Min-Max Heap

Generated by Doxygen 1.8.9.1

Thu Dec 31 2015 13:40:36

Contents

1 Min-Max Heap

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

The main advantage of the Min-Max Heap is the ability to access both the *minimum* and *maximum* values contained in the data structure in constant time. The trade-off is a slight increase in the complexity constant coefficients with respect to traditional heaps. The Min-Max heap still maintains the same *order* of complexity as traditional heaps for all operations.

The heap functions defined in _mmheap.h_ are defined as *templates*, so a heap of any type that is *less-than comparable* and *copy-constructable* is possible. The heap functions are designed to work in-place on top of a regular C++ array.

Namespaces

The file _mmheap.h_ defines two namespaces: `mmheap` and _mmheap. All of the functions necessary to use the min-max heap are exposed in the `mmheap` namespace. It should not be necessary to use the _mmheap namespace (those functions are for internal use only).

Function Reference

Full reference documentation is available in the _"docs"_ project directory. Only the most commonly used functions are described here.

`mmheap::` make_heap()

```
template <typename DataType>
void make_heap (DataType heap_array, size_t size);
```

Creates the min-max heap from an arbitrary C++ array, given the array and its size as arguments.

`mmheap::` heap_insert()

```
template <typename DataType>
void heap_insert (const DataType& value, DataType heap_array, size_t& count,
size_t max_size);
```

Inserts a new value into the heap, given the value, the heap array, the current number of items contained in the heap, and the maximum storage size of the array. The `count` will be increased by one following the function call.

`mmheap::` heap_max()

```
template <typename DataType>
DataType heap_max (DataType heap_array, size_t count);
```

Returns the maximum value contained in the heap, given the heap array and the current number of items contained in the heap.

`mmheap::` heap_min()

```
template <typename DataType>
DataType heap_min (DataType heap_array, size_t count);
```

Returns the minimum value contained in the heap, given the heap array and the current number of items contained in the heap.

mmheap:: heap_remove_max()

```
template <typename DataType>
DataType heap_remove_max (DataType heap_array, size_t& count);
```
Removes and returns the maximum value contained in the heap, given the heap array and the current number of items contained in the heap. The `count` will be decreased by one following the function call.

mmheap:: heap_remove_min()

```
template <typename DataType>
DataType heap_remove_min (DataType heap_array, size_t& count);
```
Removes and returns the minimum value contained in the heap, given the heap array and the current number of items contained in the heap. The `count` will be decreased by one following the function call.

Additional Functions

The following functions are less likely to be commonly used, but are provided under the `mmheap` namespace as well; for more information, *read the documentation in the `docs` directory*.

mmheap:: heap_insert_circular()

Add to heap, rotating the maximum value out if the heap is full.

mmheap:: heap_replace_at_index()

Replace the value at a specific index in the heap array with a new value, and restore the heap property.

mmheap:: heap_remove_at_index()

Remove and return the value at a specified index in the heap array, and restore the heap property.

mmheap:: is_heap()

Returns `true` if an arbitrary array is in a valid Min-Max heap ordering, or `false` otherwise.

License

```
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.
```

2 Namespace Documentation

2.1 _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 (const DataType *heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > min_gchild (const DataType *heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > min_child_or_gchild (const DataType *heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > max_child (const DataType *heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > max_gchild (const DataType *heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > max_child_or_gchild (const DataType *heap_array, size_t i, size_t right_index)
- template<typename DataType >
  void sift_down_min (DataType *heap_array, size_t sift_index, size_t right_index)
- template<typename DataType >
  void sift_down_max (DataType *heap_array, size_t sift_index, size_t right_index)
- template<typename DataType >
  void sift_down (DataType *heap_array, size_t sift_index, size_t right_index)
- template<typename DataType >
  void bubble_up_min (DataType *heap_array, size_t bubble_index)
- template<typename DataType >
  void bubble_up_max (DataType *heap_array, size_t bubble_index)
- template<typename DataType >
  void bubble_up (DataType *heap_array, size_t bubble_index)

2.1.1 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.

2.1.2 Function Documentation

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

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

Parameters

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

Template Parameters

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

Definition at line 418 of file mmheap.h.

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

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

Parameters

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

Template Parameters

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

Definition at line 396 of file mmheap.h.

2.1.2.3 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 374 of file mmheap.h.

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

Definition at line 62 of file mmheap.h.

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

Definition at line 60 of file mmheap.h.

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

Definition at line 61 of file mmheap.h.

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

Definition at line 57 of file mmheap.h.

2.1.2.8 size_t _mmheap::left (size_t *i*) [inline]

Definition at line 58 of file mmheap.h.

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

Definition at line 70 of file mmheap.h.

2.1.2.10 template<typename DataType > std::pair<bool, size_t> _mmheap::max_child (const 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 202 of file mmheap.h.

2.1.2.11 template<typename DataType > std::pair<bool, size_t> _mmheap::max_child_or_gchild (const 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 265 of file mmheap.h.

2.1.2.12 template<typename DataType > std::pair<bool, size_t> _mmheap::max_gchild (const 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 229 of file mmheap.h.

2.1.2.13 template<typename DataType > std::pair<bool, size_t> _mmheap::min_child (const 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 115 of file mmheap.h.

2.1.2.14 template<typename DataType > std::pair<bool, size_t> _mmheap::min_child_or_gchild (const 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 178 of file mmheap.h.

