Min-Max Heap

Generated by Doxygen 1.8.9.1

Sun Dec 20 2015 19:58:44

Contents

1 Namespace Documentation

1.1 _mmheap Namespace Reference

Functions

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

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

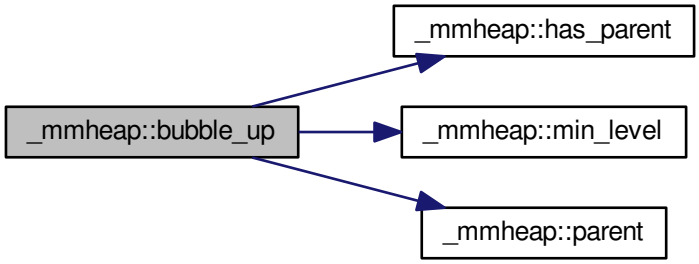
Template Parameters

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

Definition at line 419 of file mmheap.h.

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

Here is the call graph for this function:



1.1.2.2 template<typename DataType > void _mmheap::bubble_up_max (DataType * *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

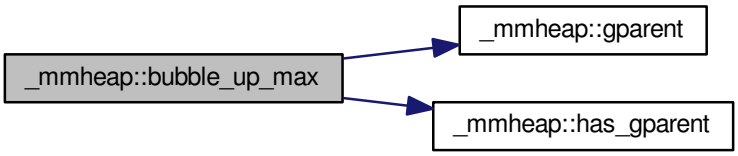
Template Parameters

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

Definition at line 397 of file mmheap.h.

References gparent(), and has_gparent().

Here is the call graph for this function:



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

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

Parameters

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

Template Parameters

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

Definition at line 375 of file mmheap.h.

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

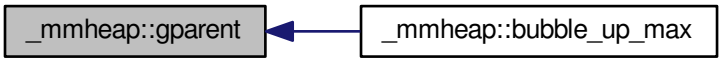
Definition at line 63 of file mmheap.h.

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

Definition at line 61 of file mmheap.h.

Referenced by bubble_up_max().

Here is the caller graph for this function:

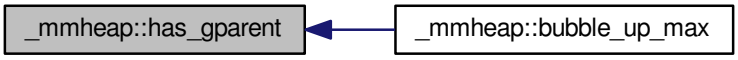


1.1.2.6 bool _mmheap::has_gparent (size_t *i*) [inline]

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

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

Parameters

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

Template Parameters

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

Returns

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

Definition at line 203 of file mmheap.h.

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

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

Parameters

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

Template Parameters

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

Returns

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

Definition at line 266 of file mmheap.h.

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

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

Parameters

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

Template Parameters

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

Returns

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

Definition at line 230 of file mmheap.h.

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

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

Parameters

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

Template Parameters

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

Returns

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

Definition at line 116 of file mmheap.h.

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

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

Parameters

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

Template Parameters

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

Returns

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

Definition at line 179 of file mmheap.h.

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

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

Parameters

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

Template Parameters

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

Returns

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

Definition at line 143 of file mmheap.h.

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

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

Parameters

i	index into the heap

Returns

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

Definition at line 97 of file mmheap.h.

Referenced by bubble_up().

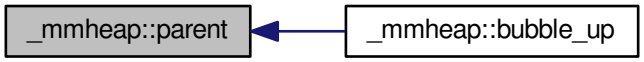
Here is the caller graph for this function:

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

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

Parameters

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

Template Parameters

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

Definition at line 356 of file mmheap.h.

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

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

Parameters

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

Template Parameters

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

Definition at line 321 of file mmheap.h.

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

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

Parameters

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

Template Parameters

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

Definition at line 286 of file mmheap.h.

1.2 mmheap Namespace Reference

Functions

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

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

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

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

1.2.1 Detailed Description

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

1.2.2 Function Documentation

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

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

Parameters

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

Template Parameters

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

Exceptions

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

Definition at line 480 of file mmheap.h.

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

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

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

Parameters

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

Template Parameters

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

Returns

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

Definition at line 547 of file mmheap.h.

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

get the maximum value in the heap

Parameters

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

Template Parameters

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

Returns

the maximum value in the heap

Exceptions

std::runtime_error	if the heap is empty

Definition at line 502 of file mmheap.h.

