bubble_up_max(int* heap_array, int bubble_index){
00355         bool finished = false;
00356         while(!finished && has_gparent(bubble_index)){
00357             finished = true;
00358             if(heap_array[bubble_index] > heap_array[gparent(bubble_index
      )]){
00359                 std::swap(heap_array[bubble_index], heap_array[gparent
      (bubble_index)]);
00360                 bubble_index    = gparent(bubble_index);
00361                 finished = false;
00362             }
00363         }
00364     }
00365
00366     /**
00367      * perform min-max heap bubble-up on an element (at `bubble_index`)
00368      *
00369      * @param heap_array    the heap
00370      * @param bubble_index  the index of the element that should be bubbled up
00371      */
00372     void bubble_up(int* heap_array, int bubble_index){
00373         if(min_level(bubble_index)){
00374             if(has_parent(bubble_index) && heap_array[bubble_index] > heap_array[
      parent(bubble_index)]){
00375                 std::swap(heap_array[bubble_index], heap_array[parent
      (bubble_index)]);
00376                 bubble_up_max(heap_array, parent(bubble_index
      ));
00377             }
00378             else{
00379                 bubble_up_min(heap_array, bubble_index);
00380             }
00381         }
00382         else{
00383             if(has_parent(bubble_index) && heap_array[bubble_index] < heap_array[
      parent(bubble_index)]){
00384                 std::swap(heap_array[bubble_index], heap_array[parent
      (bubble_index)]);
```

```
00385                 bubble_up_min(heap_array, parent(bubble_index
      ));
00386             }
00387             else{
00388                 bubble_up_max(heap_array, bubble_index);
00389             }
00390         }
00391     }
00392 }
00393
00394 /**
00395  * The `mmheap` namespace defines functions that are useful for building and
00396  * maintaining a Min-Max heap.  All necessary ("public-facing") functionality
00397  * is in this namespace.
00398  */
00399 namespace mmheap{
00400     /**
00401      * @brief   make an arbitrary array into a heap (in-place)
00402      * @details Applies Floyd's algorithm (adapted to a min-max heap) to produce
00403      *          a heap from an arbitrary array in linear time.
00404      *
00405      * @param heap_array    the array that will become a heap
00406      * @param size          the number of elements in the array
00407      */
00408     void make_heap(int* heap_array, size_t size){
00409         if(size > 1){
00410             for(int current = _mmheap::parent(size-1); current >= 0; --current){
00411                 _mmheap::sift_down(heap_array, current, size-1);
00412             }
00413         }
00414     }
00415
00416     /**
00417      * insert a new value to the heap (and update the `count`)
00418      *
00419      * @param           value       the new value to insert
00420      * @param           heap_array  the heap
00421      * @param[in,out]   count       the current number of items in the heap (will update)
00422      * @param           max_size    the physical storage allocation size of the heap
00423      * @throws std::runtime_error if the heap is full prior to the insert operation
00424      */
00425     void heap_insert(int value, int* heap_array, size_t& count, size_t max_size){
00426         if(count < max_size){
00427             heap_array[count++] = value;
00428             _mmheap::bubble_up(heap_array, count-1);
00429         }
00430         else{
00431             throw std::runtime_error("Cannot insert into heap - allocated size is full.");
00432         }
00433     }
00434
00435     /**
00436      * get the maximum value in the heap
00437      *
00438      * @param heap_array the heap
00439      * @param count      the current number of values contained in the heap
00440      *
00441      * @return the maximum value in the heap
00442      * @throws std::runtime_error if the heap is empty
00443      */
00444     int heap_max(int* heap_array, size_t count){
00445         if(count < 1){
00446             throw std::runtime_error("Cannot get max value in empty heap.");
00447         }
00448         auto m = _mmheap::max_child(heap_array, 0, count-1);
00449         return m.first ? heap_array[m.second] : heap_array[0];
00450     }
00451
00452     /**
00453      * get the minimum value in the heap
00454      *
00455      * @param heap_array the heap
00456      * @param count      the current number of values contained in the heap
00457      *
00458      * @return the minimum value in the heap
00459      * @throws std::runtime_error if the heap is empty
00460      */
00461     int heap_min(int* heap_array, size_t count){
00462         if(count < 1){
00463             throw std::runtime_error("Cannot get min value in empty heap.");
00464         }
```