2.1.2.15 template<typename DataType > std::pair<bool, size_t> _mmheap::min_gchild (const 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 142 of file mmheap.h.

2.1.2.16 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 96 of file mmheap.h.

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

Definition at line 56 of file mmheap.h.

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

Definition at line 59 of file mmheap.h.

2.1.2.19 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 355 of file mmheap.h.

2.1.2.20 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 320 of file mmheap.h.

2.1.2.21 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 285 of file mmheap.h.

2.2 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 (const DataType *heap_array, size_t count)
- template<typename DataType >
  DataType heap_min (const 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)
- template<typename DataType >
  bool is_heap (const DataType *array, size_t count)

2.2.1 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.

2.2.2 Function Documentation

2.2.2.1 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 481 of file mmheap.h.

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

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

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

Parameters

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

Template Parameters

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

Returns

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

Definition at line 548 of file mmheap.h.

2.2.2.3 template<typename DataType > DataType mmheap::heap_max (const 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 503 of file mmheap.h.

2.2.2.4 template<typename DataType > DataType mmheap::heap_min (const 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 523 of file mmheap.h.

2.2.2.5 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 637 of file mmheap.h.

2.2.2.6 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 686 of file mmheap.h.

2.2.2.7 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 661 of file mmheap.h.

2.2.2.8 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 589 of file mmheap.h.

2.2.2.9 template<typename DataType > bool mmheap::is_heap (const DataType * *array*, size_t *count*)

determine if an arbitrary array is a Min-Max heap

Parameters

array	the array to check to see if the heap property holds
count	the number of items contained in `array`

Returns

true if `array` is a Min-Max heap, `false` otherwise

Definition at line 710 of file mmheap.h.

2.2.2.10 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

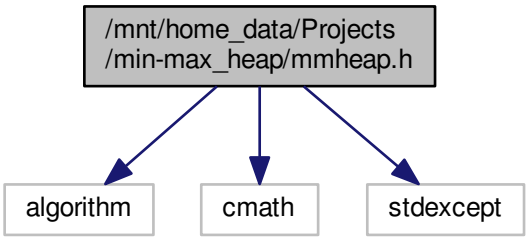
Definition at line 458 of file mmheap.h.

3 File Documentation

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

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

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 (const DataType *heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > _mmheap::min_gchild (const DataType *heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > _mmheap::min_child_or_gchild (const DataType *heap_array, size_t i, size_t right_↩
  index)
- template<typename DataType >
  std::pair< bool, size_t > _mmheap::max_child (const DataType *heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > _mmheap::max_gchild (const DataType *heap_array, size_t i, size_t right_index)
- template<typename DataType >
  std::pair< bool, size_t > _mmheap::max_child_or_gchild (const DataType *heap_array, size_t i, size_t right_↩
  index)
- template<typename DataType >
  void _mmheap::sift_down_min (DataType *heap_array, size_t sift_index, size_t right_index)
- template<typename DataType >
  void _mmheap::sift_down_max (DataType *heap_array, size_t sift_index, size_t right_index)

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

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

3.1.1 Detailed Description

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

This file defines two namespaces:

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

Author

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

Copyright

Definition in file mmheap.h.

3.2 /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
00049 /**
```

```
00050  * The `_mmheap` namespace contains functions that are only intended for internal
00051  * use by the "public-facing" functions in the `mmheap` namespace.  None of the
00052  * functions in `_mmheap::` should be necessary externally.
00053  */
00054 namespace _mmheap{
00055
00056     inline size_t  parent(size_t i)          { return (i - 1) / 2;
             }
00057     inline size_t  has_parent(size_t i)      { return i > 0;
                 }
00058     inline size_t  left  (size_t i)          { return 2*i + 1;
            }
00059     inline size_t  right (size_t i)          { return 2*i + 2;
          }
00060     inline size_t  gparent(size_t i)         { return parent(parent(i));
              }
00061     inline bool    has_gparent(size_t i)     { return i > 2;
    }
00062     inline bool    child(size_t i, size_t c) { return c == left(i) || c == right(i);    }
00063
00064     /*
00065      * fast log-base-2 based on code from:
00066      *     http://stackoverflow.com/a/11398748
00067      * @param  i value to compute the log_2 for (must be > 0)
00068      * @return log-base-2 of `i`
00069      */
00070     uint64_t log_2(uint64_t i) {
00071         static const uint64_t tab64[64] = {
00072             63,  0, 58,  1, 59, 47, 53,  2,
00073             60, 39, 48, 27, 54, 33, 42,  3,
00074             61, 51, 37, 40, 49, 18, 28, 20,
00075             55, 30, 34, 11, 43, 14, 22,  4,
00076             62, 57, 46, 52, 38, 26, 32, 41,
00077             50, 36, 17, 19, 29, 10, 13, 21,
00078             56, 45, 25, 31, 35, 16,  9, 12,
00079             44, 24, 15,  8, 23,  7,  6,  5
00080         };
00081         i |= i >> 1;
00082         i |= i >> 2;
00083         i |= i >> 4;
00084         i |= i >> 8;
00085         i |= i >> 16;
00086         i |= i >> 32;
00087         return tab64[((uint64_t)((i - (i >> 1))*0x07EDD5E59A4E28C2)) >> 58];
00088     }
00089
00090     /**
00091      * returns `true` if `i` is on a Min-Level
00092      *
00093      * @param   i index into the heap
00094      * @return  `true` if `i` is on a min-level
00095      */
00096     inline bool min_level(size_t i) {
00097         return i > 0 ? log_2(++i) % 2 == 0 : true;
00098     }
00099
00100     /**
00101      * get a pair considing of an indication of whether `i` has any children, and
00102      * if so, the index of the child containing the minimum value.
00103      *
00104      * @param   heap_array  the heap
00105      * @param   i           the index (parent) for which to find the min-child
00106      * @param   right-index the index of the right-most element that is part of the heap
00107      * @tparam  DataType    the type of data stored in the heap - must be
00108      *                      LessThanComparable, Swappable, CopyConstructable,
00109      *                      and CopyAssignable
00110      * @return  a pair where the first element is `true` if `i` has children (`false`
00111      *          otherwise), and the second element is the index of the child whose value
00112      *          is smallest (only if the first element is `true`)
00113      */
00114     template <typename DataType>
00115     std::pair<bool, size_t> min_child(const
      DataType* heap_array, size_t i, size_t
      right_index){
00116         std::pair<bool, size_t> result{false, 0};
00117         if(left(i) <= right_index){
00118             auto m = left(i);
00119             if(right(i) <= right_index && heap_array[right(i)] < heap_array[m]){
00120                 m = right(i);
00121             }
00122             result = {true, m};
```

