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;
```

```
00067     i |= i >> 2;
00068     i |= i >> 4;
00069     i |= i >> 8;
00070     i |= i >> 16;
00071     i |= i >> 32;
00072     return tab64[((uint64_t)((i - (i >> 1))*0x07EDD5E59A4E28C2)) >> 58];
00073 }
00074
00075 /**
00076  * returns `true` if `i` is on a Min-Level
00077  *
00078  * @param   i index into the heap
00079  * @return  `true` if `i` is on a min-level
00080  */
00081 inline bool min_level(size_t i) {
00082     return i > 0 ? log_2(++i) % 2 == 0 : true;
00083 }
00084
00085 /**
00086  * get a pair considing of an indication of whether `i` has any children, and
00087  * if so, the index of the child containing the minimum value.
00088  *
00089  * @param   heap_array  the heap
00090  * @param   i           the index (parent) for which to find the min-child
00091  * @param   right-index the index of the right-most element that is part of the heap
00092  * @return  a pair where the first element is `true` if `i` has children (`false`
00093  *          otherwise), and the second element is the index of the child whose value
00094  *          is smallest (only if the first element is `true`)
00095  */
00096 std::pair<bool, size_t> min_child(int* heap_array, size_t i, size_t right_index){
00097     std::pair<bool, size_t> result{false, 0};
00098     if(left(i) <= right_index){
00099         auto m = left(i);
00100         if(right(i) <= right_index && heap_array[right(i)] < heap_array[m]){
00101             m = right(i);
00102         }
00103         result = {true, m};
00104     }
00105     return result;
00106 }
00107
00108 /**
00109  * get a pair considing of an indication of whether `i` has any grandchildren, and
00110  * if so, the index of the grandchild containing the minimum value.
00111  *
00112  * @param   heap_array  the heap
00113  * @param   i           the index (parent) for which to find the min-grandchild
00114  * @param   right-index the index of the right-most element that is part of the heap
00115  * @return  a pair where the first element is `true` if `i` has grandchildren
00116  *          (`false` otherwise), and the second element is the index of the
00117  *          grandchild whose value is smallest (only if the first element is `true`)
00118  */
00119 std::pair<bool, size_t> min_gchild(int* heap_array, size_t i, size_t right_index){
00120     std::pair<bool, size_t> result{false, 0};
00121     auto l = left(i);
00122     auto r = right(i);
00123     if(left(l) <= right_index){
00124         auto m = left(l);
00125         if(right(l) <= right_index && heap_array[right(l)] < heap_array[m]){
00126             m = right(l);
00127         }
00128         if(left(r) <= right_index && heap_array[left(r)] < heap_array[m]){
00129             m = left(r);
00130         }
00131         if(right(r) <= right_index && heap_array[right(r)] < heap_array[m]){
00132             m = right(r);
00133         }
00134         result = {true, m};
00135     }
00136     return result;
00137 }
00138
00139 /**
00140  * get a pair considing of an indication of whether `i` has any children, and
00141  * if so, the index of the child or grandchild containing the minimum value.
00142  *
00143  * @param   heap_array  the heap
00144  * @param   i           the index (parent) for which to find the min-(grand)child
00145  * @param   right-index the index of the right-most element that is part of the heap
00146  * @return  a pair where the first element is `true` if `i` has children
00147  *          (`false` otherwise), and the second element is the index of the
```

```
00148  *          child or grandchild whose value is smallest (only if the first
00149  *          element is `true`)
00150  */
00151 std::pair<bool, size_t> min_child_or_gchild(int* heap_array, size_t i, size_t right_index){
00152     auto m = min_child(heap_array, i, right_index);
00153     if(m.first){
00154         auto  gm = min_gchild(heap_array, i, right_index);
00155         m.second = gm.first && heap_array[gm.second] < heap_array[m.second] ? gm.second : m.second;
00156     }
00157     return m;
00158 }
00159
00160 /**
00161  * get a pair considing of an indication of whether `i` has any children, and
00162  * if so, the index of the child containing the maximum value.
