

Implementing Relaxed Weak Queues

Jens Rasmussen

*Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark*

Abstract. I have produced a C++ implementation of *relaxed weak queues*. This newly invented data structure is a simplification of — and supports all priority-queue operations as efficiently as — a run-relaxed heap, namely: *get-highest-priority*, *insert* and *increase-priority* in $O(1)$ worst-case time, and *delete* in $O(\lg n)$ worst-case time, n denoting the number of elements stored prior to the operation. Further, the data structure supports *meld* in $O(\min\{\lg m, \lg n\})$ worst-case time, where m and n denote the sizes of the sub-collections melded.

1. Introduction

The class of *priority queues* that are considered in this paper are abstract data structures that store their elements in compartments called *nodes* and use a strict weak ordering [1, Section 25.3] \prec on the elements. A non-empty priority queue Q stores a *highest-priority element* i in Q such that $i \not\prec j$ for all j in Q ¹ and supports the following five operations:

get-highest-priority: Returns a reference to the compartment in the Q that stores the highest-priority element.

***insert*(e)**: Inserts an element e into a new compartment in Q .

***delete*(n)**: Deletes the node referenced to by n , including the element it stores, from Q .

***increase-priority*(n, e)**: Replaces the value e' contained in the node referenced to by n by e , where it is assumed that $e' \prec e$.

***meld*(Q')**: Moves all elements together with their compartments from the priority queue referenced by Q' into Q .

In this paper I will present a C++ implementation of relaxed weak queues. The implementation is inspired by a project proposal [7] for the implementation of a more enhanced priority queue than the one provided by the STL which does not support *meld* and *increase-priority* and only supports *delete* on the highest-priority element. The goal was to create an implementation

¹ Note that more than one element may satisfy this property, and that in practice this subset is in turn ordered by e.g. time of insertion, but to simplify the description it is assumed that for two elements e and e' in Q where $e \neq e'$ we have that $e \prec e'$ or $e' \prec e$.

that fulfils the container requirements of the C++ standard [1, Section 23.1], provides support for the five operations defined above, and provides support for iterators [1, Section 24] that are valid at all times and provide worst-case constant-time operations. All these enhancements were achieved using the relaxed-weak-queue data structure. To understand the implementation details a good understanding of relaxed weak queues is necessary, which is the reason why a detailed description of the data structure is given in Section 2. An overview of the implementation is given in Section 3 and details about adherence to the C++ standard are given in Section 4. The source code is well commented and is found in the appendix. Finally, benchmark comparisons with other priority queues providing, or almost providing, the above five operations are provided in Section 5.

2. Relaxed Weak Queues

Relaxed weak queues² [6] are inspired by run-relaxed heaps [3]. The key difference is that a run-relaxed heap consists of a collection of heap-ordered binomial trees, whereas a relaxed weak queue consists of a collection of *perfect weak heaps*.³ A *perfect weak heap*⁴ can be formally defined by the following three *properties*:

1. The root has no left subtree.
2. The right subtree of the root is a complete binary tree.
3. For every node, every element stored in the right subtree of that node does not have higher priority than the element stored at that node.⁵

From Properties 1 and 2 above it follows that, like binomial trees, a perfect weak heap has n elements if and only if it has height $k = \lg n$, $k \in \mathbb{N}_0$ and $n = 2^k$. An important ability of perfect weak heaps is that they can be *joined* in constant time as follows:

1. The root with the higher-priority element becomes the new root and gets its height increased by one.
2. The root with the lesser-priority element becomes the right son of the new root and obtains as its left subtree the original right subtree of the new root.

The inverse of a join is referred to as a *split*. Split and join are depicted in Figure 1.

A *weak queue* is defined as a collection of disjoint perfect weak heaps. By using this structure, worst-case constant-time insert and get-highest priority is achievable (see, for example [5]). The worst-case constant-time bound

² Hereafter *relaxed weak queue* and *queue* will be used interchangeably.

³ For more about the relationship between heap-ordered binomial trees and perfect weak heaps, see [6, Section 1].

⁴ Hereafter *perfect weak heap* and *heap* will be used interchangeably.

⁵ Reformulation of [6, Section 1].

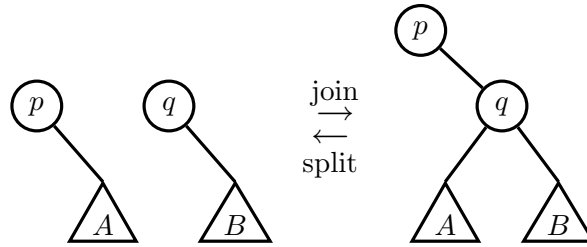


Figure 1. Joining and splitting two perfect weak heaps of the same size (assuming that the element stored at node p has no lesser priority than the element stored at node q). (The figure is originally from [6] and is used with the permission of the authors.)

on increase-priority can be achieved by relaxing Property 3 above. The resulting data structure is called a *relaxed weak queue*.

What follows is a detailed description of how the complexities of the five main operations on relaxed weak queues are achieved.

2.1 Get-Highest-Priority

To enable get-highest-priority in constant time, a pointer to the node holding the highest-priority element is kept and maintained across all operations. The only complication is when the highest-priority element is deleted (see Section 2.4).

2.2 Insert

Relaxed weak queues structure their nodes in a sequence of disjoint perfect weak heaps ordered by non-decreasing heights. This sequence is named the *heap sequence*. To describe the insert operation, some definitions are needed:

Definition 1 (Heap-height sequence). *The heap-height sequence H of some heap sequence S is a sequence of integers $\langle h_{-1}, h_0, h_1, \dots, h_n \rangle$, $n \in \mathbb{N}$, and $h_k \in \{0, 1, 2\}$ for all $k \in \{-1, 0, 1, \dots, n\}$ where h_k denotes the number of heaps of height k in S . The heap-height sequence H is also defined by its function $f_H : \mathbb{N}_0 \cup \{-1\} \rightarrow \{0, 1, 2\}$ where $f_H(i) = h_i$.*

Note that the indexing in a heap-height sequence starts from -1 only to simplify the following definition of *regularity*:

Definition 2 (Regularity). *A heap-height sequence H with function f_H of some heap sequence S is called regular when for all $\{k \in \mathbb{N}_0 \cup \{-1\} \mid H(k) = 2\}$ there exists an $i \in \mathbb{N}_0 \cup \{-1\}$ such that $i < k$ and $f_H(i) = 0$ and $f_H(j) = 1$ for all $\{j \in \mathbb{N}_0 \mid i < j < k\}$. If H is regular, then S is also said to be regular.*

Definition 3 (Inject). *An inject is an operation that takes a perfect weak heap h_j of height j , $j \in \mathbb{N}_0$ as input and adds it to the heap sequence S with a heap-height sequence H with function f_H , where $j \leq \min\{k \in \mathbb{N}_0 \mid f_H(k) \geq 1\}$. The operation has two steps:*

fix-up: *If one or more pairs of heaps of equal heights exist in S , the pair with the smallest height is joined.*

insert: *h_j is added to S .*

To insert a new element into the queue, a node is created and the element is inserted into this node. This perfect weak heap of height one is then injected into the heap sequence.

To enable lookups of new highest-priority elements in logarithmic time (see Section 2.4) and logarithmic time melding (see Section 2.5) the length of the heap-sequence is kept bounded by $\lfloor \lg n \rfloor + 1$ heaps, where n is the number of elements in the relaxed weak queue, by joining one pair of equal-sized heaps, if such a pair exists, before insertion as described above. Although new equal-sized heaps can result from the merging, it is proved in [6, Lemma 2.1, Lemma 2.2] that the sequence will indeed remain logarithmic in size. The lemmas are restated here for completeness:

Lemma 1. [6, Lemma 2.1] *Injecting a perfect weak heap h_j , $j \in \mathbb{N}_0$ of height j into a regular heap sequence with a heap-height sequence H with function f_H , where $j \leq \min\{k \in \mathbb{N}_0 \mid f_H(k) \geq 1\}$, keeps the sequence regular.*

Lemma 2. [6, Lemma 2.2] *Let τ denote the number of perfect weak heaps in a relaxed weak queue of size n . If the heap sequence is regular then $\tau \leq \lfloor \lg n \rfloor + 1$.*

2.3 Increase-Priority

To achieve the constant-time bound on increase-priority, violations of perfect weak heap Property 3 above are allowed.

A node is called *violating* if it is in some right subtree and the element of that node has higher priority than the element of the root of the subtree. A node that is *potentially* violating Property 3 is labeled *marked*. There are four types of marked nodes:

Members: are left children of marked parents.

Leaders: are not members and have a marked left child.

(A leader together with a sequence of members forms a *run*.)

Fellow: are marked nodes that are not in runs.

Chairperson: when two or more fellows of the same height exists one of them is elected *chairperson* and is thus both a fellow and a chairperson.

(A chairperson together with the fellows of the same height forms a *club*. A fellow that is the only marked node of some height is not a part of a club.)⁶

To be able to mark and unmark nodes in constant time, elements representing marked nodes are kept in one of four separate structures each holding homogeneous types. Implementation details are given in Section 3.2.

To increase the priority of an element, a pointer to the node containing the element is required to write the new priority value in constant time, after which the node is marked. Note that to ascertain whether the node is actually violating the structure, one would have to keep traversing parent pointers until reaching a parent through a right child. This would be too expensive as the worst-case length of a path to the first such parent is $\lg n$, where n is the number of elements in the queue.

2.4 Delete

Deleting an element from a relaxed weak queue poses two difficulties:

1. If the height of the perfect weak heap containing the element is higher than one, then deleting a node will destroy the *perfect* structure of the heap as the size will no longer be a power of two.
2. If the highest-priority element is deleted, the next highest-priority element must be found.

For the first problem let p_j be the heap of smallest height j in the heap sequence S of the relaxed weak queue. Split p_j into two heap sequences: $\langle p' \rangle$ and $\langle p_0, p_1, \dots, p_{j-1} \rangle$ where p_i is a heap with height i and the heap p' has height zero. The latter heap sequence is simply appended to S . Let q' be the node containing the target element, we split the sub-heap rooted at q' into two heap sequences: $\langle q' \rangle$ and $\langle q_0, q_1, \dots, q_{k-1} \rangle$ as above. Next, p' is joined with q_0 the result of which is joined with q_1 and so on until a perfect weak heap of height k is created, the root of which replaces q' . The worst-case time complexity of the splitting and joining is $O(\max\{\lg m, \lg n\})$, where m and n are the sizes of the sub-heaps rooted at p' and q' , respectively. This can be seen from the fact that both a split and a join each take $O(1)$ worst-case time, and that $j + k$ splits and k joins are performed, where $j = \lg m$ and $k = \lg n$.

The second problem – that of finding a new highest priority element – is solved by realizing that the only nodes where it can be located are either roots of the heap sequence or marked nodes. So to keep delete bounded by $O(\lg n)$ worst-case time, where n is the number of elements in the queue, the number of roots and marked nodes must be bounded by $O(\lg n)$. The length of the root-sequence is kept bounded by $\lfloor \lg n \rfloor + 1$ heaps, where n is

⁶ *Fellows* and *chairpersons* replace, but must not be confused with, *singletons* and *pairs* described in [6].

the number of elements in the relaxed weak queue, by joining one pair of equal-sized heaps before insertion (see Section 2.2).

In a similar fashion, the number of marked nodes is bounded by $\lfloor \lg n \rfloor + 1$, by preceding every marking by an unmarking operation, *reduce* (described below), that attempts to unmark a node through transformations of heaps.

2.4.1 Reduce

Two composite transformations: *run transformations* and *club transformations* — referred to collectively as λ -*reductions* — are used to reduce the number of markings. Run transformations operate on runs and club transformations operate on clubs. Thus the number of nodes that cannot be λ -reduced, i.e. fellows with no associated club, decide the upper bound on the total number of marked nodes in the queue. For completeness the original proof of the upper bound [6, Lemma 2.4] is restated:

Lemma 3. [6, Lemma 2.4] *Let λ denote the total number of marked nodes in a relaxed heap of size n . Due to λ -reductions, it must hold that $\lambda \leq \lfloor \lg n \rfloor + 1$ at all times.*

Proof. Since a root cannot be violating, a node having the maximum height can never be marked. That is, the height of any marked node must be between 0 and $\lfloor \lg n \rfloor - 1$. After increasing λ by one, if $\lambda = \lfloor \lg n \rfloor + 1$, there must exist a run or a club. Therefore, premises for a λ -reduction are fulfilled and it can be used to reduce λ by at least one. \square

The λ -reductions make use of four kinds of primitive transformations as depicted in Figure 2. To make later implementation details clear, the two λ -reductions are now described in full detail⁷. The main idea is to place two marked nodes in one of three *constellations*:

- A. They are siblings and their parent is unmarked, in which case a sibling transformation can be applied. A sibling transformation always reduces the number of marked nodes by at least one.
- B. One is the right child of the other, in which case a parent transformation will reduce the number of marked nodes by one.
- C. They are both right children of equal height with unmarked parents, in which case a pair transformation can be applied. Pair transformations always reduce the number of marked nodes by at least one.

