Extending the CPH STL with LEDA APIs

Michael Neidhardt* Bo Simonsen†

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

Abstract. Program libraries have become a central part of modern application development. They have become the central toolbox consisting of ready-to-use algorithmic components which most programmers use frequently. The CPH STL is an enhanced version of the well-known Standard Template Library (STL), which is part of the C++ standard and thereby provided by all C++ compilers. In the algorithm community LEDA is a well-known and respected library, providing the same containers and algorithms as the STL but also more advanced containers and algorithms. In this project we describe how, with a small amount of code, we can easily provide LEDA interfaces for our existing STL components. We have designed a number of adaptors, which existing programs relying on LEDA can use. This has been possible because of the CPH STL architecture. In this work we describe the relevant techniques which can be used as a foundation for the future development of LEDA interfaces for the CPH STL.

1. Introduction

The CPH STL [7] is a project which aims to produce an enhanced version of the Standard Template Library (STL). The STL has become part of the C++ standard [4] where the requirements of the different containers and algorithms are specified. LEDA [1] is an established library of efficient data structures and algorithms. Dressing up the CPH STL as LEDA is what this project is about. More specifically the aim is to create interfaces for a number of CPH STL containers, in a manner that will let any existing LEDA program swap the relevant native LEDA container for one from the CPH STL.

The architecture of the CPH STL [12][1] allows us to create several container interfaces which use the same underlying implementation of a data structure. We denote such an interface a *container*, and an implementation of a data structure a *realizator*. We extended the decoupling by seperating the storage mechanisms and the realizators. We introduced *encapsulators* that are small objects, purpose of which is to encapsulate each value stored in the realizator. Encapsulators are known from the introductory textbooks on data structures as nodes. For example, a binary search tree uses a node

*E-mail: `meem.ok@gmail.com`.
†E-mail: `bosim@diku.dk`.
[1] This paper can be found in [16].

CPHSTL Report 2009-8, March 2009, Revised April 2009 and November 2009

class to store each value which is stored in the tree. We did also decouple the iterators from the containers such that they should be given to the containers as template arguments. The realizator uses ordinary pointers to locate the elements; these pointers are denoted *concrete iterators*.

LEDA and the STL are similar in many ways. They are both template based and they provide almost the same containers; even the operations of the different containers are almost the same. The main difference between the two libraries is that LEDA does not use iterators for locating and traversing the data stored in the containers. Instead *items* are used which are pointers to encapsulators (in our terminology, concrete iterators) to locate the elements, and member functions in the containers are used for traversal. Since the construction of the LEDA containers is similar to the construction of the CPH STL realizators our hypothesis is that it should be easy to add LEDA interfaces which use the CPH STL realizators.

Example 1. Consider a LEDA 5.0 dictionary `d`, the elments can be traversed in the following way.

```
1 for (dic_item p = d.first(); p != nil; p = d.succ(p)) {
2   /* d.key(p) gives the key, and d.inf(p) gives the associated
        information. */
3 }
```

□

Why is it interesting to create these interfaces for LEDA? The main thing is probably the learning aspect for us, i.e. getting familiar with patterns and library development. Apart from that, the abstractions made in this context will offer the ability to use CPH STL containers in programs relying on LEDA, to make benchmarking easier and to change the underlying realizator for a number of structures. It can be useful if a library user is able to choose between different structures with different behaviour concerning space and time efficiency.

In this report we discuss issues regarding the development of interfaces for library containers. The interfaces are `leda::stack`, `leda::queue`, `leda::dictionary`, `leda::array`, and `leda::p_queue`. The code can be found in the appendix.

2. Requirements

The primary source for requirements in this project is the LEDA specifications, but it is also worth mentioning that libraries such as *Boost* [3] and CGAL [6] have a number of guidelines for writing library code. A development method such as XP [2] suggests using style guides. It might also be worthwhile taking a look at invariants, if any are available in the documentation. Another good idea is to make sure there are usable and useful default values for template parameters. As an example, traits or policy classes are sometimes perceived as peripheral to a task, but are required by the library in certain cases. It can be very helpful to the library user if sensible defaults

are given. Needless to say, it is also a good idea to document in sufficient detail how to create more elaborate solutions. An example can be found in the 2D Triangulation section of CGAL [6] — this describes Delaunay triangulation using two different point-location algorithms.

We will consider the following when specifying the requirements for the library code:

- Functional specifications
- Test requirements
- Design patterns
- Guarantees for structures (containers).

2.1 Functional specifications

The functional specifications are fixed and given in the LEDA manual, i.e. constructors, destructors, functions, operators and macros. The LEDA manual [1] lists specifics of the abstraction. The LEDA book (the free online version) [15] lists general rules. A central rule is LEDA rule 14 [15, Ch. 2], which states that any actual type argument must provide the following six functions:

1. A default constructor
2. A copy constructor
3. An assignment operator
4. A read function
5. A print function
6. A destructor.