```
00123         }
00124         return result;
00125     }
00126 
00127     /**
00128      * get a pair considing of an indication of whether `i` has any grandchildren, and
00129      * if so, the index of the grandchild containing the minimum value.
00130      *
00131      * @param   heap_array  the heap
00132      * @param   i           the index (parent) for which to find the min-grandchild
00133      * @param   right-index the index of the right-most element that is part of the heap
00134      * @tparam  DataType    the type of data stored in the heap - must be
00135      *                      LessThanComparable, Swappable, CopyConstructable,
00136      *                      and CopyAssignable
00137      * @return  a pair where the first element is `true` if `i` has grandchildren
00138      *          (`false` otherwise), and the second element is the index of the
00139      *          grandchild whose value is smallest (only if the first element is `true`)
00140      */
00141     template <typename DataType>
00142     std::pair<bool, size_t> min_gchild(const
      DataType* heap_array, size_t i, size_t
      right_index){
00143         std::pair<bool, size_t> result{false, 0};
00144         auto l = left(i);
00145         auto r = right(i);
00146         if(left(l) <= right_index){
00147             auto m = left(l);
00148             if(right(l) <= right_index && heap_array[right(l)] < heap_array[m]){
00149                 m = right(l);
00150             }
00151             if(left(r) <= right_index && heap_array[left(r)] < heap_array[m]){
00152                 m = left(r);
00153             }
00154             if(right(r) <= right_index && heap_array[right(r)] < heap_array[m]){
00155                 m = right(r);
00156             }
00157             result = {true, m};
00158         }
00159         return result;
00160     }
00161 
00162     /**
00163      * get a pair considing of an indication of whether `i` has any children, and
00164      * if so, the index of the child or grandchild containing the minimum value.
00165      *
00166      * @param   heap_array  the heap
00167      * @param   i           the index (parent) for which to find the min-(grand)child
00168      * @param   right-index the index of the right-most element that is part of the heap
00169      * @tparam  DataType    the type of data stored in the heap - must be
00170      *                      LessThanComparable, Swappable, CopyConstructable,
00171      *                      and CopyAssignable
00172      * @return  a pair where the first element is `true` if `i` has children
00173      *          (`false` otherwise), and the second element is the index of the
00174      *          child or grandchild whose value is smallest (only if the first
00175      *          element is `true`)
00176      */
00177     template <typename DataType>
00178     std::pair<bool, size_t> min_child_or_gchild(const
      DataType* heap_array, size_t i, size_t
      right_index){
00179         auto m = min_child(heap_array, i, right_index);
00180         if(m.first){
00181             auto  gm = min_gchild(heap_array, i, right_index);
00182             m.second = gm.first && heap_array[gm.second] < heap_array[m.second] ? gm.second : m.second
      ;
00183         }
00184         return m;
00185     }
00186 
00187     /**
00188      * get a pair considing of an indication of whether `i` has any children, and
00189      * if so, the index of the child containing the maximum value.
00190      *
00191      * @param   heap_array  the heap
00192      * @param   i           the index (parent) for which to find the max-child
00193      * @param   right-index the index of the right-most element that is part of the heap
00194      * @tparam  DataType    the type of data stored in the heap - must be
00195      *                      LessThanComparable, Swappable, CopyConstructable,
00196      *                      and CopyAssignable
00197      * @return  a pair where the first element is `true` if `i` has children (`false`
00198      *          otherwise), and the second element is the index of the child whose value
```