In the following the referred targets of the transformations are always the roots of the depicted (sub)heaps in Figure 2, e.g.: “a parent transformation is applied to s ” means that s is represented as p in Figure 2(b).

Run transformations: Let q be the leader and s its left child.

⁷ The description of λ -reductions is a reformulation of [6, Section 2].

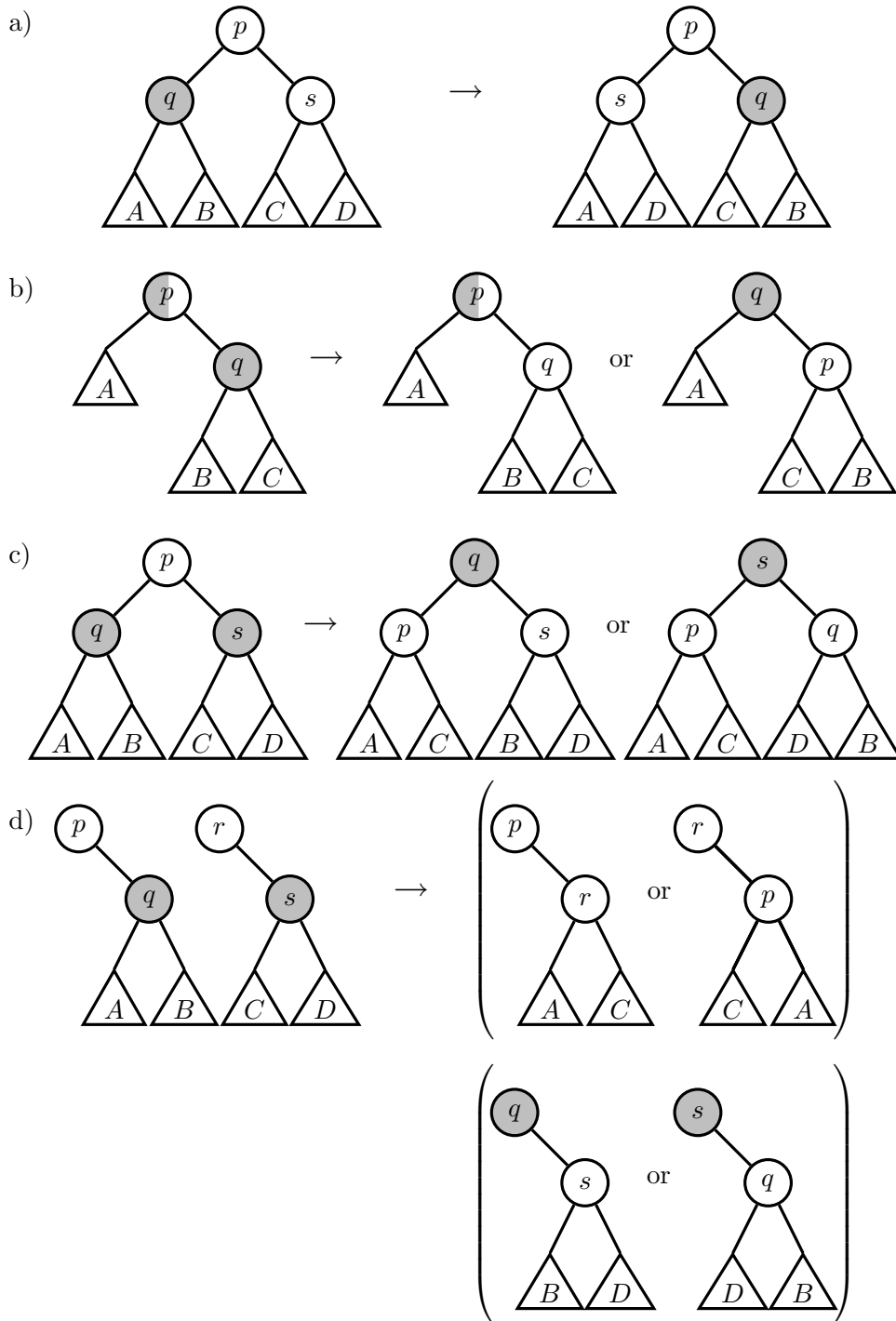


Figure 2. Primitives used in a λ -reduction: a) cleaning transformation, b) parent transformation, c) sibling transformation, and d) pair transformation. Marked nodes are drawn in grey. A node that is half grey and half white may be marked or unmarked. (The figure is originally from [6] and is used with the permission of the authors.)

1. q is a right child:
 - a) Do a parent transformation on the parent of q . Stop if the number of marked nodes decreases.
 - b) Now the parent of s is unmarked and s is a right child. If the sibling of s is marked, we are in Constellation A.
 - c) Apply the parent transformation on the former parent of q , which is now its right child and has the former left subtree of q as its left subtree. Stop if the number of marked nodes decreases.
 - d) Now s has become the right child of q so we are in Constellation B.
2. q is a left child: ($\Rightarrow q$'s parent is unmarked)
 - a) If q 's sibling p is marked, we are in Constellation A. Else perform a cleaning transformation on q 's parent.
 - b) Now p 's left child is s . If p 's right child is marked, we are in Constellation A. Else apply a cleaning transformation on p .
 - c) Now p 's right child is s . Perform a parent transformation on p . Stop if the number of marked nodes decreases.
 - d) Now both p 's children are marked and we are in Constellation A.

Club transformations: Let p and q be two fellows from the same club.

1. If any siblings of p or q are marked, we are in Constellation A. Else make p and q right children by applying cleaning transformation on their parents in case they are not.
2. If any of the parents of p or q are marked, we are in Constellation B. Else we are in Constellation C.

2.5 Meld

To meld two queues P and Q with sizes m and n , respectively, where m is no larger than n , the two main things to be done are:

1. One heap sequence must be inserted into the other.
2. Marked nodes of both queues must be merged together. This may introduce new clubs and may make the number of marked nodes exceed $\lfloor \lg(n + m) \rfloor + 1$.

To keep the size of the heap sequence bounded by $\lfloor \lg(n + m) \rfloor + 1$ two steps are taken:

1. Let S_P and S_Q be the heap sequences of P and Q respectively. S_Q is divided into two heap sequences $S_{Q'} = \{s \in S_Q \mid s \text{ has height } \geq \max\{i \in \mathbb{N}_0 \mid \text{there exists a heap in } S_P \text{ with height } i\}\}$ and $S_{Q''} = S_Q \setminus S_{Q'}$. Note that $S_{Q'}$ and $S_{Q''}$ are regular.
2. Let $S_{P \cup Q''} = S_P \cup S_{Q''}$. Note that $S_{P \cup Q''}$ may contain, for each height $h \in \mathbb{N}_0$, up to four heaps of height h . Now extract the heap with greatest height from $S_{P \cup Q''}$ and inject it into $S_{Q'}$, by Lemma 1 $S_{Q'}$ remains regular. Continue extracting heaps with greatest height from $S_{P \cup Q''}$

and injecting them into $S_{Q'}$. By Lemma 2 the size of the resulting heap sequence is bounded by $\lfloor \lg(n+m) \rfloor + 1$.

It is clear that the above steps take $O(\lg m)$ worst-case time.

Melding leaders and members from P and Q poses no problems as the marked nodes do not depend on one another. Chairpersons and fellows from P and Q are dependent as melding them may introduce two chairpersons for the same club or create new clubs. To handle this, two steps are performed:

1. Fellows are processed and joined by height. If both P and Q have a chairperson for some height $j \in \mathbb{N}_0$, then P 's chairperson becomes a fellow. If no chairperson exists for some height $h \in \mathbb{N}_0$ and the union of the fellows of height h from P and Q has two elements, then one is elected chairperson.
2. As the chairpersons from P and Q are now disjoint and independent (i.e. all new chairpersons have been elected and no two chairpersons have the same height) they can simply be joined.

As the number of marked nodes in p is bounded by $\lfloor \lg m \rfloor + 1$ and assuming that each join of the data structures containing the marked nodes can be done in constant time, meld takes $O(\lg m) = O(\min\{\lg m, \lg n\})$ worst-case time.

3. Implementation Overview

An often used solution for implementing weak heaps is to use an array approach (e.g. [4]) but due to the mutable nature of the heap sequence, a pointer-based approach was chosen instead.

The node data structure used has the typical pointers to parent, left and right child and a field holding the element; furthermore it holds a height field storing the height and two pointers to the previous and next element in the iterator sequence used, the sequence is ordered by time of insertion (i.e. at insert the element is added to the end of the iterator sequence). Lastly a pointer to an element in one of the marked node data structures is kept, it is null if the node is unmarked.

Two dummy nodes are used: *sentinel* and *one-past-end*. Sentinel is a node with all node pointers pointing to itself, it has height zero and is unmarked. It is used as a null value and has the advantage that many explicit checks for null can be avoided when only investigating the height field or whether the node is marked. One-past-end represents the value *following* the end of the iterator sequence. It is linked to both the beginning and end of the iterator sequence.

I have made use of two auxiliary structures, *heap store* and *mark store*, responsible for the heap sequence and marked nodes, respectively.⁸

3.1 Heap Store

The heap store contains the following data structures:

Heap sequence: A doubly-linked list of heaps, linked by exploiting the unused parent and left pointers of roots; letting parent point to previous root and left to next root in the sequence. To be able to check whether a node is a root one simply has to check whether the node has height equal to or higher than its left child.

Join schedule: Conceptually a stack containing pointers to the first element of pairs of heaps with equal heights, ordered by increasing height of the pairs. It consists of two data structures:

Join sequence: A dynamic array of pointers to roots of the heap sequence with size: $\lfloor \lg n \rfloor + 1$, where n is the number of elements in the queue. The value at index $i \in 0, 1, \dots, \lfloor \lg n \rfloor + 1$ either holds a pointer to the first heap in a pair of heaps of height i , if such exists, or holds a null value if no pair of height i exist.

Join queue: A stack of indexes to the join sequence. Each element in the stack has value $j \in 0, 1, \dots, \lfloor \lg n \rfloor + 1$, where n is the size of the queue, and represents a pair of heaps of height j in the heap sequence.

The reason for dividing the join schedule into two data structures when a stack would seem to suffice, is that a pair transformation or parent transformation may replace a root of the heap sequence that was scheduled for a join. To perform the replacement in the case that the former root was scheduled for a join in worst-case constant-time, a pointer back to the join schedule is needed.

To do worst-case constant-time extraction of the smallest pair of heaps in the sequence, the top of the join queue is simply accessed and used to index the join sequence. To do worst-case constant-time replacement of a root in the join schedule the height value of the root is simply used to index the join sequence.

3.1.1 Heap-Store Operations

Conceptually, the heap store provides the following operations:

inject(h): Inserts the heap h into the heap sequence with height no larger than any heap already in the sequence. To keep the sequence regular a join is made on the first pair in the sequence before the insert is performed.

⁸ They are adaptations of *heap store* and *node store*, described in [6, Section 2].

eject: Removes and returns the heap with smallest height from the sequence.

***replace*(m, n)**: Replaces the heap m with the heap n in the heap sequence, possibly updating the join schedule.

***meld*(H)**: Melds the heap store with the heap store H by the procedure described in Section 2.5.

3.2 Mark Store

The mark store contains and maintains four data structures each storing one type of marked nodes — referred to as *mark containers*. The mark containers all store the same type of element, namely a pair consisting of a pointer to the node they represent and the mark type of that node — referred to as a *mark item*. As all marked nodes point back to a mark item (necessary for unmarking of nodes in constant time), validity of the pointers must be ensured.

To provide *meld* in $O(\min\{\lg m, \lg n\})$ worst-case time, merging of mark stores must be possible in at most the same worst-case time. To achieve this, lists are used as the concrete data structures of the mark containers as they provide pointer validity and worst-case constant-time merging.

3.2.1 Fellows and Chairpersons

Fellows are grouped into separate lists by height, and the lists are kept in an array — the *fellow list array* — for random-access indexing by height. Random access of fellow lists are necessary for the maintenance of the mark container holding chairpersons across mark and unmark operations.

A special note concerning chairpersons: When a fellow lists size becomes greater than one, a chairperson mark item is created and inserted into the chairperson list. The node pointed to by the mark item at the front of the fellow list, is then linked to the pair list instead of the fellow-list mark item. By knowing that the fellow-list mark item pointing to a node of type chairperson, are at the front of fellow lists of size greater than one, they can easily be accessed through the fellow-list array when performing unmark or meld.

3.2.2 Members and Leaders

Space is saved by representing members by a single mark item only containing the mark type. This can be done as no direct access to nodes through the member mark container is necessary. When marking and unmarking nodes, runs are easily identified through parent and left pointers.