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

get the minimum value in the heap

Parameters

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

Template Parameters

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

Returns

the minimum value in the heap

Exceptions

std::runtime_error	if the heap is empty

Definition at line 522 of file mmheap.h.

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

remove and return value at a given index

Parameters

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

Template Parameters

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

Returns

the value being removed

Exceptions

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

Definition at line 636 of file mmheap.h.

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

remove and return the maximum value in the heap

Parameters

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

Template Parameters

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

Returns

the maximum value in the heap

Exceptions

std::runtime_error	if the heap is empty

Definition at line 685 of file mmheap.h.

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

remove and return the minimum value in the heap

Parameters

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

Template Parameters

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

Returns

the minimum value in the heap

Exceptions

std::runtime_error	if the heap is empty

Definition at line 660 of file mmheap.h.

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

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

Parameters

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

Template Parameters

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

Returns

the old value being replaced

Exceptions

std::runtime_error	if the heap is empty

std::range_error	if the index is out of range

Definition at line 588 of file mmheap.h.

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

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

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

Parameters

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

Template Parameters

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

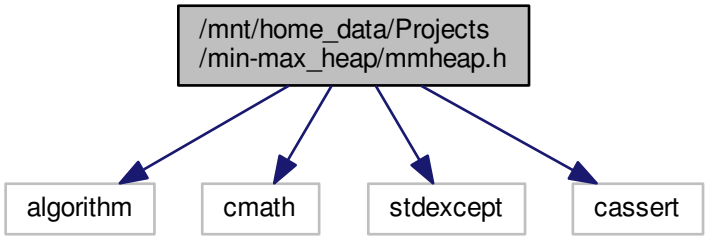
Definition at line 459 of file mmheap.h.

2 File Documentation

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

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

Include dependency graph for mmheap.h:



Namespaces

- _mmheap
- mmheap

Functions

- size_t _mmheap::parent (size_t i)
- size_t _mmheap::has_parent (size_t i)
- size_t _mmheap::left (size_t i)
- size_t _mmheap::right (size_t i)
- size_t _mmheap::gparent (size_t i)
- bool _mmheap::has_gparent (size_t i)
- bool _mmheap::child (size_t i, size_t c)
- uint64_t _mmheap::log_2 (uint64_t i)
- bool _mmheap::min_level (size_t i)
- template<typename DataType >
  std::pair< bool, size_t > _mmheap::min_child (DataType *heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > _mmheap::min_gchild (DataType *heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > _mmheap::min_child_or_gchild (DataType *heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > _mmheap::max_child (DataType *heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > _mmheap::max_gchild (DataType *heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > _mmheap::max_child_or_gchild (DataType *heap_array, size_t i, size_t right_index)
- template<typename DataType >
  void _mmheap::sift_down_min (DataType *heap_array, size_t sift_index, size_t right_index)
- template<typename DataType >
  void _mmheap::sift_down_max (DataType *heap_array, size_t sift_index, size_t right_index)
- template<typename DataType >
  void _mmheap::sift_down (DataType *heap_array, size_t sift_index, size_t right_index)
- template<typename DataType >
  void _mmheap::bubble_up_min (DataType *heap_array, size_t bubble_index)
- template<typename DataType >
  void _mmheap::bubble_up_max (DataType *heap_array, int bubble_index)
- template<typename DataType >
  void _mmheap::bubble_up (DataType *heap_array, int bubble_index)
- template<typename DataType >
  void mmheap::make_heap (DataType *heap_array, size_t size)

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

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

- template<typename DataType >
  DataType mmheap::heap_remove_at_index (size_t index, DataType *heap_array, size_t &count)
- template<typename DataType >
  DataType mmheap::heap_remove_min (DataType *heap_array, size_t &count)
- template<typename DataType >
  DataType mmheap::heap_remove_max (DataType *heap_array, size_t &count)

2.1.1 Detailed Description

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

This file defines two namespaces:

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

Author

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

Copyright

Copyright (c) 2015 Jason L Causey, Arkansas State University

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUD↩ ING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR A↩ NY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