```
00199      *          is largest (only if the first element is `true`)
00200      */
00201     template <typename DataType>
00202     std::pair<bool, size_t> max_child(const
      DataType* heap_array, size_t i, size_t
      right_index){
00203         std::pair<bool, size_t> result {false, 0};
00204         if(left(i) <= right_index){
00205             auto m = left(i);
00206             if(right(i) <= right_index && heap_array[m] < heap_array[right(i)]){
00207                 m = right(i);
00208             }
00209             result = {true, m};
00210         }
00211         return result;
00212     }
00213
00214     /**
00215      * get a pair considing of an indication of whether `i` has any grandchildren, and
00216      * if so, the index of the grandchild containing the maximum value.
00217      *
00218      * @param   heap_array  the heap
00219      * @param   i           the index (parent) for which to find the max-grandchild
00220      * @param   right-index the index of the right-most element that is part of the heap
00221      * @tparam  DataType    the type of data stored in the heap - must be
00222      *                      LessThanComparable, Swappable, CopyConstructable,
00223      *                      and CopyAssignable
00224      * @return  a pair where the first element is `true` if `i` has grandchildren
00225      *          (`false` otherwise), and the second element is the index of the
00226      *          grandchild whose value is largest (only if the first element is `true`)
00227      */
00228     template <typename DataType>
00229     std::pair<bool, size_t> max_gchild(const
      DataType* heap_array, size_t i, size_t
      right_index){
00230         std::pair<bool, size_t> result{false, 0};
00231         auto l = left(i);
00232         auto r = right(i);
00233         if(left(l) <= right_index){
00234             auto m = left(l);
00235             if(right(l) <= right_index && heap_array[m] < heap_array[right(l)]){
00236                 m = right(l);
00237             }
00238             if(left(r) <= right_index && heap_array[m] < heap_array[left(r)]){
00239                 m = left(r);
00240             }
00241             if(right(r) <= right_index && heap_array[m] < heap_array[right(r)]){
00242                 m = right(r);
00243             }
00244             result = {true, m};
00245         }
00246         return result;
00247     }
00248
00249     /**
00250      * get a pair considing of an indication of whether `i` has any children, and
00251      * if so, the index of the child or grandchild containing the maximum value.
00252      *
00253      * @param   heap_array  the heap
00254      * @param   i           the index (parent) for which to find the max-(grand)child
00255      * @param   right-index the index of the right-most element that is part of the heap
00256      * @tparam  DataType    the type of data stored in the heap - must be
00257      *                      LessThanComparable, Swappable, CopyConstructable,
00258      *                      and CopyAssignable
00259      * @return  a pair where the first element is `true` if `i` has children
00260      *          (`false` otherwise), and the second element is the index of the
00261      *          child or grandchild whose value is largest (only if the first
00262      *          element is `true`)
00263      */
00264     template <typename DataType>
00265     std::pair<bool, size_t> max_child_or_gchild(const
      DataType* heap_array, size_t i, size_t
      right_index){
00266         auto m = max_child(heap_array, i, right_index);
00267         if(m.first){
00268             auto gm  = max_gchild(heap_array, i, right_index);
00269             m.second = gm.first &&  heap_array[m.second] < heap_array[gm.second] ? gm.second : m.
      second;
00270         }
00271         return m;
00272     }
```

```
00273
00274     /**
00275      * perform min-max heap sift-down on an element (at 'sift_index') that is on a min-level
00276      *
00277      * @param heap_array  the heap
00278      * @param sift_index  the index of the element that should be sifted down
00279      * @param right_index the index of the right-most element that is part of the heap
00280      * @tparam  DataType    the type of data stored in the heap - must be
00281      *                      LessThanComparable, Swappable, CopyConstructable,
00282      *                      and CopyAssignable
00283      */
00284     template <typename DataType>
00285     void sift_down_min(DataType* heap_array, size_t sift_index, size_t right_index){
00286         bool sift_more = true;
00287         while(sift_more && left(sift_index) <= right_index){                          // if a[i] has
      children
00288             sift_more = false;
00289             auto mp = min_child_or_gchild(heap_array, sift_index, right_index);      // get min
      child or grandchild
00290             auto m  = mp.second;
00291             if(child(sift_index, m)){                                                // if the min
      was a child
00292                 if(heap_array[m] < heap_array[sift_index]){
00293                     std::swap(heap_array[m], heap_array[sift_index]);
00294                 }
00295             }
00296             else{                                                                    // min was a
      grandchild
00297                 if(heap_array[m] < heap_array[sift_index]){
00298                     std::swap(heap_array[m], heap_array[sift_index]);
00299                     if(heap_array[parent(m)] < heap_array[m]){
00300                         std::swap(heap_array[m], heap_array[parent(m)]);
00301                     }
00302                     sift_index = m;
00303                     sift_more  = true;
00304                 }
00305             }
00306         }
00307     }
00308
00309     /**
00310      * perform min-max heap sift-down on an element (at 'sift_index') that is on a max-level
00311      *
00312      * @param heap_array  the heap
00313      * @param sift_index  the index of the element that should be sifted down
00314      * @param right_index the index of the right-most element that is part of the heap
00315      * @tparam  DataType    the type of data stored in the heap - must be
00316      *                      LessThanComparable, Swappable, CopyConstructable,
00317      *                      and CopyAssignable
00318      */
00319     template <typename DataType>
00320     void sift_down_max(DataType* heap_array, size_t sift_index, size_t right_index){
00321         bool sift_more = true;
00322         while(sift_more && left(sift_index) <= right_index){                          // if a[i] has
      children
00323             sift_more = false;
00324             auto mp = max_child_or_gchild(heap_array, sift_index, right_index);      // get max
      child or grandchild
00325             auto m  = mp.second;
00326             if(child(sift_index, m)){                                                // if the max
      was a child
00327                 if(heap_array[sift_index] < heap_array[m]){
00328                     std::swap(heap_array[m], heap_array[sift_index]);
00329                 }
00330             }
00331             else{                                                                    // max was a
      grandchild
00332                 if(heap_array[sift_index] < heap_array[m]){
00333                     std::swap(heap_array[m], heap_array[sift_index]);
00334                     if(heap_array[m] < heap_array[parent(m)]){
00335                         std::swap(heap_array[m], heap_array[parent(m)]);
00336                     }
00337                     sift_index = m;
00338                     sift_more  = true;
00339                 }
00340             }
00341         }
00342     }
00343
00344     /**
00345      * perform min-max heap sift-down on an element (at 'sift_index')
```