3.2.3 Mark-Store Operations

Conceptually, the mark store provides the following operations

mark(*n*): Marks a node *n* by inserting a new mark item into the appropriate mark container and linking it to *n*. Updating of the marked node lists may be necessary (e.g. when a new leader takes over a run).

unmark(*n*): Unmarks a node *n* by removing its mark item from a mark container and setting its mark item pointer to null. Updates of the mark containers may be necessary (e.g. when a fellow is elected chairperson).

reduce: If a run or club exists, the number of marked nodes is reduced by at least one using the procedure described in Section 2.4.1.

meld(*M*): Melds the mark store with the mark store *M* using the procedure described in Section 2.5.

4. C++ Implementation Details

The relaxed weak queue class, `relaxed_weak_queue`, orders its elements from greatest to least as `std::priority_queue` and accepts the following template parameters:

V: Value stored in the queue.

C: Comparison functor to use when comparing values, it is assumed to define a strict weak ordering. Defaults to `std::less`.

A: Allocator used. Defaults to `std::allocator<V>`.

S: (Template template parameter) random-access sequence used for the fellow-list array and the priority queue. **S** must take a value type and an allocator type as its two template parameters. Defaults to `std::vector`.

The reason, why **S** is used, is that to achieve worst-case time bounds on insert, random-access sequences that only have amortized constant-time complexity on `push_back` and `pop_back` operations (most often due to their choice of the doubling array resizing strategy) cannot be used. But empirical evidence [8] suggests that worst-case constant-time random-access sequences do not perform well in practice, which is why a vector was chosen as default.

4.1 Mark Store Optimization

Due to the high cost of memory allocation and deallocation, a list of free mark items is maintained. This way, mark and unmark only move mark items between linked lists using `std::list::splice`. The size of the list is bounded by $\lfloor \lg n \rfloor + 1 + \lfloor (\lg n + 1) / 2 \rfloor$ where *n* is the number of elements in the queue, $\lfloor \lg n \rfloor + 1$ are the maximal number of marked nodes and $\lfloor (\lg n + 1) / 2 \rfloor$ is the maximal number of mark items of type chairperson.

4.2 Container

`relaxed_weak_queue` fulfills all container requirements defined in [1, Section 23.1]. It fails to meet the extended requirements defined in [1, Section 23.1.10], more specifically it may throw when performing `erase` if the comparison functor `C` throws or if, during a reduce, a marked node list throws during a `push_front` or `push_back` due to a node changing its mark type during transformations.

4.3 Iterators

`relaxed_weak_queue` supports the bidirectional iterator type. The choice of using two pointers solely dedicated to order the iterator sequence combined with the fact that no swapping of elements between nodes is done, guarantees iterator validity unless the element pointed to is erased. They perform all of their operations by dereferencing or following pointers which results in constant time complexity, worst-case.

Care must be taken when using `iterator` as it is possible to change the priority of an element, bypassing *increase-priority*, and thus make the whole data structure inconsistent.

4.4 Exception Safety

`relaxed_weak_queue` is always in a state with valid iterators, no memory-leaks⁹ and deleting or clearing (through `clear()`) the queue is always possible. It fails to meet the basic guarantee of exception-safety as it does not guarantee to be in a valid state after an exception has been thrown, due to the complexities in ensuring validity of the mark store and heap store if a comparison or an insertion to a marked node list fails during *reduce*, *delete* or *meld*.

4.5 Allocator Support

Support for custom allocators is limited by the following:

1. `pointer` and `const_pointer` are set to `T*` and `T const*` respectively.
2. `size_type` and `difference_type` are set to `size_t` and `ptrdiff_t`, respectively.
3. For efficiency reasons, simple types and objects are copied to allocated memory by replacement `new` instead of calls to `A::construct()` and they are destroyed by calls to their default destructors instead of calls to `A::destroy()`.

The above is in full accordance with the C++ standard (see [1, Section 20.1.5]).

⁹ Verified by profiling with *Valgrind* [11].

5. Benchmarks

Benchmark comparisons were performed against a CPH STL library [2] implementation of priority queues using *2-3 heaps* [10] and LEDA library (6.0) [9] implementations of priority queues using *Fibonacci heaps* and *binary heaps*. The priority queues from the LEDA library do not support `meld`.

The elements used were pairs of unsigned integers (p, e) , where p is the priority type and e is the information type of the element. For tests of n elements, the elements have values: $(1000, 1000), (1001, 1001), \dots, (1000 + n - 1, 1000 + n - 1)$.¹⁰ Simple integer comparison of priority types was used for ordering.

The operations tested were: `insert`, `erase` (delete), `top` (get-highest-priority), `pop` (delete on highest-priority element) and `increase` (increase-priority).

For all operations, before the timer was started, the following arrays were created: `a` containing the elements in sorted order and `b` containing the elements in random order. Furthermore, for `erase` and `increase`, the array `p` was created by inserting the elements in the order given by `b` into the queue to be tested `q`, and storing the returned pointers into `p`.

The following describes how the operations themselves were benchmarked:

`insert`: The elements were inserted into `q` by iterating over `b` and performing `q.insert()`.

`top + pop`: `q.top()` followed by `q.pop()` were performed until `q` was empty.

`erase`: `q` was emptied by iterating over `p` and performing `q.erase()`.

`increase`: `q` was updated by iterating over `p` and increasing the priority of each element by ten.

As can be seen from the benchmark results shown in Figures 3–7, my implementation of relaxed weak queues performs well when doing `insert` and `get-highest-priority` combined with `delete highest priority`, but performs poorly when doing `delete` and `increase`. The benchmark for `increase` could indicate that the $O(1)$ bound had not been achieved, but profiling¹¹ clearly shows that the number of member-function calls per operation and the relative execution time of each member-function is bounded by constants (e.g. `mark` and `unmark` are bounded by about 2.6 when doing `increase`).

¹⁰ LEDA's priority-queue interface only allows pairs.

¹¹ *Valgrind* was used for all profiling, see [11].

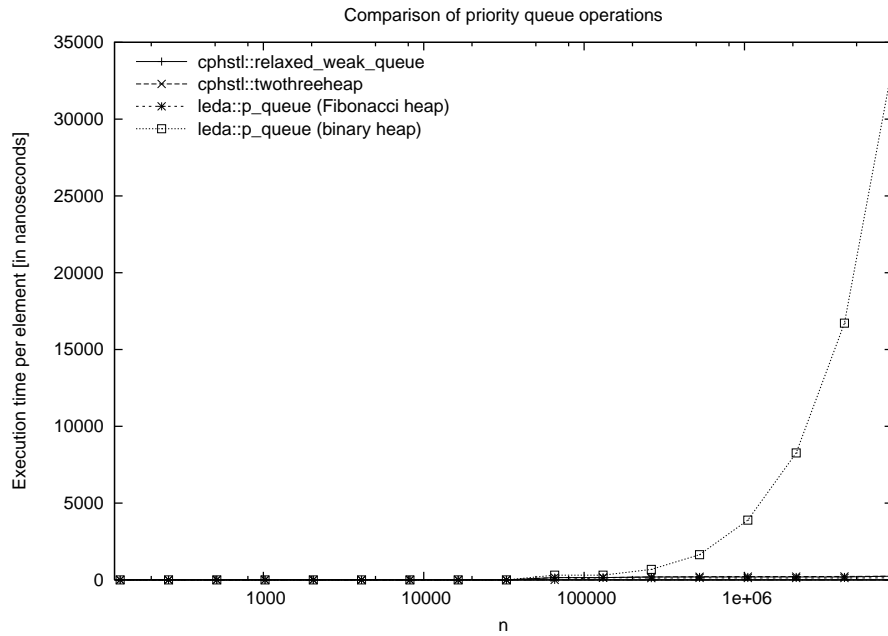


Figure 3. Benchmark of insert.

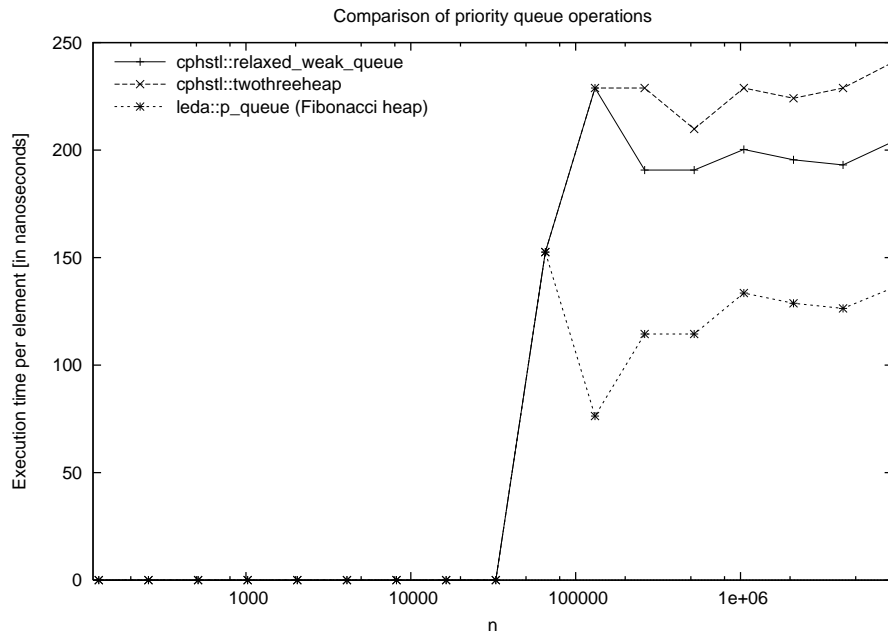


Figure 4. Benchmark of insert.

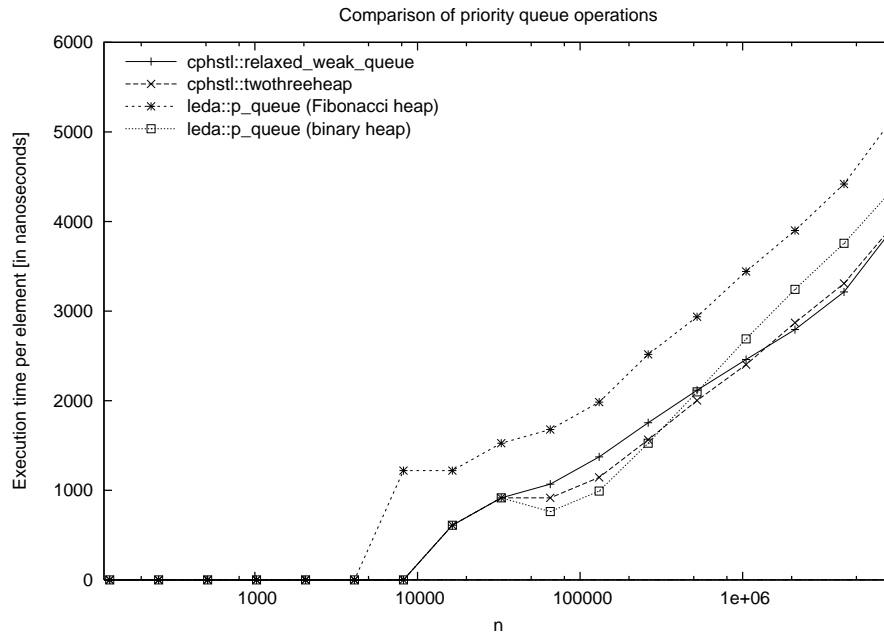


Figure 5. Benchmark of get-highest-priority followed by delete of highest-priority element.

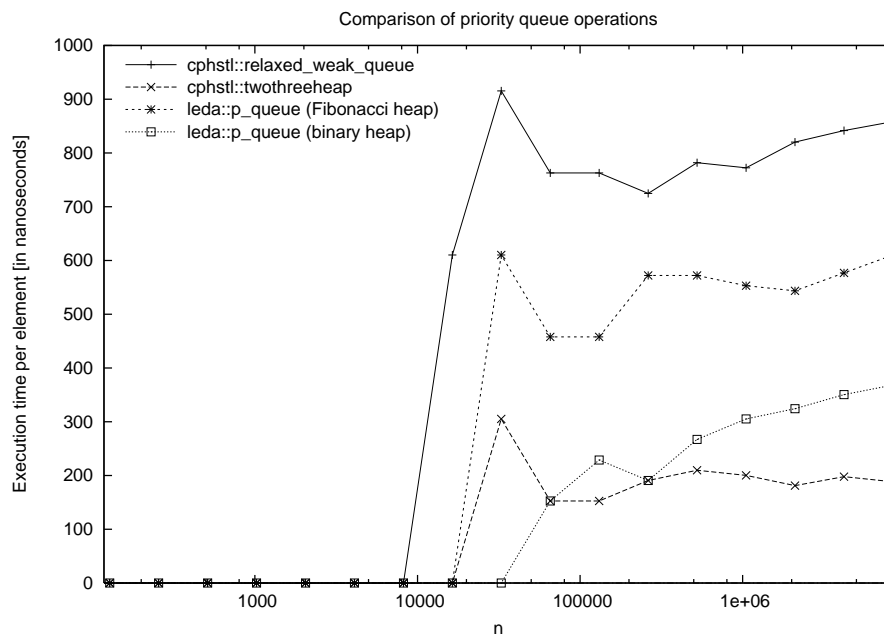


Figure 6. Benchmark of delete.