In addition, linearly ordered types must provide a compare function, a hashed type must provide a hash function and an equality operator, and a numerical type (e.g. a multiprecision number type) must provide addition, subtraction and multiplication functions and standard comparison operators. For certain structures LEDA offers macros for iteration, e.g. dictionary and set.

2.2 Test requirements

We have used test-driven development when implementing the LEDA interfaces. The key idea in test-driven development is to start programming by making a test program, i.e. before any code of the library is written. This will actually define the (external) requirements of the program, in that the test uses all the public aspects (functions/data) of the program. Furthermore, all tests are kept and will be run in all future tests, i.e. the test suite grows. This ensures that regression to old erroneous behaviour is prevented. For the containers in question, this means at least the following:

- Test all constructors and destructors
- Test all accessors, including iterators
- Test all modifiers
- Test possible combinations of operations.

2.3 Design patterns

A seminal book [9] on the topic of patterns states that "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice". Another way of putting this is that design patterns can be seen as general solutions to common problems.[2] The main pattern used in this project is the bridge pattern [9]. The form in which it is used here can be called a generic bridge and is detailed in [12]. In this form the bridge appears by letting the implementation be a template parameter to the abstraction; according to our terminology the abstraction is the container and the implementation is the realizator. A consequence is that every function that is wanted in the abstraction must be explicitly written. This causes certain dependencies to appear; hence the solution could be said to be less generic (in the common sense of this word), in that not all implementations can be expected to have the same functions or the same names as those that are in the chosen default implementation. As stated in [9] the reason for using a bridge is to be able to let abstraction and implementation vary independently, whereas one of the main reasons for using generic programming is to be able to write one piece of software for several different types. As mentioned, dependencies will appear, but having the underlying containers come from a common library is undoubtedly an advantage, as they will have developed along similar lines of thought and design goals.

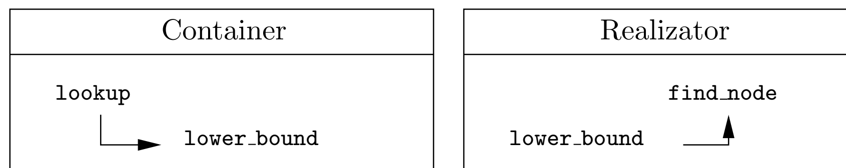

Figure 1. Detail of a bridge for the `lookup` function in `leda::dictionary`. This function finds the element with a given key.

In the CPH STL the bridge pattern has been applied as illustrated in Figure 1. The client (i.e. the user of the container) sees only the function `lookup`, the name of which is dictated by LEDA. The container calls `lower_bound` in the realizator, which in turn searches its internal structure using the function `find_node`. In this design, the two extremes (i.e. `lookup` and `find_node`) can vary independently, but the container relies on the realizator to offer a function called `lower_bound`.

In our implementation, each LEDA container provides a default template argument for the realizator such that if the user did not explicitly give the realizator a default will be used. The default realizator comes preferably

[2] Phrase from Wikipedia: `http://en.wikipedia.org/wiki/Design_pattern` (computer science).

from the CPH STL. Some of the LEDA interfaces in their original form use a fixed realizator, so in those cases this project extends LEDA.

2.4 Guarantees for library containers

We base the part on guarantees on [10], and hence there are four possible aspects:
1. Time optimality
2. Iterator validity
3. Exception safety
4. Space efficiency.

LEDA lacks iterators in many containers, so that will be left out of this project. LEDA does not guarantee exception safety, which is therefore also excluded from this project. That leaves time optimality and space efficiency. In the CPH STL, there exists at least one realizator variant, or a realizator given a specific permutation of types as template arguments, which provides these properties. Therefore we do not need to take these guarantees into account when designing the LEDA interfaces.

3. Interfaces

In this section we describe the specific requirements and the abstractions for `stack`, `queue`, `dictionary`, `array`, and `p_queue`. The interfaces of these containers are given in [1].

3.1 Stack

A stack is a fairly simple structure and it is easy to implement an abstraction for it, not least because there is already an existing abstraction `cphstl::stack` and realizator `cphstl::doubly_linked_list`. All that needs to be done is to make sure the tests cover everything, write the interface with the mentioned realization, and verify that no test fails.

LEDA guarantees that all operations take constant time, except clear which takes time linear in the number of elements. Our LEDA stack interface uses `cphstl::doubly_linked_list` as the default realizator, which supports insertion and deletion at both ends in constant time. Because of space efficiency the desirable realizator for stack would be a singly-linked list but this realizator is not available in the CPH STL yet. At the point of time when it becomes available, the default realizator in the LEDA interface should simply be changed.