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      * @tparam  DataType    the type of data stored in the heap - must be
00109      *                      LessThanComparable, Swappable, CopyConstructable,
00110      *                      and CopyAssignable
00111      * @return  a pair where the first element is `true` if `i` has children (`false`
00112      *          otherwise), and the second element is the index of the child whose value
00113      *          is smallest (only if the first element is `true`)
00114      */
00115     template <typename DataType>
00116     std::pair<bool, size_t> min_child(DataType*
      heap_array, size_t i, size_t right_index){
00117         std::pair<bool, size_t> result{false, 0};
00118         if(left(i) <= right_index){
00119             auto m = left(i);
00120             if(right(i) <= right_index && heap_array[right(i)] < heap_array[m]){
00121                 m = right(i);
00122             }
00123             result = {true, m};
00124         }
00125         return result;
00126     }
00127
00128     /**
00129      * get a pair considing of an indication of whether `i` has any grandchildren, and
00130      * if so, the index of the grandchild containing the minimum value.
00131      *
00132      * @param   heap_array  the heap
00133      * @param   i           the index (parent) for which to find the min-grandchild
00134      * @param   right-index the index of the right-most element that is part of the heap
00135      * @tparam  DataType    the type of data stored in the heap - must be
00136      *                      LessThanComparable, Swappable, CopyConstructable,
00137      *                      and CopyAssignable
00138      * @return  a pair where the first element is `true` if `i` has grandchildren
00139      *          (`false` otherwise), and the second element is the index of the
00140      *          grandchild whose value is smallest (only if the first element is `true`)
00141      */
00142     template <typename DataType>
00143     std::pair<bool, size_t> min_gchild(DataType*
      heap_array, size_t i, size_t right_index){
00144         std::pair<bool, size_t> result{false, 0};
00145         auto l = left(i);
00146         auto r = right(i);
00147         if(left(l) <= right_index){
00148             auto m = left(l);
00149             if(right(l) <= right_index && heap_array[right(l)] < heap_array[m]){
00150                 m = right(l);
00151             }
00152             if(left(r) <= right_index && heap_array[left(r)] < heap_array[m]){
00153                 m = left(r);
00154             }
00155             if(right(r) <= right_index && heap_array[right(r)] < heap_array[m]){
00156                 m = right(r);
00157             }
00158             result = {true, m};
00159         }
00160         return result;
00161     }
00162
00163     /**
```

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

```
00240                 m = left(r);
00241             }
00242             if(right(r) <= right_index && heap_array[m] < heap_array[right(r)]){
00243                 m = right(r);
00244             }
00245             result = {true, m};
00246         }
00247         return result;
00248     }
00249 
00250     /**
00251      * get a pair considing of an indication of whether `i` has any children, and
00252      * if so, the index of the child or grandchild containing the maximum value.
00253      *
00254      * @param   heap_array  the heap
00255      * @param   i           the index (parent) for which to find the max-(grand)child
00256      * @param   right-index the index of the right-most element that is part of the heap
00257      * @tparam  DataType    the type of data stored in the heap - must be
00258      *                      LessThanComparable, Swappable, CopyConstructable,
00259      *                      and CopyAssignable
00260      * @return  a pair where the first element is `true` if `i` has children
00261      *          (`false` otherwise), and the second element is the index of the
00262      *          child or grandchild whose value is largest (only if the first
00263      *          element is `true`)
00264      */
00265     template <typename DataType>
00266     std::pair<bool, size_t> max_child_or_gchild(
      DataType* heap_array, size_t i, size_t
      right_index){
00267         auto m = max_child(heap_array, i, right_index);
00268         if(m.first){
00269             auto gm  = max_gchild(heap_array, i, right_index);
00270             m.second = gm.first &&  heap_array[m.second] < heap_array[gm.second] ? gm.second : m.
      second;
00271         }
00272         return m;
00273     }
00274 
00275     /**
00276      * perform min-max heap sift-down on an element (at `sift_index`) that is on a min-level
00277      *
00278      * @param heap_array  the heap
00279      * @param sift_index  the index of the element that should be sifted down
00280      * @param right_index the index of the right-most element that is part of the heap
00281      * @tparam  DataType    the type of data stored in the heap - must be
00282      *                      LessThanComparable, Swappable, CopyConstructable,
00283      *                      and CopyAssignable
00284      */
00285     template <typename DataType>
00286     void sift_down_min(DataType* heap_array, size_t sift_index, size_t right_index){
00287         bool sift_more = true;
00288         while(sift_more && left(sift_index) <= right_index){                          // if a[i] has
      children
00289             sift_more = false;
00290             auto mp = min_child_or_gchild(heap_array, sift_index, right_index);      // get min
      child or grandchild
00291             auto m  = mp.second;
00292             if(child(sift_index, m)){                                                // if the min
      was a child
00293                 if(heap_array[m] < heap_array[sift_index]){
00294                     std::swap(heap_array[m], heap_array[sift_index]);
00295                 }
00296             }
00297             else{                                                                    // min was a
      grandchild
00298                 if(heap_array[m] < heap_array[sift_index]){
00299                     std::swap(heap_array[m], heap_array[sift_index]);
00300                     if(heap_array[parent(m)] < heap_array[m]){
00301                         std::swap(heap_array[m], heap_array[parent(m)]);
00302                     }
00303                     sift_index = m;
00304                     sift_more  = true;
00305                 }
00306             }
00307         }
00308     }
00309 
00310     /**
00311      * perform min-max heap sift-down on an element (at `sift_index`) that is on a max-level
00312      *
00313      * @param heap_array  the heap
```