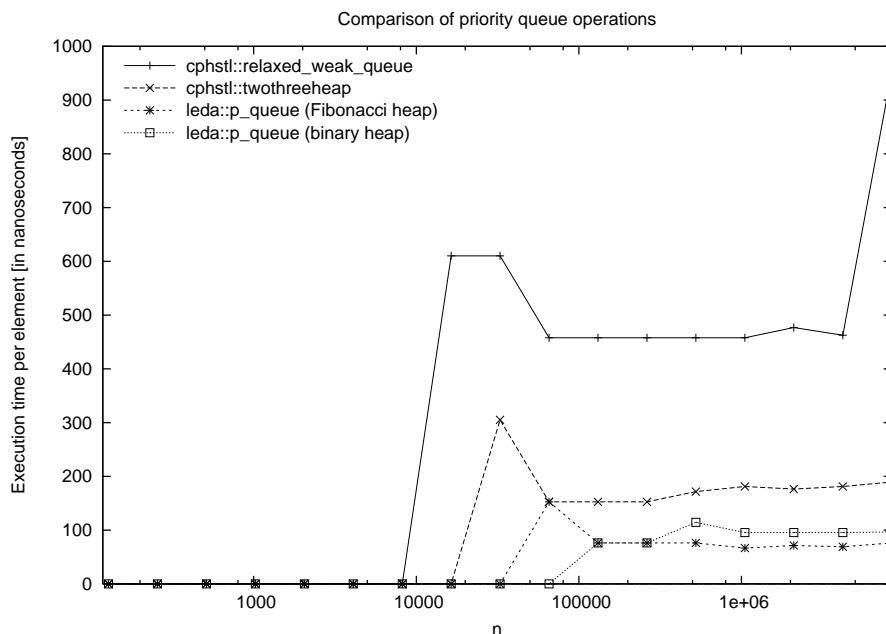


Figure 7. Benchmark of increase-priority.

The reason for the poor performance is due to the large constants involved in maintaining the mark store, specifically the time spent on doing λ -reductions. Profiling indicates that delete and increase use respectively 50% and 80% of their execution time on reduce. The bottleneck in reduce are the transformations, which account for about 90% of the total running time of reduce.¹²

The benchmark was performed on a 2.6 kernel Linux workstation with a 1600 MHz x86 processor and a 1 MB L2 cache. The compiler used was GNU's gcc version 4.2.3 with the `-O3` flag set.

6. Conclusion

I produced the first implementation of relaxed weak queues. The implementation provides meld in $\min(\lg m, \lg n)$ worst-case constant-time, where m and n denote the subcollections to be melded. Lastly I tested the implementation against other priority queues that fulfilled or nearly fulfilled the enhanced requirements sketched out in Section 1. I conclude that the data structure is indeed able to achieve the declared worst-case time bounds, but that the constants involved in maintaining marked nodes are too high for

¹² The reason for the good performance in delete-highest-priority benchmark is that it does not cause a call to reduce, because all highest-priority elements are contained in root nodes due to the fact that no increase operations are done.

the current implementation to be of practical importance. It is an open question whether there exists an implementation of relaxed weak queues that performs well in practice.

Appendix: Source code

A. relaxed_weak_queue.h++	19
B. heap_store.h++	28
C. mark_store.h++	32
D. node.h++	40
F. node_base.h++	41
E. types.h++	41
H. transformations.h++	43
G. helpers.h++	42
I. bidirectional_iterator.h++	47
J. traits.h++	48

A. relaxed_weak_queue.h++

```

/* Author: Jens Rasmussen
 *
 * Implementation of Amr Elmasry's, Claus Jensens and Jyrki Katajainens
 * Relaxed Weak Queues which offer  $O(1)$  time push, pop and increase
 * in the worst case and  $O(\lg(n))$  time erase in the worst case.
 *
 * If using the default std::vector as random-access array the worst case time
 * for push and erase become amortized.
 *
 * NB: Unlike in the article, relaxed_weak_queue is a max-priority queue, so
 * that it is more similar to std::priority_queue, also singletons and
 * pairs have been redefined as fellows and chairpersons.
 */

#ifndef _CPHSTL_RELAXED_WEAK_QUEUE_
#define _CPHSTL_RELAXED_WEAK_QUEUE_

#include<cstdlib> // size_t, ptrdiff_t
#include<cmath> // log2, ceil
#include<list> // std::list
#include<utility> // std::pair
#include<algorithm> // std::lexicographical_compare, std::equal
#include<vector> // std::vector
#include<functional> // std::less
#include<memory> // std::allocator
#include<transformations.h++> // helpers for doing transformations on heaps
#include<helpers.h++> // helpers for accessing properties of heaps
#include<bidirectional_iterator.h++> // iterator
#include<types.h++> // mark_type
#include<node.h++> // node
#include<mark_store.h++> // mark store
#include<heap_store.h++> // heap store

#ifdef __DEBUG__
#include<cassert>
#include<iostream>
#include<string>
#include<sstream>
#endif // __DEBUG__

namespace cphstl {

template <
typename V, // value
typename C = std::less<V>, // comparison
typename A = std::allocator<V>, // allocator
template <typename, typename> class S = std::vector // random-access sequence
>
class relaxed_weak_queue //: public A::template rebind<node<V, A> >::other
{
protected:
typedef node<V, A> node_type;
public:
// types

```

```

typedef V value_type;
typedef C comparator_type;
typedef A allocator_type;
typedef value_type& reference;
typedef value_type const& const_reference;
typedef value_type* pointer;
typedef value_type const* const_pointer;
typedef size_t size_type;
typedef ptrdiff_t difference_type;
typedef bidirectional_iterator<V, node_type, C, A, S, false> iterator;
typedef bidirectional_iterator<V, node_type, C, A, S, true> const_iterator;
typedef typename std::reverse_iterator<iterator> reverse_iterator;
typedef typename std::reverse_iterator<const_iterator> const_reverse_iterator;
protected:
typedef typename A::template rebind<node_type>::other node_allocator;
typedef typename A::template rebind<node_type*>::other node_ptr_allocator;
typedef heap_store<node_type, C, A, S> heap_store_type;
typedef typename A::template rebind<heap_store_type>::other
    heap_store_allocator;
typedef S<node_type*, node_ptr_allocator> node_stack_type;
typedef mark_store<node_type, C, A, S> mark_store_type;
typedef typename node_type::mark_list_iterator mark_list_iterator;
typedef typename mark_store_type::mark_list_array_iterator
    mark_list_array_iterator;
typedef typename A::template rebind<mark_store_type>::other
    mark_store_allocator;

// member variables
node_allocator allocator;
comparator_type comparator;
size_type no_of_elements; // size()
// x, y are in [first, second)
// <=> floor(lg(x)) = floor(lg(y)) = floor(lg(size()))
std::pair<size_type, size_type> lg_interval;
node_type* first_priority;
node_type* one_past_end; // equals end() and one_past_end->next equals begin()
mark_store_type* mark_store;
heap_store_type* heap_store;

public:

explicit
relaxed_weak_queue(comparator_type const& comp = comparator_type(),
                  , allocator_type const& alloc = allocator_type())
    : allocator(alloc), comparator(comp), no_of_elements(0),
      lg_interval(0, 1), first_priority(&node_type::sentinel)
{
    init();
}

// complexity: size of sequence [p, q)
template <typename IT>
explicit
relaxed_weak_queue(IT p, IT q, comparator_type const& comp = comparator_type(),
                  , allocator_type const& alloc = allocator_type())
    : allocator(alloc), comparator(comp), no_of_elements(0),
      lg_interval(0, 1), first_priority(&node_type::sentinel)
{
    init();
    push(p, q);
}

// complexity: size()
~relaxed_weak_queue()
{
    destroy_all_nodes();
    destroy_node(one_past_end);
    destroy_mark_store();
    destroy_heap_store();
}

// preserves iteration order, does not copy structure
// complexity: other.size()
relaxed_weak_queue(relaxed_weak_queue const& other)
    : allocator(other.allocator), comparator(other.comparator),
      no_of_elements(0), lg_interval(0, 1), first_priority(&node_type::sentinel)
{
    init();
    push(other.begin(), other.end());
}

// preserves iteration order, does not copy structure
// complexity: max(size(), other.size())

```

```

relaxed_weak_queue& operator=(relaxed_weak_queue const& other)
{
    if (this != &other)
    {
        clear();
        allocator = other.allocator;
        comparator = other.comparator;
        push(other.begin(), other.end());
    }
    return *this;
}

// if empty: uninitialized allocated space is returned
inline
iterator top() const throw()
{
    return iterator(first_priority);
}

// complexity: log(size())
inline
void pop()
{
    erase(top());
}

// complexity: constant
iterator push(const_reference val)
{
    node_type* n = create_node(val);
    try
    {
        if (first_priority == &node_type::sentinel)
        {
            push_to_empty_sequence(n);
            return iterator(n);
        }
        if (comparator(first_priority->value, val))
        {
            first_priority = n;
        }
        heap_store->inject(n, comparator);
        increase_size();
        add_to_iterator_sequence(n);
        return iterator(n);
    }
    catch (...)
    {
        destroy_node_and_value(n);
        throw;
    }
}

// complexity: size of sequence [p, q)
template <typename IT>
IT push(IT p, IT q)
{
    for (; p != q; ++p)
    {
        push(*p);
    }
    return p;
}

// NB: due to a design error the O(1) worst-case time cannot
// be achieved as during a reduce, either target of a pair transformation
// may be in the join-schedule of the heap store and we therefore have to do
// a cascading join

// increases priority of p - assumes that comparator(p->value, v)
// complexity: constant
void increase(iterator p, const_reference v)
{
    node_type* n = p.position;
    n->value = v;
    if (!n->is_root())
    {
        mark_store->mark(n);
        mark_store->reduce(comparator, *heap_store);
    }
    if (comparator(first_priority->value, v))
    {

```

```

    first_priority = n;
}
}
// returns a constant iterator to one-past it
// assumes that size() > 0 and it is valid
// complexity: log(size())
iterator erase(iterator it)
{
    if (size() == 1) // is root - no need to unmark
    {
        destroy_node_and_value(it.position);
        decrease_size();
        reset();
        return iterator(one_past_end);
    }
    node_type* p = it.position;
    iterator one_past_it = ++it;
    mark_store->unmark(p); // must be done before height-change
    node_type* q = heap_store->eject();
    node_type* r;
    // we may have ejected only heap
    if (heap_store->head == &node_type::sentinel)
    {
        r = split(q);
        mark_store->unmark(r);
        heap_store->head = r;
    }
    while (q->right != &node_type::sentinel)
    {
        r = split(q);
        mark_store->unmark(r);
        heap_store->add_to_heap_sequence(r);
    }
    if (p != q)
    {
        node_stack_type split_stack;
        split_stack.reserve(p->height);
        bool p_was_root = p->is_root();
        while(p->right != &node_type::sentinel)
        {
            mark_store->unmark(p->right); // do before split as p is not root
            r = split(p);
            split_stack.push_back(r);
        }
        while(!split_stack.empty())
        {
            // loop has s->left == sentinel and q->left == sentinel
            // as invariant => no children vanish when using join
            r = split_stack.back();
            split_stack.pop_back();
            q = join(q, r, comparator);
        }
        if (p_was_root)
        {
            // let q replace p in heap sequence
            heap_store->replace(q, p);
            // if q was head and if p was right root of q then eject of q made p head
            if (p == heap_store->head)
            {
                heap_store->head = q;
            }
        }
        else
        {
            (p == p->parent->right) ? set_right_child(p->parent, q)
                : set_left_child(p->parent, q);
            safe_set_left_child(q, p->left);
            p->left = p->parent = &node_type::sentinel;
            mark_store->mark(q);
            mark_store->reduce(comparator, *heap_store); // for the new marking
        }
    }
    // for decrement of number of elements
    mark_store->reduce(comparator, *heap_store);
    if (p == first_priority)
    {
        // scan all roots and marked nodes to find next first_priority
        set_new_first_priority();
    }
    del_from_iterator_sequence(p);
}

