Min-Max Heap

Generated by Doxygen 1.8.10

Wed Dec 9 2015 11:48:35

Contents

1 File Documentation

1.1 mmheap.h File Reference

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

Include dependency graph for mmheap.h:

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 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 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)
- void make_mm_heap (int ∗heap_array, size_t size)

  make an arbitrary array into a heap (in-place)
- void mm_heap_add (int value, int ∗heap_array, size_t &count, size_t max_size)
- int mm_heap_max (int ∗heap_array, size_t count)

- int mm_heap_min (int ∗heap_array, size_t count)
- std::pair< bool, int > mm_heap_ripple_add (int value, int ∗heap_array, size_t &count, size_t max_size)

    add to heap, pushing the maximum value out if the heap is full
- int mm_heap_replace_at_index (int new_value, size_t index, int ∗heap_array, size_t count)
- int mm_heap_remove_at_index (size_t index, int ∗heap_array, size_t &count)
- int mm_heap_remove_min (int ∗heap_array, size_t &count)
- int mm_heap_remove_max (int ∗heap_array, size_t &count)

1.1.1 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`

Author

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

Copyright

Definition in file mmheap.h.

1.1.2 Function Documentation

1.1.2.1 void 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 356 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 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 338 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 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 320 of file mmheap.h.

Referenced by bubble_up().

Here is the caller graph for this function:

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

Definition at line 47 of file mmheap.h.

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

Definition at line 45 of file mmheap.h.

Referenced by bubble_up_max().

Here is the caller graph for this function:

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

Definition at line 46 of file mmheap.h.

Referenced by bubble_up_max().

Here is the caller graph for this function:

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

Definition at line 42 of file mmheap.h.

Referenced by bubble_up().

Here is the caller graph for this function:

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

Definition at line 43 of file mmheap.h.

1.1.2.9 uint64_t log_2 (uint64_t *i*)

Definition at line 55 of file mmheap.h.

1.1.2.10 void make_mm_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 385 of file mmheap.h.

1.1.2.11 std::pair<bool, size_t> 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 171 of file mmheap.h.

1.1.2.12 std::pair<bool, size_t> 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 227 of file mmheap.h.

1.1.2.13 std::pair<bool, size_t> 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 195 of file mmheap.h.

1.1.2.14 std::pair<bool, size_t> 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 96 of file mmheap.h.

1.1.2.15 std::pair<bool, size_t> 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 151 of file mmheap.h.

1.1.2.16 std::pair<bool, size_t> 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 119 of file mmheap.h.

1.1.2.17 bool 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 81 of file mmheap.h.

Referenced by bubble_up().

Here is the caller graph for this function:

1.1.2.18 void mm_heap_add (int *value,* int ∗ *heap_array,* size_t & *count,* size_t *max_size*)

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

Parameters

	value	the new value to add
	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 add operation

Definition at line 402 of file mmheap.h.

1.1.2.19 int mm_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 421 of file mmheap.h.

1.1.2.20 int mm_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 438 of file mmheap.h.

1.1.2.21 int mm_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 538 of file mmheap.h.

1.1.2.22 int mm_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 579 of file mmheap.h.

1.1.2.23 int mm_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 559 of file mmheap.h.

1.1.2.24 int mm_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 493 of file mmheap.h.

1.1.2.25 std::pair<bool, int> mm_heap_ripple_add (int *value,* int ∗ *heap_array,* size_t & *count,* size_t *max_size*)

add to heap, pushing 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 shifted out of the heap (formerly the maximum) when the new value was added (set only if an overflow occurred)

Definition at line 460 of file mmheap.h.

1.1.2.26 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 305 of file mmheap.h.

1.1.2.27 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 274 of file mmheap.h.

1.1.2.28 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 243 of file mmheap.h.

1.1.2.29 size_t parent (size_t *i*) `[inline]`

Definition at line 41 of file mmheap.h.

Referenced by bubble_up().

Here is the caller graph for this function:

1.1.2.30 size_t right (size_t *i*) [inline]

Definition at line 44 of file mmheap.h.

1.2 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  * @author    Jason L Causey
00014  * @license   Released under the MIT License: http://opensource.org/licenses/MIT
00015  * @copyright Copyright (c) 2015 Jason L Causey, Arkansas State University
00016  *
00017  *   Permission is hereby granted, free of charge, to any person obtaining a copy
00018  *   of this software and associated documentation files (the "Software"), to deal
00019  *   in the Software without restriction, including without limitation the rights
00020  *   to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
00021  *   copies of the Software, and to permit persons to whom the Software is
00022  *   furnished to do so, subject to the following conditions:
00023  *
00024  *   The above copyright notice and this permission notice shall be included in
00025  *   all copies or substantial portions of the Software.
00026  *
00027  *   THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
00028  *   IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
00029  *   FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL THE
00030  *   AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
00031  *   LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
00032  *   OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
00033  *   THE SOFTWARE.
00034  */
00035
00036 #include <algorithm>
00037 #include <cmath>
00038 #include <stdexcept>
00039 #include <cassert>
00040
00041 inline size_t  parent(size_t i)          { assert(i > 0); return (i - 1) / 2;
            }
00042 inline size_t  has_parent(size_t i)      { return i > 0;
                }
00043 inline size_t  left   (size_t i)         { return 2*i + 1;
          }
00044 inline size_t  right  (size_t i)         { return 2*i + 2;
           }
00045 inline size_t  gparent(size_t i)         { assert(i > 2); return parent(
      parent(i)); }
00046 inline bool    has_gparent(size_t i)     { return i > 2;                          }
00047 inline bool    child(size_t i, size_t c) { return c == left(i) || c == right(i);    }
00048
00049 /*
00050  * fast log-base-2 based on code from:
00051  *     http://stackoverflow.com/a/11398748
00052  * @param  i value to compute the log_2 for (must be > 0)
00053  * @return log-base-2 of `i`
00054  */
00055 uint64_t log_2(uint64_t i) {
00056     static const uint64_t tab64[64] = {
00057         63,  0, 58,  1, 59, 47, 53,  2,
00058         60, 39, 48, 27, 54, 33, 42,  3,
00059         61, 51, 37, 40, 49, 18, 28, 20,
00060         55, 30, 34, 11, 43, 14, 22,  4,
00061         62, 57, 46, 52, 38, 26, 32, 41,
00062         50, 36, 17, 19, 29, 10, 13, 21,
00063         56, 45, 25, 31, 35, 16,  9, 12,
00064         44, 24, 15,  8, 23,  7,  6,  5
00065     };
00066     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
 * @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`)
 */
std::pair<bool, size_t> min_child(int* 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
 * @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`)
 */
std::pair<bool, size_t> min_gchild(int* 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
 * @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`)
 */
std::pair<bool, size_t> min_child_or_gchild(int* 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
 * @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`)
 */
std::pair<bool, size_t> max_child(int* 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 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
 * @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`)
 */