```
00314      * @param sift_index  the index of the element that should be sifted down
00315      * @param right_index the index of the right-most element that is part of the heap
00316      * @tparam  DataType    the type of data stored in the heap - must be
00317      *                      LessThanComparable, Swappable, CopyConstructable,
00318      *                      and CopyAssignable
00319      */
00320     template <typename DataType>
00321     void sift_down_max(DataType* heap_array, size_t sift_index, size_t right_index){
00322         bool sift_more = true;
00323         while(sift_more && left(sift_index) <= right_index){                          // if a[i] has
    children
00324             sift_more = false;
00325             auto mp = max_child_or_gchild(heap_array, sift_index, right_index);      // get max
    child or grandchild
00326             auto m  = mp.second;
00327             if(child(sift_index, m)){                                                 // if the max
    was a child
00328                 if(heap_array[sift_index] < heap_array[m]){
00329                     std::swap(heap_array[m], heap_array[sift_index]);
00330                 }
00331             }
00332             else{                                                                     // max was a
    grandchild
00333                 if(heap_array[sift_index] < heap_array[m]){
00334                     std::swap(heap_array[m], heap_array[sift_index]);
00335                     if(heap_array[m] < heap_array[parent(m)]){
00336                         std::swap(heap_array[m], heap_array[parent(m)]);
00337                     }
00338                     sift_index = m;
00339                     sift_more  = true;
00340                 }
00341             }
00342         }
00343     }
00344
00345     /**
00346      * perform min-max heap sift-down on an element (at `sift_index`)
00347      *
00348      * @param heap_array  the heap
00349      * @param sift_index  the index of the element that should be sifted down
00350      * @param right_index the index of the right-most element that is part of the heap
00351      * @tparam  DataType    the type of data stored in the heap - must be
00352      *                      LessThanComparable, Swappable, CopyConstructable,
00353      *                      and CopyAssignable
00354      */
00355     template <typename DataType>
00356     void sift_down(DataType* heap_array, size_t sift_index, size_t right_index){
00357         if(min_level(sift_index)){
00358             sift_down_min(heap_array, sift_index, right_index);
00359         }
00360         else{
00361             sift_down_max(heap_array, sift_index, right_index);
00362         }
00363     }
00364
00365     /**
00366      * perform min-max heap bubble-up on an element (at `bubble_index`) that is on a min-level
00367      *
00368      * @param heap_array    the heap
00369      * @param bubble_index  the index of the element that should be bubbled up
00370      * @tparam  DataType    the type of data stored in the heap - must be
00371      *                      LessThanComparable, Swappable, CopyConstructable,
00372      *                      and CopyAssignable
00373      */
00374     template <typename DataType>
00375     void bubble_up_min(DataType* heap_array, size_t bubble_index){
00376         bool finished = false;
00377         while(!finished && has_gparent(bubble_index)){
00378             finished = true;
00379             if(heap_array[bubble_index] < heap_array[gparent(bubble_index)]){
00380                 std::swap(heap_array[bubble_index], heap_array[gparent(bubble_index)]);
00381                 bubble_index = gparent(bubble_index);
00382                 finished     = false;
00383             }
00384         }
00385     }
00386
00387     /**
00388      * perform min-max heap bubble-up on an element (at `bubble_index`) that is on a max-level
00389      *
00390      * @param heap_array    the heap
```