00163  *
00164  * @param   heap_array  the heap
00165  * @param   i           the index (parent) for which to find the max-child
00166  * @param   right-index the index of the right-most element that is part of the heap
00167  * @return  a pair where the first element is `true` if `i` has children (`false`
00168  *          otherwise), and the second element is the index of the child whose value
00169  *          is largest (only if the first element is `true`)
00170  */
00171 std::pair<bool, size_t> max_child(int* heap_array, size_t i, size_t right_index){
00172     std::pair<bool, size_t> result {false, 0};
00173     if(left(i) <= right_index){
00174         auto m = left(i);
00175         if(right(i) <= right_index && heap_array[right(i)] > heap_array[m]){
00176             m = right(i);
00177         }
00178         result = {true, m};
00179     }
00180     return result;
00181 }
00182
00183 /**
00184  * get a pair considing of an indication of whether `i` has any grandchildren, and
00185  * if so, the index of the grandchild containing the maximum value.
00186  *
00187  * @param   heap_array  the heap
00188  * @param   i           the index (parent) for which to find the max-grandchild
00189  * @param   right-index the index of the right-most element that is part of the heap
00190  * @return  a pair where the first element is `true` if `i` has grandchildren
00191  *          (`false` otherwise), and the second element is the index of the
00192  *          grandchild whose value is largest (only if the first element is `true`)
00193  */
00194
00195 std::pair<bool, size_t> max_gchild(int* heap_array, size_t i, size_t right_index){
00196     std::pair<bool, size_t> result{false, 0};
00197     auto l = left(i);
00198     auto r = right(i);
00199     if(left(l) <= right_index){
00200         auto m = left(l);
00201         if(right(l) <= right_index && heap_array[right(l)] > heap_array[m]){
00202             m = right(l);
00203         }
00204         if(left(r) <= right_index && heap_array[left(r)] > heap_array[m]){
00205             m = left(r);
00206         }
00207         if(right(r) <= right_index && heap_array[right(r)] > heap_array[m]){
00208             m = right(r);
00209         }
00210         result = {true, m};
00211     }
00212     return result;
00213 }
00214
00215 /**
00216  * get a pair considing of an indication of whether `i` has any children, and
00217  * if so, the index of the child or grandchild containing the maximum value.
00218  *
00219  * @param   heap_array  the heap
00220  * @param   i           the index (parent) for which to find the max-(grand)child
00221  * @param   right-index the index of the right-most element that is part of the heap
00222  * @return  a pair where the first element is `true` if `i` has children
00223  *          (`false` otherwise), and the second element is the index of the
00224  *          child or grandchild whose value is largest (only if the first
00225  *          element is `true`)
00226  */
00227 std::pair<bool, size_t> max_child_or_gchild(int* heap_array, size_t i, size_t right_index){
00228     auto m = max_child(heap_array, i, right_index);
```

```
00229     if(m.first){
00230         auto gm  = max_gchild(heap_array, i, right_index);
00231         m.second = gm.first && heap_array[gm.second] > heap_array[m.second] ? gm.second : m.second;
00232     }
00233     return m;
00234 }
00235
00236 /**
00237  * perform min-max heap sift-down on an element (at 'sift_index') that is on a min-level
00238  *
00239  * @param heap_array  the heap
00240  * @param sift_index  the index of the element that should be sifted down
00241  * @param right_index the index of the right-most element that is part of the heap
00242  */
00243 void mmheap_sift_down_min(int* heap_array, size_t sift_index, size_t
      right_index){
00244     bool sift_more = true;
00245     while(sift_more && left(sift_index) <= right_index){                                    // if
       a[i] has children
00246         sift_more = false;
00247         auto mp = min_child_or_gchild(heap_array, sift_index, right_index); // get min child or
       grandchild
00248         auto m  = mp.second;
00249         if(child(sift_index, m)){  // if the min was a child
00250             if(heap_array[m] < heap_array[sift_index]){
00251                 std::swap(heap_array[m], heap_array[sift_index]);
00252             }
00253         }
00254         else{ // min was a grandchild
00255             if(heap_array[m] < heap_array[sift_index]){
00256                 std::swap(heap_array[m], heap_array[sift_index]);
00257                 if(heap_array[m] > heap_array[parent(m)]){
00258                     std::swap(heap_array[m], heap_array[parent(m)]);
00259                 }
00260                 sift_index = m;
00261                 sift_more  = true;
00262             }
00263         }
00264     }
00265 }
00266
00267 /**