```

```

    destroy_node_and_value(p);
    decrease_size();
    return one_past_it;
}

// other is empty and reset after operation
// complexity: min(log(size()), log(other.size()))
void meld(relaxed_weak_queue& other)
{
    if (other.empty())
    {
        return;
    }
    if (empty())
    {
        swap(other);
        return;
    }
    if (size() < other.size())
    {
        std::swap(heap_store, other.heap_store);
        std::swap(mark_store, other.mark_store);
    }
    no_of_elements += other.size();
    increase_size_on_meld();

    mark_store->meld(*other.mark_store, comparator, *heap_store);
    heap_store->meld(*other.heap_store, comparator);

    if (comparator(first_priority->value, other.first_priority->value))
    {
        first_priority = other.first_priority;
    }
    // splice iterator lists
    one_past_end->prev->next = other.one_past_end->next;
    other.one_past_end->next->prev = one_past_end->prev;
    one_past_end->prev = other.one_past_end->prev;
    other.one_past_end->prev->next = one_past_end;

    other.reset();
}

// complexity: constant
void swap(relaxed_weak_queue& other)
{
    std::swap(comparator, other.comparator);
    std::swap(allocator, other.allocator);
    std::swap(mark_store, other.mark_store);
    std::swap(heap_store, other.heap_store);
    std::swap(first_priority, other.first_priority);
    std::swap(no_of_elements, other.no_of_elements);
    std::swap(lg_interval, other.lg_interval);
    std::swap(one_past_end, other.one_past_end);
}

// complexity: size()
void clear()
{
    destroy_all_nodes();
    mark_store->clear();
    heap_store->clear();
    reset();
}

iterator begin()
{
    return iterator(one_past_end->next);
}
const_iterator begin() const
{
    return const_iterator(one_past_end->next);
}
iterator end()
{
    return iterator(one_past_end);
}
const_iterator end() const
{
    return const_iterator(one_past_end);
}
reverse_iterator rbegin()
{

```

```

    return reverse_iterator(end());
}
const_reverse_iterator rbegin() const
{
    return const_reverse_iterator(end());
}
reverse_iterator rend()
{
    return reverse_iterator(begin());
}
const_reverse_iterator rend() const
{
    return const_reverse_iterator(begin());
}

allocator_type get_allocator() const
{
    return allocator_type(allocator);
}

comparator_type get_comparator() const
{
    return comparator;
}

size_type size() const
{
    return no_of_elements;
}
// the calculation is based on the assumption that no new nodes are marked
// as the equation involves solving for x: a = x + lg(x), an approximation
// is made, so the result is slightly below the true value.
size_type max_size() const
{
    // no of nodes that can be allocated before free memory is used up
    size_type a(allocator.max_size());
    size_type b(sizeof(node_type)); // size of nodes
    size_type c(sizeof(node_type::node_list)); // size of fellow lists
    size_type d(size()); // no of allocated nodes
    size_type e(mark_store->no_of_fellow_lists());
    // solve for n: a = n + ((lg(n + d) - e) * c) / b)
    // (in words: a = n + no of added fellow lists)
    return (a - ceil(((ceil(log2(a + d)) - e) * c) / b));
}
bool empty() const
{
    return size() == 0;
}

inline
bool operator==(relaxed_weak_queue const& other) const
{
    return size() == other.size()
        && std::equal(begin(), end(), other.begin());
}
inline
bool operator!=(relaxed_weak_queue const& other) const
{
    return !(*this == other);
}
inline
bool operator<(relaxed_weak_queue const& other)
{
    return std::lexicographical_compare(begin(), end(),
        other.begin(), other.end(), comparator);
}
inline
bool operator>(relaxed_weak_queue const& other)
{
    return other < *this;
}
inline
bool operator<=(relaxed_weak_queue const& other)
{
    return !(other < *this);
}
inline
bool operator>=(relaxed_weak_queue const& other)
{
    return !(*this < other);
}

```

```

protected: // helpers

void destroy_all_nodes()
{
    node_type* iter = one_past_end->next;
    node_type* n;
    while (iter != one_past_end)
    {
        n = iter;
        iter = iter->next;
        destroy_node_and_value(n);
    }
}

inline
node_type* create_node()
{
    node_type* n = allocator.allocate(1);
    try
    {
        new (n) node_type();
    }
    catch (...)
    {
        destroy_node(n);
        throw;
    }
    return n;
}

inline
node_type* create_node(const_reference val)
{
    node_type* n = allocator.allocate(1);
    try
    {
        new (n) node_type(val);
    }
    catch (...)
    {
        destroy_node(n);
        throw;
    }
    return n;
}

inline
void destroy_node(node_type* n)
{
    allocator.deallocate(n, 1);
}

inline
void destroy_node_and_value(node_type* n)
{
    n->~node_type();
    destroy_node(n);
}

inline
mark_store_allocator get_mark_store_allocator()
{
    return mark_store_allocator(allocator);
}

inline
heap_store_allocator get_heap_store_allocator()
{
    return heap_store_allocator(allocator);
}

void create_mark_store()
{
    mark_store_allocator alloc = get_mark_store_allocator();
    mark_store = alloc.allocate(1);
    try
    {
        new (mark_store) mark_store_type(alloc);
    }
    catch (...)
    {
        alloc.deallocate(mark_store, 1);
        throw;
    }
}

void destroy_mark_store()
{
    mark_store->~mark_store_type();
    mark_store_allocator alloc = get_mark_store_allocator();
}

```

```

    alloc.deallocate(mark_store, 1);
}
void create_heap_store()
{
    heap_store_allocator alloc = get_heap_store_allocator();
    heap_store = alloc.allocate(1);
    try
    {
        new (heap_store) heap_store_type(alloc);
    }
    catch (...)
    {
        alloc.deallocate(heap_store, 1);
        throw;
    }
}
void destroy_heap_store()
{
    heap_store->~heap_store_type();
    heap_store_allocator alloc = get_heap_store_allocator();
    alloc.deallocate(heap_store, 1);
}
// assumes that first_priority is no longer in heap sequence
// new first_priority is either a root or a marked node
void set_new_first_priority()
{
    node_type* p = first_priority = heap_store->head;
    // root list
    for (; p != &node_type::sentinel; p = next_root(p))
    {
        if (comparator(first_priority->value, p->value))
        {
            first_priority = p;
        }
    }
    // runs
    mark_list_iterator iter = mark_store->leader_list.begin();
    mark_list_iterator _end = mark_store->leader_list.end();
    for (; iter != _end; ++iter)
    {
        for (p = iter->node; p->is.marked(); p = p->left)
        {
            if (comparator(first_priority->value, p->value))
            {
                first_priority = p;
            }
        }
    }
    // fellows
    mark_list_array_iterator v_iter = mark_store->fellow_list_array.begin();
    mark_list_array_iterator v__end = mark_store->fellow_list_array.end();
    for (; v_iter != v__end; ++v_iter)
    {
        iter = (*v_iter)->begin();
        _end = (*v_iter)->end();
        for (; iter != _end; ++iter)
        {
            if (comparator(first_priority->value, iter->node->value))
            {
                first_priority = p;
            }
        }
    }
}
// pushes a new node onto empty heap sequence
void push_to_empty_sequence(node_type* n)
{
    first_priority = n;
    heap_store->head = n;
    next_root(n) = prev_root(n) = &node_type::sentinel;
    add_to_iterator_sequence(n);
    increase_size();
}
void init()
{
    create_heap_store();
    try
    {
        create_mark_store();
        try
        {

```

```

        create_one_past_end();
    }
    catch (...)
    {
        destroy_mark_store();
        throw;
    }
}
catch (...)
{
    destroy_heap_store();
    throw;
}
}
void reset()
{
    first_priority = heap_store->head = &node_type::sentinel;
    one_past_end->prev = one_past_end->next = one_past_end;
    no_of_elements = lg_interval.first = 0;
    lg_interval.second = 1;
}
void create_one_past_end()
{
    one_past_end = create_node();
    one_past_end->parent = one_past_end->left = &node_type::sentinel;
    one_past_end->prev = one_past_end->next = one_past_end;
}
inline
void add_to_iterator_sequence(node_type* n)
{
    n->prev = one_past_end->prev;
    n->prev->next = n;
    n->next = one_past_end;
    n->next->prev = n;
}
inline
void del_from_iterator_sequence(node_type* n)
{
    n->prev->next = n->next;
    n->next->prev = n->prev;
}
inline
void increase_size()
{
    if (++no_of_elements == lg_interval.second)
    {
        do_increase_size();
    }
}
// size can maximally double at merges so maximally
// one new fellow list is created
inline
void increase_size_on_meld()
{
    if (no_of_elements >= lg_interval.second)
    {
        do_increase_size();
    }
}
inline
void do_increase_size()
{
    mark_store->expand();
    heap_store->expand();
    lg_interval.first = lg_interval.second;
    lg_interval.second = lg_interval.second << 1;
}
inline
void decrease_size()
{
    if (--no_of_elements < lg_interval.first)
    {
        mark_store->contract();
        heap_store->contract();
        lg_interval.second = lg_interval.first;
        lg_interval.first = lg_interval.first >> 1;
    }
}
}
// test functions:
#ifdef __DEBUG__

```

```

public:
#include<test_functions.h++>
#endif // __DEBUG__

};

} // namespace cphstl

#endif // __CPHSTL_RELAXED_WEAK_QUEUE__

```

B. heap_store.h++

```

/* Represents the heap store of relaxed weak queues.
 * It is responsible for maintaining the heap sequence
 * (i.e. doubly linked list of roots of perfect weak binary trees).
 *
 * */

#ifndef __CPHSTL_HEAP_STORE__
#define __CPHSTL_HEAP_STORE__

#include<types.h++> // size_type
#include<helpers.h++> // helpers accessing properties of heaps

namespace cphstl {

template<
typename N, // node
typename C, // comparator
typename A, // allocator
template<typename, typename> class S // random-access sequence
>
class heap_store
{
public:
typedef N node_type;
typedef C comparator_type;
typedef typename A::template rebind<size_type>::other allocator_type;
protected:
typedef S<node_type*, allocator_type> join_sequence_type;
typedef typename A::template rebind<size_type>::other size_type_allocator;
typedef S<size_type, size_type_allocator> join_queue_type;
public:
node_type* head;
// join_sequence[i] != sentinel
// <=> there are two heaps in heap sequence with height i
join_sequence_type join_sequence;
// the smallest pair in heap sequence has height i = join_queue.back()
join_queue_type join_queue;

heap_store(allocator_type const& a)
: join_sequence(a), join_queue(a), head(&node_type::sentinel)
{}

~heap_store()
{}

// removes and returns the smallest perfect weak heap in the sequence
// assumes that heap is not empty (in which case sentinel is ejected)
node_type* eject()
{
node_type* r = head;
if (r == join_sequence[r->height]) // was scheduled for a join
{
join_sequence[r->height] = &node_type::sentinel;
join_queue.pop_back();
}
next_head();
return r;
}

// inserts a binary heap into the heap sequence that is not larger than
// any of the heaps already in the sequence
inline
void inject(node_type* h, comparator_type const& comparator)
{
sequence_fixup(comparator);
insert_heap(h);
}

// assumes that other has no nodes that are higher than this' highest node
// assumes that other and this are not empty

```

```

void meld(heap_store& other, comparator_type const& comparator)
{
    // set other_end == end of others heap sequence
    node_type* other_end = other.head;
    node_type* tmp = next_root(other_end);
    for (; tmp != &node_type::sentinel; other_end = tmp,
        tmp = next_root(tmp))
    {}
    if (other_end->height <= head->height)
    {
        for (; other_end != &node_type::sentinel; other_end = prev_root(other_end))
        {
            inject(other_end, comparator);
        }
    }
    else
    {
        node_type* this_end = head;
        size_type max_height(other_end->height - 1);
        // remove join jobs with heights in [0, max_height]
        size_type i;
        for (; !join_queue.empty()
            && (i = join_queue.back()) <= max_height; join_queue.pop_back())
        {
            join_sequence[i] = &node_type::sentinel;
        }
        // go to last node of this' heap sequence with height <= max_height
        tmp = next_root(this_end);
        for (; tmp != &node_type::sentinel && tmp->height <= max_height; )
        {
            this_end = tmp;
            tmp = next_root(tmp);
        }
        head = next_root(this_end);
        if (head == &node_type::sentinel)
        {
            tmp = prev_root(other_end);
            next_root(other_end) = prev_root(other_end) = &node_type::sentinel;
            head = other_end;
        }
        else
        {
            prev_root(head) = &node_type::sentinel;
        }
        // inject all nodes from the sub-sequences other_end and this_end downwards
        while (other_end != &node_type::sentinel)
        {
            if (other_end->height >= this_end->height)
            {
                other_end = inject_and_iterate(other_end, comparator);
            }
            else // this_end cannot be sentinel
            {
                this_end = inject_and_iterate(this_end, comparator);
            }
        }
        while (this_end != &node_type::sentinel)
        {
            this_end = inject_and_iterate(this_end, comparator);
        }
    }
    other.clear();
}

// assumes that _new is not part of heap sequence and that _old is
// makes _new take _old's place in heap sequence (takes care of join schedule)
void replace(node_type* _new, node_type* _old)
{
    if (next_root(_old) != &node_type::sentinel
        && _old->height == next_root(_old)->height)
    {
        // old is in join schedule - make _new replace it
        join_sequence[_old->height] = _new;
    }
    upd_root_list_left(_new, _old);
    upd_root_list_right(_new, _old);
    if (head == _old)
    {
        head = _new;
    }
}