```
00391      * @param bubble_index  the index of the element that should be bubbled up
00392      * @tparam  DataType    the type of data stored in the heap - must be
00393      *                      LessThanComparable, Swappable, CopyConstructable,
00394      *                      and CopyAssignable
00395      */
00396     template <typename DataType>
00397     void bubble_up_max(DataType* heap_array, int bubble_index){
00398         bool finished = false;
00399         while(!finished && has_gparent(bubble_index)){
00400             finished = true;
00401             if(heap_array[gparent(bubble_index)] < heap_array[bubble_index]){
00402                 std::swap(heap_array[bubble_index], heap_array[gparent
      (bubble_index)]);
00403                 bubble_index = gparent(bubble_index);
00404                 finished     = false;
00405             }
00406         }
00407     }
00408
00409     /**
00410      * perform min-max heap bubble-up on an element (at `bubble_index`)
00411      *
00412      * @param heap_array    the heap
00413      * @param bubble_index  the index of the element that should be bubbled up
00414      * @tparam  DataType    the type of data stored in the heap - must be
00415      *                      LessThanComparable, Swappable, CopyConstructable,
00416      *                      and CopyAssignable
00417      */
00418     template <typename DataType>
00419     void bubble_up(DataType* heap_array, int bubble_index){
00420         if(min_level(bubble_index)){
00421             if(has_parent(bubble_index) && heap_array[parent
      (bubble_index)] < heap_array[bubble_index]){
00422                 std::swap(heap_array[bubble_index], heap_array[parent
      (bubble_index)]);
00423                 bubble_up_max(heap_array, parent(bubble_index));
00424             }
00425             else{
00426                 bubble_up_min(heap_array, bubble_index);
00427             }
00428         }
00429         else{
00430             if(has_parent(bubble_index) && heap_array[bubble_index] < heap_array[
      parent(bubble_index)]){
00431                 std::swap(heap_array[bubble_index], heap_array[parent
      (bubble_index)]);
00432                 bubble_up_min(heap_array, parent(bubble_index));
00433             }
00434             else{
00435                 bubble_up_max(heap_array, bubble_index);
00436             }
00437         }
00438     }
00439 }
00440
00441 /**
00442  * The `mmheap` namespace defines functions that are useful for building and
00443  * maintaining a Min-Max heap.  All necessary ("public-facing") functionality
00444  * is in this namespace.
00445  */
00446 namespace mmheap{
00447     /**
00448      * @brief   make an arbitrary array into a heap (in-place)
00449      * @details Applies Floyd's algorithm (adapted to a min-max heap) to produce
00450      *          a heap from an arbitrary array in linear time.
00451      *
00452      * @param heap_array    the array that will become a heap
00453      * @param size          the number of elements in the array
00454      * @tparam  DataType    the type of data stored in the heap - must be
00455      *                      LessThanComparable, Swappable, CopyConstructable,
00456      *                      and CopyAssignable
00457      */
00458     template <typename DataType>
00459     void make_heap(DataType* heap_array, size_t size){
00460         if(size > 1){
00461             for(int current = _mmheap::parent(size-1); current >= 0; --current){
00462                 _mmheap::sift_down(heap_array, current, size-1);
00463             }
00464         }
00465     }
00466
```