00268  * perform min-max heap sift-down on an element (at 'sift_index') that is on a max-level
00269  *
00270  * @param heap_array  the heap
00271  * @param sift_index  the index of the element that should be sifted down
00272  * @param right_index the index of the right-most element that is part of the heap
00273  */
00274 void mmheap_sift_down_max(int* heap_array, size_t sift_index, size_t
      right_index){
00275     bool sift_more = true;
00276     while(sift_more && left(sift_index) <= right_index){                                    // if
       a[i] has children
00277         sift_more = false;
00278         auto mp = max_child_or_gchild(heap_array, sift_index, right_index); // get max child or
       grandchild
00279         auto m  = mp.second;
00280         if(child(sift_index, m)){  // if the max was a child
00281             if(heap_array[m] > heap_array[sift_index]){
00282                 std::swap(heap_array[m], heap_array[sift_index]);
00283             }
00284         }
00285         else{ // max was a grandchild
00286             if(heap_array[m] > heap_array[sift_index]){
00287                 std::swap(heap_array[m], heap_array[sift_index]);
00288                 if(heap_array[m] < heap_array[parent(m)]){
00289                     std::swap(heap_array[m], heap_array[parent(m)]);
00290                 }
00291                 sift_index = m;
00292                 sift_more  = true;
00293             }
00294         }
00295     }
00296 }
00297
00298 /**
00299  * perform min-max heap sift-down on an element (at 'sift_index')
00300  *
00301  * @param heap_array  the heap
00302  * @param sift_index  the index of the element that should be sifted down
00303  * @param right_index the index of the right-most element that is part of the heap
```

```
00304  */
00305 void mmheap_sift_down(int* heap_array, size_t sift_index, size_t right_index){
00306     if(min_level(sift_index)){
00307         mmheap_sift_down_min(heap_array, sift_index, right_index);
00308     }
00309     else{
00310         mmheap_sift_down_max(heap_array, sift_index, right_index);
00311     }
00312 }
00313
00314 /**
00315  * perform min-max heap bubble-up on an element (at `bubble_index`) that is on a min-level
00316  *
00317  * @param heap_array    the heap
00318  * @param bubble_index  the index of the element that should be bubbled up
00319  */
00320 void bubble_up_min(int* heap_array, size_t bubble_index){
00321     bool finished = false;
00322     while(!finished && has_gparent(bubble_index)){
00323         finished = true;
00324         if(heap_array[bubble_index] < heap_array[gparent(bubble_index)]){
00325             std::swap(heap_array[bubble_index], heap_array[gparent(bubble_index)]);
00326             bubble_index    = gparent(bubble_index);
00327             finished = false;
00328         }
00329     }
00330 }
00331
00332 /**
00333  * perform min-max heap bubble-up on an element (at `bubble_index`) that is on a max-level
00334  *
00335  * @param heap_array    the heap
00336  * @param bubble_index  the index of the element that should be bubbled up
00337  */
00338 void bubble_up_max(int* heap_array, int bubble_index){
00339     bool finished = false;
00340     while(!finished && has_gparent(bubble_index)){
00341         finished = true;
00342         if(heap_array[bubble_index] > heap_array[gparent(bubble_index
      )]){
00343             std::swap(heap_array[bubble_index], heap_array[gparent(bubble_index)]);
00344             bubble_index    = gparent(bubble_index);
00345             finished = false;
00346         }
00347     }
00348 }
00349
00350 /**
00351  * perform min-max heap bubble-up on an element (at `bubble_index`)
00352  *
00353  * @param heap_array    the heap
00354  * @param bubble_index  the index of the element that should be bubbled up
00355  */
00356 void bubble_up(int* heap_array, int bubble_index){
00357     if(min_level(bubble_index)){
00358         if(has_parent(bubble_index) && heap_array[bubble_index] > heap_array[
      parent(bubble_index)]){
00359             std::swap(heap_array[bubble_index], heap_array[parent(bubble_index)]);
00360             bubble_up_max(heap_array, parent(bubble_index
      ));
00361         }
00362         else{
00363             bubble_up_min(heap_array, bubble_index);
00364         }
00365     }
00366     else{
00367         if(has_parent(bubble_index) && heap_array[bubble_index] < heap_array[
      parent(bubble_index)]){
00368             std::swap(heap_array[bubble_index], heap_array[parent(bubble_index)]);
00369             bubble_up_min(heap_array, parent(bubble_index
      ));
00370         }
00371         else{
00372             bubble_up_max(heap_array, bubble_index);
00373         }
00374     }
00375 }
00376
00377 /**
00378  * @brief   make an arbitrary array into a heap (in-place)
00379  * @details Applies Floyd's algorithm (adapted to a min-max heap) to produce
```

```
00380  *          a heap from an arbitrary array in linear time.