```

```

// insert r into start of root list, assumes that all of r's satellite
// data are set to sentinel and first != sentinel
inline
void add_to_heap_sequence(node_type* r)
{
    next_root(r) = head;
    prev_root(head) = r;
    prev_root(r) = &node_type::sentinel;
    head = r;
}

void clear()
{
    join_sequence.clear();
    join_queue.clear();
    head = &node_type::sentinel;
}

inline
void expand()
{
    join_sequence.push_back(&node_type::sentinel);
}

inline
void contract()
{
    join_sequence.pop_back();
}

protected: // helpers

// joins two heaps of equal size in the heap sequence (if existant)
void sequence_fixup(comparator_type const& comparator)
{
    if (join_queue.empty())
    {
        return;
    }
    size_type i = join_queue.back();
    join_queue.pop_back();
    node_type* p = join_sequence[i];
    join_sequence[i] = &node_type::sentinel;
    bool is_head = (p == head);
    p = join_root_pair(p, comparator);
    if (is_head)
    {
        head = p;
    }
    if (next_root(p) != &node_type::sentinel)
    {
        push_to_join_if_pair(p);
    }
}

// assumes: h->height <= head->height and head != sentinel;
inline
void insert_heap(node_type* h)
{
    add_to_heap_sequence(h);
    push_to_join_if_pair(h);
}

inline
node_type* next_head()
{
    head = next_root(head);
    prev_root(head) = &node_type::sentinel;
}

// insert p after q in root list
inline
void put_after_root(node_type* p, node_type* q)
{
    upd_root_list_right(p, q);
    prev_root(p) = q;
    next_root(q) = p;
}

// insert p before q in root list
inline
void put_before_root(node_type* p, node_type* q)
{
    upd_root_list_left(p, q);
    next_root(p) = q;
}

```

```

    prev_root(q) = p;
}
// if p has same height as next root in sequence
// then push them on heap for later joining
// assumes that h and next_root(h) are roots
inline
void push_to_join_if_pair(node_type* h)
{
    size_type h_height = h->height;
    if (h_height == next_root(h)->height)
    {
        join_sequence[h_height] = h;
        join_queue.push_back(h_height);
    }
}
inline
node_type* inject_and_iterate(node_type* p, comparator_type const& comp)
{
    node_type* q = prev_root(p);
    inject(p, comp);
    return q;
}
#ifdef __DEBUG__
// useful for debugging
public:
    bool _validate()
    {
        bool is_valid = true;
        // SEQUENCE
        node_type* p = head;
        node_type* q = next_root(p);
        bool were_equal = false;
        for (; q != &node_type::sentinel; p = q, q = next_root(q))
        {
            if (were_equal)
            {
                is_valid = is_valid && p->height < q->height;
                were_equal = false;
            }
            else
            {
                is_valid = is_valid && p->height <= q->height;
                were_equal = p->height == q->height;
            }
        }
        // JOIN SCHEDULE
        join_sequence_type js(join_sequence);
        join_queue_type jq(join_queue);
        if (!jq.empty())
        {
            p = js[jq.back()];
            jq.pop_back();
            is_valid = is_valid && p->height == next_root(p)->height;
        }
        while (!jq.empty())
        {
            q = js[jq.back()];
            jq.pop_back();
            is_valid = is_valid && p->height < q->height;
            is_valid = is_valid && q->height == next_root(q)->height;
            p = q;
        }
        // ROOTS
        p = head;
        for (; p != &node_type::sentinel; p = next_root(p))
        {
            is_valid = is_valid && !p->is_marked();
            is_valid = is_valid && p->is_root();
        }
        return is_valid;
    }
}
#endif // __DEBUG__
};

} // namespace cphstl
#endif // __CPHSTL_HEAP_STORE__

```

C. mark_store.h++

```

/* Represents the mark store which has responsibility of all marked nodes.
 *
 * Chairpersons, leaders and members are kept in doubly linked lists.
 * Fellows are kept in a random-access sequence passed as template parameter.
 *
 */

#ifndef _CPHSTL_MARK_STORE_
#define _CPHSTL_MARK_STORE_

#include<types.h++>           // mark_type, mark_info, size_type
#include<helpers.h++>         // helpers for accessing properties of heaps
#include<transformations.h++> // node-tree transformations
#include<heap_store.h++>

namespace cphstl {

template <
typename N,                  // node
typename C,                  // comparator
typename A,                  // allocator
template <typename, typename> class S // random-access sequence
>
class mark_store
{
public:
    typedef N node_type;
    typedef C comparator_type;
protected:
    typedef mark_info<node_type, mark_type> mark_info_type;
    typedef typename A::template rebind<mark_info_type>::other
        mark_info_allocator;
    typedef typename node_type::mark_list mark_list;
    typedef typename A::template rebind<mark_list>::other mark_list_allocator;
    typedef typename A::template rebind<mark_list*>::other
        mark_list_ptr_allocator;
    typedef S<mark_list*, mark_list_ptr_allocator> mark_list_array;
    typedef heap_store<N, C, A, S> heap_store_type;
public:
    typedef typename A::template rebind<mark_store<N, C, A, S> >::other
        allocator_type;
    typedef typename node_type::mark_list_iterator mark_list_iterator;
    typedef typename mark_list_array::iterator mark_list_array_iterator;

    mark_list_allocator allocator;
    // list of allocated, but uninitialized, mark_item's
    mark_list free_list;
    // run-members
    mark_list member_list;
    // run-leaders
    mark_list leader_list;
    // fellow_list_array[i] contains list of all fellows of height i
    mark_list_array fellow_list_array;
    // chairperson_list elements represent a fellow_list_array[i] that has size > 1
    mark_list chairperson_list;

public:
    mark_store(allocator_type const& a)
        : allocator(a), member_list(a), leader_list(a),
          fellow_list_array(a), chairperson_list(a)
    {
        init();
    }

    ~mark_store()
    {
        rem_all_fellow_lists();
    }

    // unmarks a node through either run or club transformations
    void reduce(comparator_type const& comparator, heap_store_type& heap_store)
    {
        if (!leader_list.empty()) // leader exists => run transformation
        {
            node_type* leader = leader_list.front().node;
            node_type* leader_parent = leader->parent;
            if (leader == leader_parent->right) // is leader a right child?
            {
                bool is_parent_marked = leader_parent->is_marked();
                parent_trans(leader_parent, comparator, heap_store, *this);
            }
        }
    }
};

```

```

if (!is_parent_marked && leader->is_marked())
{
    // no nodes were unmarked
    // leader_parent is now leader->right and has leaders
    // former member-list at leader_parent->right
    if (leader_parent->left->is_marked())
    {
        // sibling is marked - a sibling transformation and we are done
        sibling_trans(leader_parent, comparator, heap_store, *this);
        return;
    }
    // (else) apply the parent transformation to leaders former parent
    // which is now leaders right child and has leaders former member(s)
    // as left children
    node_type* leader_parent_right = leader_parent->right;
    parent_trans(leader_parent, comparator, heap_store, *this);
    // leader_parent is now leader_parent_right->right
    if (leader_parent_right->is_marked())
    {
        // a parent transformation and we are done as both leader
        // and its right child (i.e. leader_parent_right) are marked
        parent_trans(leader, comparator, heap_store, *this);
    }
}
}
else // leader is a left child => leader's parent is unmarked
{
    node_type* sibling = leader_parent->right;
    if (sibling->is_marked())
    {
        sibling_trans(leader_parent, comparator, heap_store, *this);
        return;
    }
    cleaning_trans(leader_parent, *this);
    // leader_parent->left = sibling
    // leader_parent->right = leader
    // sibling->left = leader's member list
    if (sibling->right->is_marked())
    {
        sibling_trans(sibling, comparator, heap_store, *this);
        return;
    }
    cleaning_trans(sibling, *this);
    // sibling->right is now leader's member list
    parent_trans(sibling, comparator, heap_store, *this);
    // either an unmark was performed
    // (in which case leader_parent->left is marked) or
    // leader_parent is parent of two marked siblings
    if (leader_parent->left->is_marked())
    {
        sibling_trans(leader_parent, comparator, heap_store, *this);
    }
}
}
else if (!chairperson_list.empty()) // no leaders exist
{
    // chairperson exists => club transformation
    node_type* first = chairperson_list.front().node;
    node_type* first_parent = first->parent;
    bool is_first_left = first_parent->left == first;
    if (is_first_left ? first_parent->right->is_marked()
        : first_parent->left->is_marked())
    {
        sibling_trans(first_parent, comparator, heap_store, *this);
        return;
    }
    // first's sibling was not marked
    node_type* second = (++fellow_list_array[first->height]->begin())->node;
    node_type* second_parent = second->parent;
    bool is_second_left = second_parent->left == second;
    if (is_second_left ? second_parent->right->is_marked()
        : second_parent->left->is_marked())
    {
        // second's sibling marked
        sibling_trans(second_parent, comparator, heap_store, *this);
        return;
    }
    // second's sibling was not marked
    // make sure that first and second are right children
    if (is_first_left)
    {
        cleaning_trans(first_parent, *this);
    }
}
}

```

```

    }
    if (is_second_left)
    {
        cleaning_trans(second_parent, *this);
    }
    // check if parents are marked or are roots
    // if so - remove marking via parent transformation
    if (first_parent->is_marked() || first_parent->is_root())
    {
        parent_trans(first_parent, comparator, heap_store, *this);
        return;
    }
    if (second_parent->is_marked() || second_parent->is_root())
    {
        parent_trans(second_parent, comparator, heap_store, *this);
        return;
    }
    // both parents are unmarked - proceed with pair transformation
    pair_trans(first_parent, second_parent, comparator, *this);
}
}
void unmark(node_type* n)
{
    if (!n->is_marked())
    {
        return;
    }
    switch(n->mark_info->type)
    {
    case FELLOW:
    {
        mark_list_iterator iter = n->mark_info;
        mark_list& fellow_list = *fellow_list_array[n->height];
        free_list.splice(free_list.end(), fellow_list, iter);
        // check if fellow list is now of size 1
        // - if so remove chairperson_list element
        if (has_size_one(fellow_list))
        {
            iter = fellow_list.begin();
            mark_list_iterator q = iter->node->mark_info;
            free_list.splice(free_list.end(), chairperson_list, q);
            iter->node->mark_info = iter;
        }
        break;
    }
    case CHAIRPERSON:
    {
        // remove from fellow list and if list at fellow_list_array[n->height]
        // was of size 2 before operation then remove chairperson-element
        // from chairperson list
        mark_list_iterator iter = n->mark_info;
        mark_list& fellow_list = *fellow_list_array[n->height];
        if (has_size_two(fellow_list))
        {
            free_list.splice(free_list.end(), chairperson_list, iter);
        }
        else
        {
            // update the node that will be the new head of the fellow_list
            mark_list_iterator new_head = ++fellow_list.begin();
            new_head->node->mark_info = iter;
            iter->node = new_head->node;
        }
        // a chairperson node is always represented at the head of the fellow_list
        iter = fellow_list.begin();
        free_list.splice(free_list.end(), fellow_list, iter);
        break;
    }
    case LEADER:
    {
        (n->left->left->is_marked()) ? move_run_to_run(n->left, LEADER)
            : move_run_to_club(n->left, MEMBER);
        mark_list_iterator iter = n->mark_info;
        free_list.splice(free_list.end(), leader_list, iter);
        break;
    }
    case MEMBER:
    {
        // let n1,n2,...,ni,...,nn be the run and n = ni
        // if i < 3 then no new leader will be created for n1,n2,...,ni-1
        // if i = n-1 then the leader nn becomes a fellow
        node_type* n_left = n->left; // ni-1
    }
}

```

```

    if (n->left->is_marked())
    {
        (n->left->left->is_marked()) ? move_run_to_run(n->left, LEADER)
                                   : move_run_to_club(n->left, MEMBER);
    }
    if (n->parent->mark_info->type == LEADER)
    {
        // nn becomes a fellow
        move_run_to_club(n->parent, LEADER);
    }
    break;
}
}
n->clear_mark();
}

void mark(node_type* n)
{
    if (n->is_marked())
    {
        return;
    }
    bool is_fellow = true;
    node_type* n_left = n->left;
    node_type* n_parent = n->parent;
    // if n is left child of a marked parent then n becomes a member
    if (n_parent->is_marked() && n_parent->left == n)
    {
        // case: n is new MEMBER
        is_fellow = false;
        add_member(n);
        // if the marked parent was a fellow or a chairperson then make it leader
        // (cannot have been a leader as its left child was unmarked)
        if (n_parent->mark_info->type != MEMBER)
        {
            move_club_to_run(n_parent, LEADER);
        }
    }
    // check if n_left must now be moved to member list
    if (n_left->is_marked())
    {
        // n's left child is either a fellow/chairperson or a leader
        // (cannot have been a member as its parent was unmarked)
        // if is_fellow was set to false, then n is a member, otherwise
        // it is now a leader
        if (is_fellow)
        {
            // case: n is a new LEADER
            add_leader(n);
        }
        is_fellow = false;
        (n_left->mark_info->type == LEADER) ? move_run_to_run(n_left, MEMBER)
                                           : move_club_to_run(n_left, MEMBER);
    }
    if (is_fellow)
    {
        // case: n is a new FELLOW or chairperson
        add_fellow(n);
    }
}

// assumes: other.fellow_list_array.size() <= fellow_list_array.size()
void meld(mark_store& other, comparator_type const& comparator,
          heap_store_type& heap_store)
{
    size_type new_markings(0);
    // step 1: meld members
    // make members from other point to this member list
    mark_list_iterator iter = other.leader_list.begin();
    mark_list_iterator _end = other.leader_list.end();
    node_type* n;
    for (; iter != _end; ++iter)
    {
        n = iter->node->left;
        add_member(n);
        ++new_markings;
        n = n->left;
        for (; n->is_marked(); n = n->left)
        {
            add_member(n);
            ++new_markings;
        }
    }
}