std::pair<bool, size_t> max_gchild(int* 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 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
 * @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`)
 */
std::pair<bool, size_t> max_child_or_gchild(int* 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[gm.second] > heap_array[m.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
 */
void mmheap_sift_down_min(int* 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[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`) 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
 */
void mmheap_sift_down_max(int* 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[m] > heap_array[sift_index]){
                std::swap(heap_array[m], heap_array[sift_index]);
            }
        }
        else{ // max was a grandchild
            if(heap_array[m] > heap_array[sift_index]){
                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
 */
void mmheap_sift_down(int* heap_array, size_t sift_index, size_t right_index){
    if(min_level(sift_index)){
        mmheap_sift_down_min(heap_array, sift_index, right_index);
    }
    else{
        mmheap_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
 */
void bubble_up_min(int* 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
 */
void bubble_up_max(int* heap_array, int 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')
 *
 * @param heap_array    the heap
 * @param bubble_index  the index of the element that should be bubbled up
 */
void bubble_up(int* heap_array, int bubble_index){
    if(min_level(bubble_index)){
        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_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);
        }
    }
}

/**
 * @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
 */
void make_mm_heap(int* heap_array, size_t size){
    if(size > 1){
        for(int current = parent(size-1); current >= 0; --current){
            mmheap_sift_down(heap_array, current, size-1);
        }
    }
}

/**
 * add a new value to the heap (and update the ‘count‘)
 *
 * @param           value       the new value to add
 * @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
 * @throws std::runtime_error if the heap is full prior to the add operation
 */
void mm_heap_add(int value, int* heap_array, size_t& count, size_t max_size){
    if(count < max_size){
        heap_array[count++] = value;
        bubble_up(heap_array, count-1);
    }
    else{
        throw std::runtime_error("Cannot add to 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
 *
 * @return the maximum value in the heap
 * @throws std::runtime_error if the heap is empty
 */
int mm_heap_max(int* heap_array, size_t count){
    if(count < 1){
        throw std::runtime_error("Cannot get max value in empty heap.");
    }
    auto m = 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
 *
 * @return the minimum value in the heap
 * @throws std::runtime_error if the heap is empty
 */
int mm_heap_min(int* 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, pushing 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
 *
 * @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 shifted out of the heap (formerly the maximum) when the new value
 *         was added (set only if an overflow occurred)
 */
std::pair<bool, int> mm_heap_ripple_add(int value, int* heap_array, size_t& count, size_t
max_size){
    int max_value = 0;
    bool rippled  = count == max_size ? true : false;
    if(!rippled){
        mm_heap_add(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 ? max_child(heap_array, 0, max_size-1).second : 0;
        max_value     = heap_array[m];
        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
        }
    }
    return std::pair<bool, int>{rippled, 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
 *
 * @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
 */
int mm_heap_replace_at_index(int new_value, size_t index, int* 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.");
    }
    int old_value     = heap_array[index];
    heap_array[index] = new_value;
    if(min_level(index)){
        if(new_value < old_value){
            bubble_up_min(heap_array, index);
        }
        else{
            if(has_parent(index) && heap_array[parent(index)] < new_value){
                std::swap(heap_array[parent(index)], heap_array[index]);
            }
            mmheap_sift_down_min(heap_array, index, count-1);
        }
    }
    else{
        if(new_value > old_value){
            bubble_up_max(heap_array, index);
        }
        else{
            if(has_parent(index) && new_value < heap_array[parent(index)]){
                std::swap(heap_array[parent(index)], heap_array[index]);
            }
            mmheap_sift_down_max(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)
 *
 * @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
 */
int mm_heap_remove_at_index(size_t index, int* 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.");
    }
    int old_value = mm_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)
 *
 * @return the minimum value in the heap
 * @throws std::runtime_error if the heap is empty
 */
int mm_heap_remove_min(int* heap_array, size_t& count){
    if(count == 0){
        throw std::runtime_error("Cannot remove from empty heap.");
    }
    int value = heap_array[0];
    std::swap(heap_array[0], heap_array[count-1]);
    --count;
    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)
 *
 * @return the maximum value in the heap
 * @throws std::runtime_error if the heap is empty
 */
int mm_heap_remove_max(int* heap_array, size_t& count){
    if(count == 0){
        throw std::runtime_error("Cannot remove from empty heap.");
    }
    auto value = heap_array[0];
    auto m     = max_child(heap_array, 0, count-1);
    if(m.first){
        value = m.second;
    }
    else{
        m.second = 0;
    }
    mm_heap_remove_at_index(m.second, heap_array, count);
    return value;
}

#endif
```

Index