00381  *
00382  * @param heap_array    the array that will become a heap
00383  * @param size          the number of elements in the array
00384  */
00385 void make_mm_heap(int* heap_array, size_t size){
00386     if(size > 1){
00387         for(int current = parent(size-1); current >= 0; --current){
00388             mmheap_sift_down(heap_array, current, size-1);
00389         }
00390     }
00391 }
00392
00393 /**
00394  * add a new value to the heap (and update the `count`)
00395  *
00396  * @param           value       the new value to add
00397  * @param           heap_array  the heap
00398  * @param[in,out]   count       the current number of items in the heap (will update)
00399  * @param           max_size    the physical storage allocation size of the heap
00400  * @throws std::runtime_error if the heap is full prior to the add operation
00401  */
00402 void mm_heap_add(int value, int* heap_array, size_t& count, size_t max_size){
00403     if(count < max_size){
00404         heap_array[count++] = value;
00405         bubble_up(heap_array, count-1);
00406     }
00407     else{
00408         throw std::runtime_error("Cannot add to heap - allocated size is full.");
00409     }
00410 }
00411
00412 /**
00413  * get the maximum value in the heap
00414  *
00415  * @param heap_array the heap
00416  * @param count      the current number of values contained in the heap
00417  *
00418  * @return the maximum value in the heap
00419  * @throws std::runtime_error if the heap is empty
00420  */
00421 int mm_heap_max(int* heap_array, size_t count){
00422     if(count < 1){
00423         throw std::runtime_error("Cannot get max value in empty heap.");
00424     }
00425     auto m = max_child(heap_array, 0, count-1);
00426     return m.first ? heap_array[m.second] : heap_array[0];
00427 }
00428
00429 /**
00430  * get the minimum value in the heap
00431  *
00432  * @param heap_array the heap
00433  * @param count      the current number of values contained in the heap
00434  *
00435  * @return the minimum value in the heap
00436  * @throws std::runtime_error if the heap is empty
00437  */
00438 int mm_heap_min(int* heap_array, size_t count){
00439     if(count < 1){
00440         throw std::runtime_error("Cannot get min value in empty heap.");
00441     }
00442     return heap_array[0];
00443 }
00444
00445 /**
00446  * @brief   add to heap, pushing the maximum value out if the heap is full
00447  * @details Add to the min-max heap in such a way that the maximum value is removed
00448  *          at the same time if the heap has reached its storage capacity.
00449  *
00450  * @param         value         new value to add
00451  * @param         heap_array    the heap
00452  * @param[in,out] count         number of values currently in the heap (will update)
00453  * @param         max_size      maximum physical size allocated for the heap
00454  *
00455  * @return a pair consising of a flag and a value; the first element is a flag
00456  *         indicating that overflow occurred, and the second element is the value
00457  *         that shifted out of the heap (formerly the maximum) when the new value
00458  *         was added (set only if an overflow occurred)
00459  */
00460 std::pair<bool, int> mm_heap_ripple_add(int value, int* heap_array, size_t& count, size_t
```

```
    max_size){
00461     int max_value = 0;
00462     bool rippled  = count == max_size ? true : false;
00463     if(!rippled){
00464         mm_heap_add(value, heap_array, count, max_size);
00465     }
00466     else{       // if the heap is full, replace the max value with the new add...