```

```

    }
  }
  // step 2: meld leaders
  new_markings += other.leader_list.size();
  leader_list.splice(leader_list.end(), other.leader_list);
  // step 3: meld fellows and chairpersons
  mark_list_array_iterator a_iter = other.fellow_list_array.begin();
  mark_list_array_iterator a_end = other.fellow_list_array.end();
  for (; a_iter != a_end; ++a_iter)
  {
    iter = (*a_iter)->begin();
    _end = (*a_iter)->end();
    for (; iter != _end; ++iter)
    {
      add_fellow(iter->node);
      ++new_markings;
    }
  }
  if (new_markings == 0)
  {
    ++new_markings;
  }
  while (--new_markings > 0)
  {
    reduce(comparator, heap_store);
  }
  other.fellow_list_array.clear();
  other.chairperson_list.clear();
}

void expand()
{
  add_fellow_list();
  // TODO: must make two as chairpersons actually use two elements
  add_free_list_element();
  add_free_list_element();
}

void contract()
{
  rem_fellow_list();
  rem_free_list_element();
}

void clear()
{
  rem_all_fellow_lists();
  leader_list.clear();
  chairperson_list.clear();
}

protected: // helpers

void init()
{
  mark_info_type e = {&node_type::sentinel, MEMBER};
  member_list.push_back(e);
}

void rem_all_fellow_lists()
{
  while (!fellow_list_array.empty())
  {
    mark_list* nl = fellow_list_array.back();
    fellow_list_array.pop_back();
    nl->~mark_list();
    allocator.deallocate(nl, 1);
  }
}

void add_fellow_list()
{
  mark_list* nl = allocator.allocate(1);
  new (nl) mark_list(allocator);
  try
  {
    fellow_list_array.push_back(nl);
  }
  catch (...)
  {
    nl->~mark_list();
    allocator.deallocate(nl, 1);
    throw;
  }
}

```

```

}
void add_free_list_element()
{
    mark_info_type e = {&node_type::sentinel, LEADER};
    free_list.push_back(e);
}
void rem_fellow_list()
{
    mark_list* nl = fellow_list_array.back();
    fellow_list_array.pop_back();
    nl->mark_list();
    allocator.deallocate(nl, 1);
}
void rem_free_list_element()
{
    free_list.pop_back();
}

inline
mark_list_iterator get_mark_item(node_type* n, mark_type t)
{
    mark_list_iterator it = free_list.begin();
    it->node = n;
    it->type = t;
    return it;
}

inline
void add_leader(node_type* n)
{
    mark_list_iterator iter = get_mark_item(n, LEADER);
    leader_list.splice(leader_list.end(), free_list, iter);
    n->mark_info = iter;
}
inline
void add_member(node_type* n)
{
    n->mark_info = member_list.begin();
}
void add_fellow(node_type* n)
{
    mark_list& fellow_list = *fellow_list_array[n->height];
    if (has_size_one(fellow_list))
    {
        mark_list_iterator iter = get_mark_item(n, CHAIRPERSON);
        chairperson_list.splice(chairperson_list.end(), free_list, iter);
        n->mark_info = iter;
        iter = get_mark_item(n, FELLOW);
        fellow_list.splice(fellow_list.begin(), free_list, iter);
    }
    else
    {
        mark_list_iterator iter = get_mark_item(n, FELLOW);
        fellow_list.splice(fellow_list.end(), free_list, iter);
        n->mark_info = iter;
    }
}

inline
void move_to_free(node_type* n, mark_list& from_list)
{
    free_list.splice(free_list, from_list, n->mark_info);
    n->clear_mark();
}
inline
void move_club_to_run(node_type* n, mark_type t)
{
    n->mark_info->type == CHAIRPERSON ? move_chairperson_to_run(n, t)
                                     : move_fellow_to_run(n, t);
}
// move n from fellow list to either leader or member list
void move_fellow_to_run(node_type* n, mark_type t)
{
    mark_list& fellow_list = *fellow_list_array[n->height];
    if (has_size_two(fellow_list))
    {
        // move chairperson-list element to free_list
        mark_list_iterator p = fellow_list.begin();
        mark_list_iterator q = p->node->mark_info;
        free_list.splice(free_list.end(), chairperson_list, q);
        // set node that pointed to chairperson-list to point to fellow list
        p->node->mark_info = p;
    }
}

```

```

    }
    // move n to new list
    move_to_run(n, t, fellow_list);
}
void move_chairperson_to_run(node_type* n, mark_type t)
{
    mark_list& fellow_list = *fellow_list_array[n->height];
    mark_list_iterator iter = n->mark_info;
    if (has_size_two(fellow_list))
    {
        // no one to take over chairperson-list element
        // - move chairperson-list element to free list
        free_list.splice(free_list.end(), chairperson_list, iter);
    }
    else
    {
        // promote next fellow element to chairperson object
        node_type* new_chairperson = (++fellow_list.begin())->node;
        new_chairperson->mark_info = iter;
        iter->node = new_chairperson;
    }
    // move n to new list
    n->mark_info = fellow_list.begin();
    move_to_run(n, t, fellow_list);
}
inline
void move_to_run(node_type* n, mark_type t, mark_list& from_list)
{
    if (t == LEADER)
    {
        n->mark_info->type = t;
        leader_list.splice(leader_list.end(), from_list, n->mark_info);
    }
    else
    {
        free_list.splice(free_list.end(), from_list, n->mark_info);
        add_member(n);
    }
}
void move_leader_to_club(node_type* n)
{
    mark_list& fellow_list = *fellow_list_array[n->height];
    mark_list_iterator q = n->mark_info;
    if (has_size_one(fellow_list))
    {
        // create new chairperson
        mark_list_iterator p = get_mark_item(n, CHAIRPERSON);
        chairperson_list.splice(chairperson_list.end(), free_list, p);
        n->mark_info = p;
        fellow_list.splice(fellow_list.begin(), leader_list, q);
    }
    else
    {
        fellow_list.splice(fellow_list.end(), leader_list, q);
    }
    q->type = FELLOW;
}
inline
void move_run_to_club(node_type* n, mark_type t)
{
    if (t == MEMBER)
    {
        add_fellow(n);
    }
    else //(t == LEADER)
    {
        move_leader_to_club(n);
    }
}
inline
void move_run_to_run(node_type* n, mark_type t)
{
    if (t == LEADER)
    {
        add_leader(n);
    }
    else
    {
        free_list.splice(free_list.end(), leader_list, n->mark_info);
        add_member(n);
    }
}

```

```

}

// returns whether n_parent_fellow_list has size == 1
// can't do a size() on lists as these may have linear complexity!
inline
bool has_size_one(mark_list& n_list)
{
    mark_list_iterator iter = n_list.begin();
    mark_list_iterator _end = n_list.end();
    return (iter != _end) && (++iter == _end);
}

// returns whether n_parent_fellow_list has size == 2
// can't do a size() on lists as these may have linear complexity!
inline
bool has_size_two(mark_list& n_list)
{
    mark_list_iterator iter = n_list.begin();
    mark_list_iterator _end = n_list.end();
    return (iter != _end) && (++iter != _end) && (++iter == _end);
}

// returns whether n_parent_fellow_list has size >= 2
// can't do a size() on lists as these may have linear complexity!
inline
bool is_club(mark_list& n_list)
{
    mark_list_iterator iter = n_list.begin();
    mark_list_iterator _end = n_list.end();
    return (iter != _end) && (++iter != _end);
}

#ifdef __DEBUG__
// useful for debugging
public:
    bool _validate()
    {
        bool is_valid = true;
        // CHAIRPERSONS
        mark_list_iterator iter = chairperson_list.begin();
        mark_list_iterator _end = chairperson_list.end();
        for (; iter != _end; ++iter)
        {
            is_valid = is_valid && iter->type == CHAIRPERSON;
            node_type* n = iter->node;
            is_valid = is_valid && n->mark_info == iter;
            is_valid = is_valid && fellow_list_array[n->height]->front().node == n;
        }
        // RUNS
        is_valid = is_valid && member_list.front().type == MEMBER;
        iter = leader_list.begin();
        _end = leader_list.end();
        for (; iter != _end; ++iter)
        {
            node_type* n = iter->node;
            is_valid = is_valid && iter->type == LEADER;
            is_valid = is_valid && n->mark_info == iter;
            do
            {
                is_valid = is_valid && n->left->mark_info == member_list.begin();
                is_valid = is_valid && iter->node->mark_info == iter;
                n = n->left;
            }
            while(n->left != &node_type::sentinel && n->left->is_marked());
        }
        mark_list_array_iterator a_iter = fellow_list_array.begin();
        mark_list_array_iterator a__end = fellow_list_array.end();
        for (; a_iter != a__end; ++a_iter)
        {
            iter = (*a_iter)->begin();
            _end = (*a_iter)->end();
            if (iter != _end && is_club(**a_iter))
            {
                is_valid = is_valid && iter->type == FELLOW;
                is_valid = is_valid && iter->node->mark_info->type == CHAIRPERSON;
                is_valid = is_valid && iter->node->mark_info != iter;
                is_valid = is_valid && iter->node->mark_info->node == iter->node;
                ++iter;
            }
        }
        for (; iter != _end; ++iter)
        {
            is_valid = is_valid && iter->type == FELLOW;
            is_valid = is_valid && iter->node->mark_info == iter;
        }
    }
}

```

```

    }
    return is_valid;
}
#endif // __DEBUG__
};

} // namespace cphstl
#endif // __CPHSTL_MARK_STORE__

```

D. node.h++

```

/* Node used by relaxed_weak_queue
 *
 * Uses a static sentinel node as a null value - usefull for avoiding
 * special null cases when only checking height or node_store.
 *
 */

#ifndef __CPHSTL_NODE__
#define __CPHSTL_NODE__

#include <node_base.h++>

namespace cphstl {

template <
typename V,
typename A
>
class node : public node_base<node<V, A>, A>
{
public:
    typedef size_t size_type;
    typedef V value_type;
    typedef typename node_base<node, A>::mark_list_iterator mark_list_iterator;
public:
    value_type value;
    size_type height;
    // mark_info = null <=> node is unmarked
    // else mark_info is iterator of list of its marked type
    mark_list_iterator mark_info;

    node* parent; // parent
    node* left; // left child
    node* right; // right child
    node* prev; // previous node in iteration
    node* next; // next node in iteration

    // for efficiency reasons left, parent (root list), next and prev
    // (for iteration) are not initialized as they are assigned later,
    // before ever being read
    node<V, A>()
        : height(0), mark_info(mark_list_iterator(0)), right(&sentinel)
    {}

    node<V, A>(value_type const& v)
        : value(v), height(0), mark_info(mark_list_iterator(0)), right(&sentinel)
    {}

    node<V, A>(size_type h, node* n)
        : height(h), mark_info(mark_list_iterator(0)), parent(n), left(n), right(n),
          prev(n), next(n)
    {}

    inline
    bool is_root() const
    {
        return height >= parent->height;
    }
    inline
    bool is_marked() const
    {
        return mark_info != nil_mark_info;
    }
    inline
    void clear_mark()
    {
        mark_info = nil_mark_info;
    }
    static node<V, A> sentinel;
    static mark_list_iterator nil_mark_info;
};

```

```

};

template <typename V, typename A>
node<V, A> node<V, A>::sentinel(0, &node<V, A>::sentinel);

template <typename V, typename A>
typename node<V, A>::mark_list_iterator node<V, A>::nil_mark_info(0);

} // namespace cphstl
#endif // _CPHSTL_NODE_

```

E. types.h++

```

#ifndef _CPHSTL_TYPES_
#define _CPHSTL_TYPES_

#include <cstddef>

namespace cphstl {
typedef size_t size_type;

enum mark_type {FELLOW, CHAIRPERSON, MEMBER, LEADER};

template<
typename N, // node type
typename T // node mark type
>
struct mark_info
{
    N* node;
    T type;
};

} // namespace cphstl
#endif // _CPHSTL_TYPES_