```
00465         return heap_array[0];
00466     }
00467
00468     /**
00469      * @brief   add to heap, rotating the maximum value out if the heap is full
00470      * @details Add to the min-max heap in such a way that the maximum value is removed
00471      *          at the same time if the heap has reached its storage capacity.
00472      *
00473      * @param         value         new value to add
00474      * @param         heap_array    the heap
00475      * @param[in,out] count         number of values currently in the heap (will update)
00476      * @param         max_size      maximum physical size allocated for the heap
00477      *
00478      * @return a pair consising of a flag and a value; the first element is a flag
00479      *         indicating that overflow occurred, and the second element is the value
00480      *         that rotated out of the heap (formerly the maximum) when the new value
00481      *         was added (set only if an overflow occurred)
00482      */
00483     std::pair<bool, int> heap_insert_circular(int value, int* heap_array,
      size_t& count, size_t max_size){
00484         int max_value = 0;
00485         bool overflowed  = count == max_size ? true : false;
00486         if(!overflowed){
00487             heap_insert(value, heap_array, count, max_size);
00488         }
00489         else{                                                    // if the heap is full, replace the
       max value with the new add...
00490             auto m         = max_size > 1 ? _mmheap::max_child(heap_array, 0, max_size-1).second : 0;
00491             max_value      = heap_array[m];
00492             if(value < max_value){                               // if the new value is larger than
       the one rotating out, just rotate the new value
00493                 heap_array[m] = value;
00494                 if(max_size > 1){                                // if this is non-trivial
00495                     if(value < heap_array[0]){                   // check that the new value isn't
       the new min
00496                         std::swap(heap_array[0], heap_array[m]);    //  (if it is, make it so)
00497                     }
00498                     _mmheap::sift_down(heap_array, m, max_size-1);  // sift the new item down
00499                 }
00500             }
00501             else{
00502                 max_value = value;
00503             }
00504         }
00505         return std::pair<bool, int>{overflowed, max_value};
00506     }
00507
00508
00509     /**
00510      * replace and return the value at a given index with a new value
00511      *
00512      * @param new_value   new value to insert
00513      * @param index       index of the value to replace
00514      * @param heap_array  the heap
00515      * @param count       number of values currently stored in the heap
00516      *
00517      * @return  the old value being replaced
00518      * @throws  std::runtime_error if the heap is empty
00519      * @throws  std::range_error   if the index is out of range
00520      */
00521     int heap_replace_at_index(int new_value, size_t index, int* heap_array,
      size_t count){
00522         if(count == 0){
00523             throw std::runtime_error("Cannot replace value in empty heap.");
00524         }
00525         if(index > count){
00526             throw std::range_error("Index beyond end of heap.");
00527         }
00528         int old_value     = heap_array[index];
00529         heap_array[index] = new_value;
00530         if(_mmheap::min_level(index)){
00531             if(new_value < old_value){
00532                 _mmheap::bubble_up_min(heap_array, index);
00533             }
00534             else{
00535                 if(_mmheap::has_parent(index) && heap_array[_mmheap::parent(index)] < new_value){
00536                     _mmheap::bubble_up(heap_array, index);
00537                 }
00538                 _mmheap::sift_down(heap_array, index, count-1);
00539             }
00540         }
```

```
00541         else{
00542             if(new_value > old_value){
00543                 _mmheap::bubble_up_max(heap_array, index);
00544             }
00545             else{
00546                 if(_mmheap::has_parent(index) && new_value < heap_array[_mmheap::parent(index)]){
00547                     _mmheap::bubble_up(heap_array, index);
00548                 }
00549                 _mmheap::sift_down(heap_array, index, count-1);
00550             }
00551         }
00552         return old_value;
00553     }
00554 
00555     /**
00556      * remove and return value at a given index
00557      *
00558      * @param         index      index to remove
00559      * @param         heap_array the heap
00560      * @param[in,out] count      current number of values in the heap (will update)
00561      *
00562      * @return  the value being removed
00563      * @throws  std::runtime_error if the heap is empty
00564      * @throws  std::range_error   if the index is out of range
00565      */
00566     int heap_remove_at_index(size_t index, int* heap_array, size_t& count){
00567         if(count == 0){
00568             throw std::runtime_error("Cannot remove value in empty heap.");
00569         }
00570         if(index > count){
00571             throw std::range_error("Index beyond end of heap.");
00572         }
00573         int old_value = heap_replace_at_index(heap_array[count-1], index, heap_array, count);
00574         --count;
00575         return old_value;
00576     }
00577 
00578     /**
00579      * remove and return the minimum value in the heap
00580      *
00581      * @param heap_array the array
00582      * @param count      the current number of values in the heap (will update)
00583      *
00584      * @return the minimum value in the heap
00585      * @throws std::runtime_error if the heap is empty
00586      */
00587     int heap_remove_min(int* heap_array, size_t& count){
00588         if(count == 0){
00589             throw std::runtime_error("Cannot remove from empty heap.");
00590         }
00591         int value = heap_array[0];
00592         std::swap(heap_array[0], heap_array[count-1]);
00593         --count;
00594         if(count > 0){
00595             _mmheap::sift_down(heap_array, 0, count-1);
00596         }
00597         return value;
00598     }
00599 
00600     /**
00601      * remove and return the maximum value in the heap
00602      *
00603      * @param heap_array the array
00604      * @param count      the current number of values in the heap (will update)
00605      *
00606      * @return the maximum value in the heap
00607      * @throws std::runtime_error if the heap is empty
00608      */
00609     int heap_remove_max(int* heap_array, size_t& count){
00610         if(count == 0){
00611             throw std::runtime_error("Cannot remove from empty heap.");
00612         }
00613         auto value = heap_array[0];
00614         auto m     = _mmheap::max_child(heap_array, 0, count-1);
00615         if(m.first){
00616             value = heap_array[m.second];
00617         }
00618         else{
00619             m.second = 0;
00620         }
00621         heap_remove_at_index(m.second, heap_array, count);
```

```
00622         return value;
00623     }
00624 }
00625
00626 #endif
```

Index