00467         auto m         = max_size > 1 ? max_child(heap_array, 0, max_size-1).second : 0;
00468         max_value      = heap_array[m];
00469         heap_array[m] = value;
00470         if(max_size > 1){                                   // if this is non-trivial
00471             if(value < heap_array[0]){                      // check that the new value isn't the new
     min
00472                 std::swap(heap_array[0], heap_array[m]);    //  (if it is, make it so)
00473             }
00474             mmheap_sift_down(heap_array, m, max_size-1);    // sift the new item down
00475         }
00476     }
00477     return std::pair<bool, int>{rippled, max_value};
00478 }
00479
00480
00481 /**
00482  * replace and return the value at a given index with a new value
00483  *
00484  * @param new_value   new value to insert
00485  * @param index       index of the value to replace
00486  * @param heap_array  the heap
00487  * @param count       number of values currently stored in the heap
00488  *
00489  * @return  the old value being replaced
00490  * @throws  std::runtime_error if the heap is empty
00491  * @throws  std::range_error   if the index is out of range
00492  */
00493 int mm_heap_replace_at_index(int new_value, size_t index, int* heap_array
    , size_t count){
00494     if(count == 0){
00495         throw std::runtime_error("Cannot replace value in empty heap.");
00496     }
00497     if(index > count){
00498         throw std::range_error("Index beyond end of heap.");
00499     }
00500     int old_value     = heap_array[index];
00501     heap_array[index] = new_value;
00502     if(min_level(index)){
00503         if(new_value < old_value){
00504             bubble_up_min(heap_array, index);
00505         }
00506         else{
00507             if(has_parent(index) && heap_array[parent(index)] < new_value){
00508                 std::swap(heap_array[parent(index)], heap_array[index]);
00509             }
00510             mmheap_sift_down_min(heap_array, index, count-1);
00511         }
00512     }
00513     else{
00514         if(new_value > old_value){
00515             bubble_up_max(heap_array, index);
00516         }
00517         else{
00518             if(has_parent(index) && new_value < heap_array[parent(index)]){
00519                 std::swap(heap_array[parent(index)], heap_array[index]);
00520             }
00521             mmheap_sift_down_max(heap_array, index, count-1);
00522         }
00523     }
00524     return old_value;
00525 }
00526
00527 /**
00528  * remove and return value at a given index
00529  *
00530  * @param         index      index to remove
00531  * @param         heap_array the heap
00532  * @param[in,out] count      current number of values in the heap (will update)
00533  *
00534  * @return  the value being removed
00535  * @throws  std::runtime_error if the heap is empty
00536  * @throws  std::range_error   if the index is out of range
00537  */
00538 int mm_heap_remove_at_index(size_t index, int* heap_array, size_t& count){
```

```
00539     if(count == 0){
00540         throw std::runtime_error("Cannot remove value in empty heap.");
00541     }
00542     if(index > count){
00543         throw std::range_error("Index beyond end of heap.");
00544     }
00545     int old_value = mm_heap_replace_at_index(heap_array[count-1], index, heap_array, count);
00546     --count;
00547     return old_value;
00548 }
00549
00550 /**
00551  * remove and return the minimum value in the heap
00552  *
00553  * @param heap_array the array
00554  * @param count      the current number of values in the heap (will update)
00555  *
00556  * @return the minimum value in the heap
00557  * @throws std::runtime_error if the heap is empty
00558  */
00559 int mm_heap_remove_min(int* heap_array, size_t& count){
00560     if(count == 0){
00561         throw std::runtime_error("Cannot remove from empty heap.");
00562     }
00563     int value = heap_array[0];
00564     std::swap(heap_array[0], heap_array[count-1]);
00565     --count;
00566     mmheap_sift_down(heap_array, 0, count-1);
00567     return value;
00568 }
00569
00570 /**
00571  * remove and return the maximum value in the heap
00572  *
00573  * @param heap_array the array
00574  * @param count      the current number of values in the heap (will update)
00575  *
00576  * @return the maximum value in the heap
00577  * @throws std::runtime_error if the heap is empty
00578  */
00579 int mm_heap_remove_max(int* heap_array, size_t& count){
00580     if(count == 0){
00581         throw std::runtime_error("Cannot remove from empty heap.");
00582     }
00583     auto value = heap_array[0];
00584     auto m     = max_child(heap_array, 0, count-1);
00585     if(m.first){
00586         value = m.second;
00587     }
00588     else{
00589         m.second = 0;
00590     }
00591     mm_heap_remove_at_index(m.second, heap_array, count);
00592     return value;
00593 }
00594
00595 #endif
```

Index