```
00346      *
00347      * @param heap_array  the heap
00348      * @param sift_index  the index of the element that should be sifted down
00349      * @param right_index the index of the right-most element that is part of the heap
00350      * @tparam  DataType    the type of data stored in the heap - must be
00351      *                      LessThanComparable, Swappable, CopyConstructable,
00352      *                      and CopyAssignable
00353      */
00354     template <typename DataType>
00355     void sift_down(DataType* heap_array, size_t sift_index, size_t right_index){
00356         if(min_level(sift_index)){
00357             sift_down_min(heap_array, sift_index, right_index);
00358         }
00359         else{
00360             sift_down_max(heap_array, sift_index, right_index);
00361         }
00362     }
00363
00364     /**
00365      * perform min-max heap bubble-up on an element (at `bubble_index`) that is on a min-level
00366      *
00367      * @param heap_array    the heap
00368      * @param bubble_index  the index of the element that should be bubbled up
00369      * @tparam  DataType    the type of data stored in the heap - must be
00370      *                      LessThanComparable, Swappable, CopyConstructable,
00371      *                      and CopyAssignable
00372      */
00373     template <typename DataType>
00374     void bubble_up_min(DataType* heap_array, size_t bubble_index){
00375         bool finished = false;
00376         while(!finished && has_gparent(bubble_index)){
00377             finished = true;
00378             if(heap_array[bubble_index] < heap_array[gparent(bubble_index)]){
00379                 std::swap(heap_array[bubble_index], heap_array[gparent(bubble_index)]);
00380                 bubble_index = gparent(bubble_index);
00381                 finished     = false;
00382             }
00383         }
00384     }
00385
00386     /**
00387      * perform min-max heap bubble-up on an element (at `bubble_index`) that is on a max-level
00388      *
00389      * @param heap_array    the heap
00390      * @param bubble_index  the index of the element that should be bubbled up
00391      * @tparam  DataType    the type of data stored in the heap - must be
00392      *                      LessThanComparable, Swappable, CopyConstructable,
00393      *                      and CopyAssignable
00394      */
00395     template <typename DataType>
00396     void bubble_up_max(DataType* heap_array, size_t bubble_index){
00397         bool finished = false;
00398         while(!finished && has_gparent(bubble_index)){
00399             finished = true;
00400             if(heap_array[gparent(bubble_index)] < heap_array[bubble_index]){
00401                 std::swap(heap_array[bubble_index], heap_array[gparent(bubble_index)]);
00402                 bubble_index = gparent(bubble_index);
00403                 finished     = false;
00404             }
00405         }
00406     }
00407
00408     /**
00409      * perform min-max heap bubble-up on an element (at `bubble_index`)
00410      *
00411      * @param heap_array    the heap
00412      * @param bubble_index  the index of the element that should be bubbled up
00413      * @tparam  DataType    the type of data stored in the heap - must be
00414      *                      LessThanComparable, Swappable, CopyConstructable,
00415      *                      and CopyAssignable
00416      */
00417     template <typename DataType>
00418     void bubble_up(DataType* heap_array, size_t bubble_index){
00419         if(min_level(bubble_index)){
00420             if(has_parent(bubble_index) && heap_array[parent(bubble_index)] < heap_array[bubble_index]
    ){
00421                 std::swap(heap_array[bubble_index], heap_array[parent(bubble_index)]);
00422                 bubble_up_max(heap_array, parent(bubble_index));
00423             }
00424             else{
00425                 bubble_up_min(heap_array, bubble_index);
```