```
00467     /**
00468      * insert a new value to the heap (and update the `count`)
00469      *
00470      * @param           value       the new value to insert
00471      * @param           heap_array  the heap
00472      * @param[in,out]   count       the current number of items in the heap (will update)
00473      * @param           max_size    the physical storage allocation size of the heap
00474      * @tparam  DataType    the type of data stored in the heap - must be
00475      *                      LessThanComparable, Swappable, CopyConstructable,
00476      *                      and CopyAssignable
00477      * @throws std::runtime_error if the heap is full prior to the insert operation
00478      */
00479     template <typename DataType>
00480     void heap_insert(const DataType& value, DataType* heap_array, size_t& count,
      size_t max_size){
00481         if(count < max_size){
00482             heap_array[count++] = value;
00483             _mmheap::bubble_up(heap_array, count-1);
00484         }
00485         else{
00486             throw std::runtime_error("Cannot insert into heap - allocated size is full.");
00487         }
00488     }
00489
00490     /**
00491      * get the maximum value in the heap
00492      *
00493      * @param heap_array the heap
00494      * @param count      the current number of values contained in the heap
00495      * @tparam  DataType    the type of data stored in the heap - must be
00496      *                      LessThanComparable, Swappable, CopyConstructable,
00497      *                      and CopyAssignable
00498      * @return the maximum value in the heap
00499      * @throws std::runtime_error if the heap is empty
00500      */
00501     template <typename DataType>
00502     DataType heap_max(DataType* heap_array, size_t count){
00503         if(count < 1){
00504             throw std::runtime_error("Cannot get max value in empty heap.");
00505         }
00506         auto m = _mmheap::max_child(heap_array, 0, count-1);
00507         return m.first ? heap_array[m.second] : heap_array[0];
00508     }
00509
00510     /**
00511      * get the minimum value in the heap
00512      *
00513      * @param heap_array the heap
00514      * @param count      the current number of values contained in the heap
00515      * @tparam  DataType    the type of data stored in the heap - must be
00516      *                      LessThanComparable, Swappable, CopyConstructable,
00517      *                      and CopyAssignable
00518      * @return the minimum value in the heap
00519      * @throws std::runtime_error if the heap is empty
00520      */
00521     template <typename DataType>
00522     DataType heap_min(DataType* heap_array, size_t count){
00523         if(count < 1){
00524             throw std::runtime_error("Cannot get min value in empty heap.");
00525         }
00526         return heap_array[0];
00527     }
00528
00529     /**
00530      * @brief   add to heap, rotating the maximum value out if the heap is full
00531      * @details Add to the min-max heap in such a way that the maximum value is removed
00532      *          at the same time if the heap has reached its storage capacity.
00533      *
00534      * @param         value         new value to add
00535      * @param         heap_array    the heap
00536      * @param[in,out] count         number of values currently in the heap (will update)
00537      * @param         max_size      maximum physical size allocated for the heap
00538      * @tparam  DataType    the type of data stored in the heap - must be
00539      *                      LessThanComparable, Swappable, CopyConstructable,
00540      *                      and CopyAssignable
00541      * @return a pair consising of a flag and a value; the first element is a flag
00542      *         indicating that overflow occurred, and the second element is the value
00543      *         that rotated out of the heap (formerly the maximum) when the new value
00544      *         was added (set only if an overflow occurred)
00545      */
00546     template <typename DataType>
```

```
00547     std::pair<bool, DataType> heap_insert_circular(const DataType& value,
      DataType* heap_array, size_t& count, size_t max_size){
00548         auto max_value  = DataType{};
00549         bool overflowed = count == max_size ? true : false;
00550         if(!overflowed){
00551             heap_insert(value, heap_array, count, max_size);
00552         }
00553         else{                                                   // if the heap is full, replace the
       max value with the new add...
00554             auto m         = max_size > 1 ? _mmheap::max_child(heap_array, 0, max_size-1).second : 0;
00555             max_value      = heap_array[m];
00556             if(value < max_value){                              // if the new value is larger than
       the one rotating out, just rotate the new value
00557                 heap_array[m] = value;
00558                 if(max_size > 1){                               // if this is non-trivial
00559                     if(value < heap_array[0]){                  // check that the new value isn't
       the new min
00560                         std::swap(heap_array[0], heap_array[m]);    //  (if it is, make it so)
00561                     }
00562                     _mmheap::sift_down(heap_array, m, max_size-1);  // sift the new item down
00563                 }
00564             }
00565             else{
00566                 max_value = value;
00567             }
00568         }
00569         return std::pair<bool, int>{overflowed, max_value};
00570     }
00571
00572
00573     /**
00574      * replace and return the value at a given index with a new value
00575      *
00576      * @param new_value   new value to insert
00577      * @param index       index of the value to replace
00578      * @param heap_array  the heap
00579      * @param count       number of values currently stored in the heap
00580      * @tparam  DataType    the type of data stored in the heap - must be
00581      *                      LessThanComparable, Swappable, CopyConstructable,
00582      *                      and CopyAssignable
00583      * @return  the old value being replaced
00584      * @throws  std::runtime_error if the heap is empty
00585      * @throws  std::range_error   if the index is out of range
00586      */
00587     template <typename DataType>
00588     DataType heap_replace_at_index(const DataType& new_value, size_t index,
      DataType* heap_array, size_t count){
00589         if(count == 0){
00590             throw std::runtime_error("Cannot replace value in empty heap.");
00591         }
00592         if(index > count){
00593             throw std::range_error("Index beyond end of heap.");
00594         }
00595         auto old_value    = heap_array[index];
00596         heap_array[index] = new_value;
00597         if(_mmheap::min_level(index)){
00598             if(new_value < old_value){
00599                 _mmheap::bubble_up_min(heap_array, index);
00600             }
00601             else{
00602                 if(_mmheap::has_parent(index) && heap_array[_mmheap::parent(index)] < new_value){
00603                     _mmheap::bubble_up(heap_array, index);
00604                 }
00605                 _mmheap::sift_down(heap_array, index, count-1);
00606             }
00607         }
00608         else{
00609             if(old_value < new_value){
00610                 _mmheap::bubble_up_max(heap_array, index);
00611             }
00612             else{
00613                 if(_mmheap::has_parent(index) && new_value < heap_array[_mmheap::parent(index)]){
00614                     _mmheap::bubble_up(heap_array, index);
00615                 }
00616                 _mmheap::sift_down(heap_array, index, count-1);
00617             }
00618         }
00619         return old_value;
00620     }
00621
00622     /**
```