```

F. node_base.h++

```

/* Using CRTP to declare the two mutually dependent types:
 * node and mark_info, by using it as base for node
 */

#ifndef _CPHSTL_NODE_BASE_
#define _CPHSTL_NODE_BASE_

#include <list> // std::list
#include <types.h++> // mark_type, mark_info

namespace cphstl {
template <
typename N,
typename A
>
class node_base
{
public:
    typedef mark_info<N, mark_type> mark_info_type;
    typedef typename A::template rebind<mark_info_type>::other
        mark_info_allocator;
    typedef typename std::list<mark_info_type, mark_info_allocator> mark_list;
    typedef typename mark_list::iterator mark_list_iterator;
};

} // namespace cphstl
#endif // _CPHSTL_NODE_BASE_

```

G. helpers.h++

```

/* Helpers for accessing properties of heaps
 *
 * */
#ifndef _CPHSTL_HELPERS_
#define _CPHSTL_HELPERS_

namespace cphstl {

template <typename node_type>
inline
void upd_root_list_left(node_type* p, node_type* q)
{
    p->parent = q->parent;
    if (p->parent != &node_type::sentinel)
    {
        p->parent->left = p;
    }
}

template <typename node_type>
inline
void upd_root_list_right(node_type* p, node_type* q)
{
    p->left = q->left;
    if (p->left != &node_type::sentinel)
    {
        p->left->parent = p;
    }
}

// safe version of set_right_child
template <typename node_type>
inline
void safe_set_right_child(node_type* p, node_type* r)
{
    p->right = r;
    if (r != &node_type::sentinel)
    {
        r->parent = p;
    }
}

// safe version of set_left_child
template <typename node_type>
inline
void safe_set_left_child(node_type* p, node_type* l)
{
    p->left = l;
    if (l != &node_type::sentinel)
    {
        l->parent = p;
    }
}

// assumes that r and l != sentinel
template <typename node_type>
inline
void set_children(node_type* p, node_type* l, node_type* r)
{
    set_left_child(p, l);
    set_right_child(p, r);
}

// assumes that r != sentinel
template <typename node_type>
inline
void set_right_child(node_type* p, node_type* r)
{
    p->right = r;
    r->parent = p;
}

// assumes that l != sentinel
template <typename node_type>
inline
void set_left_child(node_type* p, node_type* l)
{
    p->left = l;
    l->parent = p;
}
}

```



```

inline
void split(node_type* p, split_pair& sp)
{
    sp.is_right_child = p->parent->right == p;
    sp.first = p;
    sp.second = p->right;
    sp.parent = p->parent;
    sp.left = p->left;
    --sp.first->height;
    safe_set_right_child(sp.first, sp.second->left);
    sp.second->left = sp.second->parent = &node_type::sentinel;
}

// returns the resulting root
template <typename node_type, typename comparator>
inline
node_type* join(node_type* p, node_type* q, comparator const& comp)
{
    return
        comp(q->value, p->value) ? do_join(p, q)
        : do_join(q, p);
}

template <typename node_type>
inline
node_type* do_join(node_type* p, node_type* q)
{
    safe_set_left_child(q, p->right);
    set_right_child(p, q);
    ++p->height;
    return p;
}

template <typename node_type>
void join_split_pair(split_pair<node_type>& sp)
{
    safe_set_left_child(sp.second, sp.first->right);
    set_right_child(sp.first, sp.second);
    ++sp.first->height;

    sp.first->parent = sp.parent;
    sp.first->left = sp.left;
    (sp.is_right_child ? sp.parent->right
        : sp.parent->left) = sp.first;
    if (sp.left != &node_type::sentinel)
    {
        sp.left->parent = sp.first;
    }
}

/*
 *
 *      p
 *     / \
 *    [q]  s
 *   / \ / \
 *  A  B C  D   ->   A  s  [q]
 *                   / \ / \
 *                  A  D C  B
 *
 * */
template<typename node_type, typename mark_store_type>
inline
void cleaning_trans(node_type* p, mark_store_type& mark_store)
{
    node_type* s = p->right;
    node_type* q = p->left;
    mark_store.unmark(q);
    std::swap(p->left, p->right);
    if (q->height > 0)
    {
        std::swap(s->left, q->left);
        s->left->parent = s;
        q->left->parent = q;
    }
    mark_store.mark(q);
}

/*
 *
 *      (p)
 *     / \
 *    A  [q]
 *       / \
 *      B  C
 *
 *      ->
 *
 *      (p)
 *     / \
 *    A  q
 *       / \
 *      B  C
 *
 *      OR
 *
 *      [q]
 *     / \
 *    A  p
 *       / \
 *      C  B
 *
 * */
template<typename node_type, typename comp_type, typename mark_store_type,
         typename heap_store_type>
node_type* parent_trans(node_type* p, comp_type const& comparator,

```



```

    }
    // step 3 - do a parent transformation
    mark_store.mark(q);
    return parent_trans(q, comparator, heap_store, mark_store);
}

// helper for pair_trans
template<typename node_type, typename comp_type, typename mark_store_type>
inline
void do_pair_trans(split_pair<node_type>& p_surr, split_pair<node_type>& r_surr,
                  comp_type const& comparator, mark_store_type& mark_store)
{
    std::swap(p_surr.second, r_surr.first);
    if (comparator(r_surr.first->value, r_surr.second->value))
    {
        std::swap(r_surr.first, r_surr.second);
    }
    // this order is essential to prevent p from overwriting q if they are
    // root-neighbours
    join_split_pair<node_type>(r_surr);
    do_join(p_surr.first, p_surr.second);
    mark_store.mark(r_surr.first);
}
/*
 *
 *      p           r           p           r           [q]           [s]
 *      \          /          \          /          \          /          \
 *      [q]       [s]  ->   r       OR   p       ,   s       OR   q
 *      / \      / \      / \      / \      / \      / \
 *     A  B    C  D    A  C    C  A    B  D    D  B
 *
 * - assumptions:
 * - p != r
 * - height(q) = height(s)
 * - neither p nor r are roots
 */
template<typename node_type, typename comp_type,
         typename mark_store_type>
inline
void pair_trans(node_type* p, node_type* r, comp_type const& comparator,
               mark_store_type& mark_store)
{
    {
        mark_store.unmark(p->right);
        mark_store.unmark(r->right);
        split_pair<node_type> p_surr;
        split_pair<node_type> r_surr;
        split(p, p_surr);
        split(r, r_surr);
        if (comparator(r->value, p->value))
        {
            do_pair_trans<node_type, comp_type, mark_store_type>
                (p_surr, r_surr, comparator, mark_store);
        }
        else
        {
            do_pair_trans<node_type, comp_type, mark_store_type>
                (r_surr, p_surr, comparator, mark_store);
        }
    }
}

// assumes that p is not root and p and p's left child are unmarked
template <typename node_type>
inline
void parent_left_swap(node_type* p)
{
    {
        node_type* q = p->left;
        --p->height;
        ++q->height;
        q->parent = p->parent;
        (p == p->parent->right ? q->parent->right : q->parent->left) = q;
        // children
        std::swap(q->right, p->right);
        q->right->parent = q;
        p->left = q->left;
        if (p->height > 0) // i.e. > 1 before operation
        {
            p->right->parent = p->left->parent = p;
        }
        set_left_child(q, p);
    }
}

// assumes that p and p->left are equal-sized roots of perfect weak heaps
template <typename node_type, typename comparator>
inline

```

```

node_type* join_root_pair(node_type* p, comparator const& comp)
{
    node_type* q = p->left; // next root to the right
    if(comp(q->value, p->value)) // p is root
    {
        upd_root_list_right(p, q);
        do_join_roots(p, q);
        return p;
    }
    else // q is root
    {
        upd_root_list_left(q, p);
        do_join_roots(q, p);
        return q;
    }
}

template <typename node_type>
inline
void do_join_roots(node_type* p, node_type* q)
{
    q->parent = p;
    q->left = p->right;
    if (p->right != &node_type::sentinel)
    {
        p->right->parent = q;
    }
    p->right = q;
    ++p->height;
}

} // namespace cphstl
#endif // _CPHSTL_TRANSFORMATIONS_

```

I. bidirectional_iterator.h++

```

/* Adaption of Jyrki Katajainens bidirectional_iterator
 * for use with relaxed_weak_queue
 *
 * */

#ifndef _CPHSTL_BIDIRECTIONAL_ITERATOR_
#define _CPHSTL_BIDIRECTIONAL_ITERATOR_

#include <iterator> // std::bidirectional_iterator_tag
#include <traits.h++> // if_then_else
#include <relaxed_weak_queue.h++> // relaxed_weak_queue

namespace cphstl {

// forward declaration
template<typename V, typename C, typename A,
        template <typename, typename> class S>
class relaxed_weak_queue;

template <
    typename V, // value
    typename N, // node
    // (can't pass rwqueue, so pass parameters instead, see 11.4§2 in the standard)
    typename C, // rwqueue's comparator
    typename A, // rwqueue's allocator
    template <typename, typename> class S, // rwqueue's sequence
    bool is_const = true // is constant iterator?
>
class bidirectional_iterator {
    friend class bidirectional_iterator<V, N, C, A, S, !is_const>;
    friend class relaxed_weak_queue<V, C, A, S>;
protected:
    typedef N node_type;
    typedef C container_type;
public:
    typedef std::bidirectional_iterator_tag iterator_category;
    typedef V value_type;
    typedef
    typename relaxed_weak_queue<V, C, A, S>::difference_type difference_type;
    typedef typename if_then_else<is_const, value_type const*, value_type*>::type
        pointer;
    typedef typename if_then_else<is_const, value_type const&, value_type&>::type
        reference;

    bidirectional_iterator() : position(0)

```

```

{}

bidirectional_iterator(node_type* p) : position(p)
{}

bidirectional_iterator(bidirectional_iterator<V, N, C, A, S> const& b_i)
: position(b_i.position)
{}

~bidirectional_iterator()
{}

reference operator*() const
{
    return position->value;
}

pointer operator->() const
{
    return &(position->value);
}

bidirectional_iterator& operator++()
{
    position = position->next;
    return (*this);
}

bidirectional_iterator operator++(int)
{
    bidirectional_iterator temp = *this;
    ++(*this);
    return temp;
}

bidirectional_iterator& operator--()
{
    position = position->prev;
    return (*this);
}

bidirectional_iterator operator--(int)
{
    bidirectional_iterator temp = *this;
    --(*this);
    return temp;
}

template <bool both>
bool operator==(bidirectional_iterator<V, N, C, A, S, both> const& b_i) const
{
    return (*this).position == b_i.position;
}

template <bool both>
bool operator!=(bidirectional_iterator<V, N, C, A, S, both> const& b_i) const
{
    return (*this).position != b_i.position;
}

private:
    node_type* position;
};

} // namespace cphstl

#endif // _CPHSTL_BIDIRECTIONAL_ITERATOR_

```

J. traits.h++

```

#ifndef __CPHSTL_TRAITS__
#define __CPHSTL_TRAITS__

namespace cphstl {

    template <bool, typename T, typename U>
    class if_then_else;

    template <typename T, typename U>
    class if_then_else<true, T, U> {
    public:
        typedef T type;
    };

```

```
};  
  
template <typename T, typename U>  
class if_then_else <false, T, U> {  
public:  
    typedef U type;  
};  
  
template <typename X, typename Y>  
class types {  
public:  
    enum { are_same = 0 };  
};  
  
template <typename X>  
class types <X, X> {  
public:  
    enum { are_same = 1 };  
};  
  
} // namespace cphstl  
#endif // __CPHSTL_TRAITS__
```

References

- [1] British Standards Institute, *The C++ Standard: Incorporating Technical Corrigendum 1*, 2nd Edition, John Wiley and Sons, Ltd. (2003).
- [2] Department of Computing, University of Copenhagen, The CPH STL, Website accessible at <http://www.cphstl.dk/> (2000–2008).
- [3] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation, *Commun. ACM* **31**, 11 (1988), 1343–1354.
- [4] R. D. Dutton, Weak-heap sort, *BIT* **33**, 3 (1993), 372–381.
- [5] A. Elmasry, C. Jensen, and J. Katajainen, A framework for speeding up priority-queue operations, CPH STL Report 2004-3, Department of Computing, University of Copenhagen (2004).
- [6] A. Elmasry, C. Jensen, and J. Katajainen, Relaxed Weak Queues: An Alternative to Run-Relaxed Haps, CPH STL Report 2005-2, Department of Computing, University of Copenhagen (2005).
- [7] J. Katajainen, Project proposal: A meldable, iterator-valid priority queue, CPH STL Report 2005-1, Department of Computing, University of Copenhagen (2005).
- [8] M. D. Kristensen, A Vector Implementation for the CPH STL, CPH STL Report 2004-2, Department of Computing, University of Copenhagen (2004).
- [9] K. Mehlhorn and S. Näher, Leda: a platform for combinatorial and geometric computing, *Commun. ACM* **38**, 1 (1995), 96–102.
- [10] T. Takaoka, Theory of 2-3 heaps, *Computing and Combinatorics* (1999), 41–50.
- [11] Valgrind Developers, Valgrind, Website accessible at <http://valgrind.org/> (2002–2008).