```
00426             }
00427         }
00428         else{
00429             if(has_parent(bubble_index) && heap_array[bubble_index] < heap_array[parent(bubble_index)]
      ){
00430                 std::swap(heap_array[bubble_index], heap_array[parent(bubble_index)]);
00431                 bubble_up_min(heap_array, parent(bubble_index));
00432             }
00433             else{
00434                 bubble_up_max(heap_array, bubble_index);
00435             }
00436         }
00437     }
00438 }
00439
00440 /**
00441  * The `mmheap` namespace defines functions that are useful for building and
00442  * maintaining a Min-Max heap.  All necessary ("public-facing") functionality
00443  * is in this namespace.
00444  */
00445 namespace mmheap{
00446     /**
00447      * @brief   make an arbitrary array into a heap (in-place)
00448      * @details Applies Floyd's algorithm (adapted to a min-max heap) to produce
00449      *          a heap from an arbitrary array in linear time.
00450      *
00451      * @param heap_array    the array that will become a heap
00452      * @param size          the number of elements in the array
00453      * @tparam  DataType    the type of data stored in the heap - must be
00454      *                      LessThanComparable, Swappable, CopyConstructable,
00455      *                      and CopyAssignable
00456      */
00457     template <typename DataType>
00458     void make_heap(DataType* heap_array, size_t size){
00459         if(size > 1){
00460             bool finished = false;
00461             for(size_t current = _mmheap::parent(size-1); !finished; --current){
00462                 _mmheap::sift_down(heap_array, current, size-1);
00463                 finished = current == 0;
00464             }
00465         }
00466     }
00467
00468     /**
00469      * insert a new value to the heap (and update the `count`)
00470      *
00471      * @param           value       the new value to insert
00472      * @param           heap_array  the heap
00473      * @param[in,out]   count       the current number of items in the heap (will update)
00474      * @param           max_size    the physical storage allocation size of the heap
00475      * @tparam  DataType    the type of data stored in the heap - must be
00476      *                      LessThanComparable, Swappable, CopyConstructable,
00477      *                      and CopyAssignable
00478      * @throws std::runtime_error if the heap is full prior to the insert operation
00479      */
00480     template <typename DataType>
00481     void heap_insert(const DataType& value, DataType* heap_array, size_t& count,
      size_t max_size){
00482         if(count < max_size){
00483             heap_array[count++] = value;
00484             _mmheap::bubble_up(heap_array, count-1);
00485         }
00486         else{
00487             throw std::runtime_error("Cannot insert into heap - allocated size is full.");
00488         }
00489     }
00490
00491     /**
00492      * get the maximum value in the heap
00493      *
00494      * @param heap_array the heap
00495      * @param count      the current number of values contained in the heap
00496      * @tparam  DataType    the type of data stored in the heap - must be
00497      *                      LessThanComparable, Swappable, CopyConstructable,
00498      *                      and CopyAssignable
00499      * @return the maximum value in the heap
00500      * @throws std::runtime_error if the heap is empty
00501      */
00502     template <typename DataType>
00503     DataType heap_max(const DataType* heap_array, size_t count){
00504         if(count < 1){
```

```
00505             throw std::runtime_error("Cannot get max value in empty heap.");
00506         }
00507         auto m = _mmheap::max_child(heap_array, 0, count-1);
00508         return m.first ? heap_array[m.second] : heap_array[0];
00509     }
00510 
00511     /**
00512      * get the minimum value in the heap
00513      *
00514      * @param heap_array the heap
00515      * @param count      the current number of values contained in the heap
00516      * @tparam  DataType    the type of data stored in the heap - must be
00517      *                      LessThanComparable, Swappable, CopyConstructable,
00518      *                      and CopyAssignable
00519      * @return the minimum value in the heap
00520      * @throws std::runtime_error if the heap is empty
00521      */
00522     template <typename DataType>
00523     DataType heap_min(const DataType* heap_array, size_t count){
00524         if(count < 1){
00525             throw std::runtime_error("Cannot get min value in empty heap.");
00526         }
00527         return heap_array[0];
00528     }
00529 
00530     /**
00531      * @brief   add to heap, rotating the maximum value out if the heap is full
00532      * @details Add to the min-max heap in such a way that the maximum value is removed
00533      *          at the same time if the heap has reached its storage capacity.
00534      *
00535      * @param         value         new value to add
00536      * @param         heap_array    the heap
00537      * @param[in,out] count         number of values currently in the heap (will update)
00538      * @param         max_size      maximum physical size allocated for the heap
00539      * @tparam  DataType    the type of data stored in the heap - must be
00540      *                      DefaultConstructable, LessThanComparable, Swappable,
00541      *                      CopyConstructable, and CopyAssignable
00542      * @return a pair consising of a flag and a value; the first element is a flag
00543      *         indicating that overflow occurred, and the second element is the value
00544      *         that rotated out of the heap (formerly the maximum) when the new value
00545      *         was added (set only if an overflow occurred)
00546      */
00547     template <typename DataType>
00548     std::pair<bool, DataType> heap_insert_circular(const DataType& value,
      DataType* heap_array, size_t& count, size_t max_size){
00549         auto max_value  = DataType{};
00550         bool overflowed = count == max_size ? true : false;
00551         if(!overflowed){
00552             heap_insert(value, heap_array, count, max_size);
00553         }
00554         else{                                                                    // if the heap
      is full, replace the max value with the new add...
00555             auto m         = max_size > 1 ? _mmheap::max_child(heap_array, 0, max_size-1).second : 0;
00556             max_value      = heap_array[m];
00557             if(value < max_value){                                               // if the new
      value is larger than the one rotating out, just rotate the new value
00558                 heap_array[m] = value;
00559                 if(max_size > 1){                                                // if this is
      non-trivial
00560                     if(value < heap_array[0]){                                   // check that
      the new value isn't the new min
00561                         std::swap(heap_array[0], heap_array[m]);                 //  (if it is,
      make it so)
00562                     }
00563                     _mmheap::sift_down(heap_array, m, max_size-1);               // sift the
      new item down
00564                 }
00565             }
00566             else{
00567                 max_value = value;
00568             }
00569         }
00570         return std::pair<bool, DataType>{overflowed, max_value};
00571     }
00572 
00573 
00574     /**
00575      * replace and return the value at a given index with a new value
00576      *
00577      * @param new_value   new value to insert
00578      * @param index       index of the value to replace
```