```
00623      * remove and return value at a given index
00624      *
00625      * @param         index      index to remove
00626      * @param         heap_array the heap
00627      * @param[in,out] count      current number of values in the heap (will update)
00628      * @tparam  DataType    the type of data stored in the heap - must be
00629      *                      LessThanComparable, Swappable, CopyConstructable,
00630      *                      and CopyAssignable
00631      * @return  the value being removed
00632      * @throws  std::runtime_error if the heap is empty
00633      * @throws  std::range_error   if the index is out of range
00634      */
00635     template <typename DataType>
00636     DataType heap_remove_at_index(size_t index, DataType* heap_array, size_t&
       count){
00637         if(count == 0){
00638             throw std::runtime_error("Cannot remove value in empty heap.");
00639         }
00640         if(index > count){
00641             throw std::range_error("Index beyond end of heap.");
00642         }
00643         auto old_value = heap_replace_at_index(heap_array[count-1], index, heap_array, count);
00644         --count;
00645         return old_value;
00646     }
00647
00648     /**
00649      * remove and return the minimum value in the heap
00650      *
00651      * @param heap_array the array
00652      * @param count      the current number of values in the heap (will update)
00653      * @tparam  DataType    the type of data stored in the heap - must be
00654      *                      LessThanComparable, Swappable, CopyConstructable,
00655      *                      and CopyAssignable
00656      * @return the minimum value in the heap
00657      * @throws std::runtime_error if the heap is empty
00658      */
00659     template <typename DataType>
00660     DataType heap_remove_min(DataType* heap_array, size_t& count){
00661         if(count == 0){
00662             throw std::runtime_error("Cannot remove from empty heap.");
00663         }
00664         auto value = heap_array[0];
00665         std::swap(heap_array[0], heap_array[count-1]);
00666         --count;
00667         if(count > 0){
00668             _mmheap::sift_down(heap_array, 0, count-1);
00669         }
00670         return value;
00671     }
00672
00673     /**
00674      * remove and return the maximum value in the heap
00675      *
00676      * @param heap_array the array
00677      * @param count      the current number of values in the heap (will update)
00678      * @tparam  DataType    the type of data stored in the heap - must be
00679      *                      LessThanComparable, Swappable, CopyConstructable,
00680      *                      and CopyAssignable
00681      * @return the maximum value in the heap
00682      * @throws std::runtime_error if the heap is empty
00683      */
00684     template <typename DataType>
00685     DataType heap_remove_max(DataType* heap_array, size_t& count){
00686         if(count == 0){
00687             throw std::runtime_error("Cannot remove from empty heap.");
00688         }
00689         auto value = heap_array[0];
00690         auto m     = _mmheap::max_child(heap_array, 0, count-1);
00691         if(m.first){
00692             value = heap_array[m.second];
00693         }
00694         else{
00695             m.second = 0;
00696         }
00697         heap_remove_at_index(m.second, heap_array, count);
00698         return value;
00699     }
00700 }
00701
00702 #endif
```

Index