Interface 1. The interface of `leda::stack` accepts the following template arguments (`V` denotes the type of the values stored and `R` denotes the type of the realizator):

```
1 namespace leda {
2   template <
3     typename V,
```

```
4     typename R = cphstl::doubly_linked_list<V>
5   >
6   class stack;
7 }
```

3.2 Queue

A queue is about as simple as a stack, and so no problems were expected here either. The only thing that caused extra work was the fact that LEDA's stack has functions to add an element to the front as well as the back of a queue. Strictly speaking this makes the structure a double-ended queue, also known as a *deque*. The obvious default realization is `cphstl::doubly_linked_list` but `cphstl::deque` is also possible. We do not have an interface for `cphstl::deque` yet and we do not have a suitable kernel for the vector framework for realizing this interface; such a kernel is illustrated in [16, p. 19]. A doubly-linked list seems more appropriate as the realizing data structure since there is no need for random access in `leda::queue`, which is the only thing that deque provides and the doubly-linked list does not.

Our implementation of `leda::queue` provides traversal mechanisms for both LEDA and the STL. Regarding the STL, a user can obtain STL compliant iterators by calling the member functions `begin()` and `end()`. As described the LEDA traversal mechanism is a bit different. Here the user gets a pointer to the node in the list instead of an iterator. In older versions of LEDA ($<$ 6.0) the user could use the member functions `first()` and `last()` to get an item (a pointer) pointing at the first and the last element respectively. The value stored at a given item can be retrieved using `inf()` and the item can be advanced using `succ`. However in LEDA 6.0, these member functions have been renamed to `first_item`, `last_item`, `next_item`. We will use the names of the member functions from LEDA 6.0 for all LEDA container interfaces. The items for `leda::queue` will be pointers to corresponding nodes stored in the realizator (`cphstl::doubly_linked_list`). The member functions in our encapsulators should be declared private, and their respective realizators and LEDA interfaces should be declared as friends to the encapsulator classes (see, for example, [5, p. 539]). The fact that the member functions in the LEDA interfaces were renamed does not really matter, since iteration in LEDA should be performed using the iteration macros.

The LEDA traversal mechanism is implemented in the following way (this is the same for all LEDA container interfaces):

```
1 #define nil 0
2
3 namespace leda {
4   template <typename V, typename R, typename I, typename J>
5   typename queue<V, R, I, J>::item
6   queue<V, R, I, J>::first_item() {
```

```
7     return (*this).kernel.begin();
8   }
9   template <typename V, typename R, typename I, typename J>
10  typename queue<V, R, I, J>::item
11  queue<V, R, I, J>::last_item() {
12    return (*(*this).kernel.end()).predecessor();
13  }
14  template <typename V, typename R, typename I, typename J>
15  typename queue<V, R, I, J>::item
16  queue<V, R, I, J>::next_item(item p) {
17    return (*p).successor();
18  }
19 }
```

As for `stack`, LEDA guarantees that all `queue` operations take constant time, except `clear`, which takes time linear in the number of elements stored.

Interface 2. The interface of `leda::queue` accepts the following template arguments (`V` denotes the type of the values stored, `R` denotes the type of the realizator, `I` denotes the type of the mutable iterator, and `J` denotes the type of the immutable iterator):

```
1 namespace leda {
2   template <
3     typename V,
4     typename R = cphstl::doubly_linked_list<V>,
5     typename I = cphstl::node_iterator<typename R::encapsulator_type
        , false>,
6     typename J = cphstl::node_iterator<typename R::encapsulator_type
        , true>
7   >
8   class queue;
9 }
```

3.3 Dictionary

Associative containers in the CPH STL are realized using the binary-search-tree framework [17]. The framework accepts a balancing policy (or a balancer), an encapsulator, and a searching policy. The LEDA `dictionary` interface has been constructed such that it uses the search-tree framework (`cphstl::tree`). The LEDA `dictionary` interface is constructed such that it uses `cphstl::tree` with no policies given in advance which means that the default template arguments of `cphstl::tree` are used. When these default template arguments are used, `cphstl::tree` provides a red-black tree using a 6-word node and guarantees iterator operations in $O(1)$ worst-case time.

The difficult part was how to handle the comparison function and how to store the data. LEDA's dictionary is a collection of items containing a key of type `K` with associated information of type `V`. These two are also the only template parameters of LEDA's `dictionary`. The elements of type `K` are linearly ordered, with the ordering coming from a compare function.