```
00579      * @param heap_array  the heap
00580      * @param count       number of values currently stored in the heap
00581      * @tparam  DataType    the type of data stored in the heap - must be
00582      *                      LessThanComparable, Swappable, CopyConstructable,
00583      *                      and CopyAssignable
00584      * @return  the old value being replaced
00585      * @throws  std::runtime_error if the heap is empty
00586      * @throws  std::range_error   if the index is out of range
00587      */
00588     template <typename DataType>
00589     DataType heap_replace_at_index(const DataType& new_value, size_t index,
       DataType* heap_array, size_t count){
00590         if(count == 0){
00591             throw std::runtime_error("Cannot replace value in empty heap.");
00592         }
00593         if(index > count){
00594             throw std::range_error("Index beyond end of heap.");
00595         }
00596         auto old_value    = heap_array[index];
00597         heap_array[index] = new_value;
00598         if(_mmheap::min_level(index)){
00599             if(new_value < old_value){
00600                 _mmheap::bubble_up_min(heap_array, index);
00601             }
00602             else{
00603                 if(_mmheap::has_parent(index) && heap_array[_mmheap::parent(index)] < new_value){
00604                     _mmheap::bubble_up(heap_array, index);
00605                 }
00606                 _mmheap::sift_down(heap_array, index, count-1);
00607             }
00608         }
00609         else{
00610             if(old_value < new_value){
00611                 _mmheap::bubble_up_max(heap_array, index);
00612             }
00613             else{
00614                 if(_mmheap::has_parent(index) && new_value < heap_array[_mmheap::parent(index)]){
00615                     _mmheap::bubble_up(heap_array, index);
00616                 }
00617                 _mmheap::sift_down(heap_array, index, count-1);
00618             }
00619         }
00620         return old_value;
00621     }
00622
00623     /**
00624      * remove and return value at a given index
00625      *
00626      * @param         index      index to remove
00627      * @param         heap_array the heap
00628      * @param[in,out] count      current number of values in the heap (will update)
00629      * @tparam  DataType    the type of data stored in the heap - must be
00630      *                      LessThanComparable, Swappable, CopyConstructable,
00631      *                      and CopyAssignable
00632      * @return  the value being removed
00633      * @throws  std::runtime_error if the heap is empty
00634      * @throws  std::range_error   if the index is out of range
00635      */
00636     template <typename DataType>
00637     DataType heap_remove_at_index(size_t index, DataType* heap_array, size_t&
       count){
00638         if(count == 0){
00639             throw std::runtime_error("Cannot remove value in empty heap.");
00640         }
00641         if(index > count){
00642             throw std::range_error("Index beyond end of heap.");
00643         }
00644         auto old_value = heap_replace_at_index(heap_array[count-1], index, heap_array, count);
00645         --count;
00646         return old_value;
00647     }
00648
00649     /**
00650      * remove and return the minimum value in the heap
00651      *
00652      * @param heap_array the array
00653      * @param count      the current number of values in the heap (will update)
00654      * @tparam  DataType    the type of data stored in the heap - must be
00655      *                      LessThanComparable, Swappable, CopyConstructable,
00656      *                      and CopyAssignable
00657      * @return the minimum value in the heap
```

```
00658      * @throws std::runtime_error if the heap is empty
00659      */
00660     template <typename DataType>
00661     DataType heap_remove_min(DataType* heap_array, size_t& count){
00662         if(count == 0){
00663             throw std::runtime_error("Cannot remove from empty heap.");
00664         }
00665         auto value = heap_array[0];
00666         std::swap(heap_array[0], heap_array[count-1]);
00667         --count;
00668         if(count > 0){
00669             _mmheap::sift_down(heap_array, 0, count-1);
00670         }
00671         return value;
00672     }
00673 
00674     /**
00675      * remove and return the maximum value in the heap
00676      *
00677      * @param heap_array the array
00678      * @param count      the current number of values in the heap (will update)
00679      * @tparam  DataType    the type of data stored in the heap - must be
00680      *                      LessThanComparable, Swappable, CopyConstructable,
00681      *                      and CopyAssignable
00682      * @return the maximum value in the heap
00683      * @throws std::runtime_error if the heap is empty
00684      */
00685     template <typename DataType>
00686     DataType heap_remove_max(DataType* heap_array, size_t& count){
00687         if(count == 0){
00688             throw std::runtime_error("Cannot remove from empty heap.");
00689         }
00690         auto value = heap_array[0];
00691         auto m     = _mmheap::max_child(heap_array, 0, count-1);
00692         if(m.first){
00693             value = heap_array[m.second];
00694         }
00695         else{
00696             m.second = 0;
00697         }
00698         heap_remove_at_index(m.second, heap_array, count);
00699         return value;
00700     }
00701 
00702     /**
00703      * determine if an arbitrary array is a Min-Max heap
00704      *
00705      * @param   array   the array to check to see if the heap property holds
00706      * @param   count   the number of items contained in `array`
00707      * @return  true if `array` is a Min-Max heap, `false` otherwise
00708      */
00709     template <typename DataType>
00710     bool is_heap(const DataType* array, size_t count){
00711         bool result = true;                                                    // the empty
    heap and single item are trivially heaps...
00712         if(count > 1){                                                         // more work
    if two or more items
00713             auto i  = count - 1;
00714             while(result && _mmheap::has_parent(i)){                           // after this
    loop, we either fail, or make it to root with result=true
00715                 auto sub_root = _mmheap::parent(i);
00716                 auto value    = array[sub_root];
00717                 if(_mmheap::min_level(sub_root)){                              // min level:
    we must be smaller than children & grandchildren
00718                     auto min_value = _mmheap::min_child_or_gchild(array, sub_root, count-1);
00719                     result &= (!min_value.first)
00720                            || value <  array[min_value.second]
00721                            || value == array[min_value.second];
00722                 }
00723                 else{                                                          // max level:
    we must be larger than children & grandchildren
00724                     auto max_value = _mmheap::max_child_or_gchild(array, sub_root, count-1);
00725                     result &= (!max_value.first)
00726                            || array[max_value.second] <  value
00727                            || array[max_value.second] == value;
00728                 }
00729                 --i;
00730             }
00731         }
00732         return result;
00733     }
```

```
00734 }
00735
00736 #endif
```

3.3 /mnt/home_data/Projects/min-max_heap/README.md File Reference

3.4 /mnt/home_data/Projects/min-max_heap/README.md

```
00001 ## Min-Max Heap
00002 Defines functions for maintaining a Min-Max Heap, as described by Adkinson:
00003 <blockquote>
00004     M. D. Atkinson, J.-R. Sack, N. Santoro, and T. Strothotte. 1986.
00005     Min-max heaps and generalized priority queues.
00006     Commun. ACM 29, 10 (October 1986), 996-1000.
00007     DOI=http://dx.doi.org/10.1145/6617.6621
00008 </blockquote>
00009
00010 The main advantage of the Min-Max Heap is the ability to access both the _minimum_ and _maximum_
       values contained in the data structure in constant time.  The trade-off is a slight increase in the complexity
       constant coefficients with respect to traditional heaps.  The Min-Max heap still maintains the same _order_ of
       complexity as traditional heaps for all operations.
00011
00012 The heap functions defined in _`mmheap.h`_ are defined as _templates_, so a heap of any type that is
       _less-than comparable_ and _copy-constructable_ is possible.  The heap functions are designed to work
       in-place on top of a regular C++ array.
00013
00014 ### Namespaces
00015 The file _`mmheap.h`_ defines two namespaces: `mmheap` and `_mmheap`.  All of the functions necessary
       to use the min-max heap are exposed in the `mmheap` namespace.  It should not be necessary to use the
       `_mmheap` namespace (those functions are for internal use only).
00016
00017 ### Function Reference
00018 Full reference documentation is available in the _"docs"_ project directory.  Only the most commonly
       used functions are described here.
00019
00020 #### `mmheap:: make_heap()`
00021 `template <typename DataType>`<br />
00022 `void make_heap (DataType heap_array, size_t size);`<br />
00023 Creates the min-max heap from an arbitrary C++ array, given the array and its size as arguments.
00024
00025 #### `mmheap:: heap_insert()`
00026 `template <typename DataType>`<br />
00027 `void heap_insert (const DataType& value, DataType heap_array, size_t& count, size_t max_size);`<br
       />
00028 Inserts a new value into the heap, given the value, the heap array, the current number of items
       contained in the heap, and the maximum storage size of the array.  The `count` will be increased by one following
       the function call.
00029
00030 #### `mmheap:: heap_max()`
00031 `template <typename DataType>`<br />
00032 `DataType heap_max (DataType heap_array, size_t count);`<br />
00033 Returns the maximum value contained in the heap, given the heap array and the current number of items
       contained in the heap.
00034
00035 #### `mmheap:: heap_min()`
00036 `template <typename DataType>`<br />
00037 `DataType heap_min (DataType heap_array, size_t count);`<br />
00038 Returns the minimum value contained in the heap, given the heap array and the current number of items
       contained in the heap.
00039
00040 #### `mmheap:: heap_remove_max()`
00041 `template <typename DataType>`<br />
00042 `DataType heap_remove_max (DataType heap_array, size_t& count);`<br />
00043 Removes and returns the maximum value contained in the heap, given the heap array and the current
       number of items contained in the heap.  The `count` will be decreased by one following the function call.
00044
00045 #### `mmheap:: heap_remove_min()`
00046 `template <typename DataType>`<br />
00047 `DataType heap_remove_min (DataType heap_array, size_t& count);`<br />
00048 Removes and returns the minimum value contained in the heap, given the heap array and the current
       number of items contained in the heap.  The `count` will be decreased by one following the function call.
00049
00050 #### Additional Functions
00051 The following functions are less likely to be commonly used, but are provided under the `mmheap`
       namespace as well; for more information, _read the documentation in the `docs` directory_.
00052
00053 ##### `mmheap:: heap_insert_circular()`
```

```
00054 Add to heap, rotating the maximum value out if the heap is full.
00055
00056 ##### `mmheap:: heap_replace_at_index()`
00057 Replace the value at a specific index in the heap array with a new value, and restore the heap
       property.
00058
00059 ##### `mmheap:: heap_remove_at_index()`
00060 Remove and return the value at a specified index in the heap array, and restore the heap property.
00061
00062 ##### `mmheap:: is_heap()`
00063 Returns `true` if an arbitrary array is in a valid Min-Max heap ordering, or `false` otherwise.
00064
00065 ### License
00066 This library is released under the MIT License: http://opensource.org/licenses/MIT
00067 <pre>
00068 Copyright (c) 2015 Jason L Causey, Arkansas State University
00069
00070 Permission is hereby granted, free of charge, to any person obtaining a copy
00071 of this software and associated documentation files (the "Software"), to deal
00072 in the Software without restriction, including without limitation the rights
00073 to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
00074 copies of the Software, and to permit persons to whom the Software is
00075 furnished to do so, subject to the following conditions:
00076 The above copyright notice and this permission notice shall be included in
00077 all copies or substantial portions of the Software.
00078
00079 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
00080 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
00081 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL THE
00082 AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
00083 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
00084 OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
00085 THE SOFTWARE.
00086 </pre>
00087
```