For basic types LEDA defines this function, but for others the client must supply it. Another complication here was that LEDA distinguishes three cases $(<, =, >)$ in comparisons, and returns one of three numeric values. The STL uses two $(<, \geq)$ and returns a Boolean value, so a conversion must take place if the user supplies a three-way `compare` function.

This has been solved by designing a class template which is an adaptor for a LEDA compare function. The code for this class template looks as follows:

```
1  template <typename T>
2  class stl_compare_less : public std::binary_function<T, T, bool> {
3  public:
4    stl_compare_less(int (*_f)(T const&, T const&)) {
5      (*this).f = _f;
6    }
7    bool operator()(T const& a, T const& b) const {
8      if((*this).f(a, b) > -1) {
9        return false;
10     }
11
12     return true;
13   }
14 private:
15   int (*f)(T const&, T const&);
16 };
```

The class template can be constructed using a pointer to a compare function. This is useful for the parameterized constructor since it accepts such a function. What happens when the functor is invoked is that the compare function is called and if the result is 1 ($\mathtt{a} > \mathtt{b}$) or 0 ($\mathtt{a} = \mathtt{b}$) it returns `false`; if the result is -1 ($\mathtt{a} < \mathtt{b}$) it returns `true`.

Most functions in the dictionary are fairly straightforward and can be realized with the given realizator. The non-trivial parts are using `lower_bound` correctly (for `lookup` and others), finding out how to use concrete iterators, and getting access to the key and the information when these reside inside a node of the realizator structure. LEDA dictionary has template arguments `K` and `V` for the key and information of the dictionary, respectively. Together they form a pair of type `V'`. This is the element as seen from the point of view of the realizator.

The default realization, `cphstl::tree`, has four relevant template parameters: `K, V, F, C`. Here `K` is the key type and `V` is the element type, with `K` contained within `V`. To get the key from an element, an extractor function of type `F` must be supplied. Furthermore, a compare function of type `C` comparing keys of type `K` is needed. The remaining template parameters for the default realization will not be used and will not be discussed.

Since an element is of type `std::pair<K, I>`, a simple extractor function can be implemented as follows:

```
1 template <typename K, typename I>
2 K extract(std::pair<K, I> p) {
```

```
3   return p.first;
4 }
```

LEDA guarantees that, for a dictionary with n elements (i.e. n (key, info) pairs) `insert`, `lookup`, `del_item`, `del` all take $O(\log n)$ time, `key`, `inf`, `empty`, `size`, `change_inf` take $O(1)$ time, and `clear` takes $O(n)$ time. The realizator `cphstl::tree` meets these requirements for almost any kind of balancing policy (AVL-tree, red-black tree. and AA-tree).

Interface 3. Our implementation of `leda::dictionary` accepts the following template arguments (`K` denoting the key, `V` the information, `C` a LEDA-to-STL comparator, `R` the realizator, `I` the mutable iterator, and `J` the immutable iterator):

```
1   template <
2     typename K,
3     typename V,
4     typename C = leda::stl_compare_less<K>,
5     typename R = cphstl::tree<K, std::pair<K const, V>, cphstl::
          unnamed::key_functor<K const, V>, C>,
6     typename I = cphstl::node_iterator<typename R::node_type, false
          >,
7     typename J = cphstl::node_iterator<typename R::node_type, true>
8   >
9   class dictionary;
```

3.4 Array

Our LEDA implementation of `array` is based on the `vector` framework as described in [13]. This implementation is appropriate since the original implementation provided by LEDA is also based on C++ `vector`s. The real difference between LEDA `array` and STL `vector` is that range of indices can be chosen arbitrarily in LEDA `array`. This means that we can get an array, the indices of which are in the range $[x; y]$, where $x, y \in \mathbb{N}$.

The member functions in the LEDA `array` class are similar to the member functions provided by the CPH STL `vector` framework. The difference is that the LEDA `array` class provides a number of algorithms. The algorithms are `sort`, `permute`, `binary_search`, `binary_locate`, and `unique`. These algorithms have been implemented using the existing STL algorithms. Some of these algorithms (e.g. `sort`) accept a compare function; this has been implemented using the approach described in the section on LEDA dictionary.

The time- and space requirements are given in the LEDA manual. The guarantees are that `access` should take $O(1)$ worst-case time, sorting should take $O(n \lg n)$ worst-case time, and binary search should take $O(\lg n)$ worst-case time (n denotes the size of the vector i.e. $n = y - x + 1$). The space requirement is $O(I\ \texttt{sizeof}(E))$ worst-case space, where E denotes the type of the element stored and I is not given in the manual page[3]. Since

[3] `http://www.algorithmic-solutions.info/leda_manual/array.html`

`leda::array` is based on STL `vector`s, the space consumption may be unbounded according to our observations described in [13]. The user can select the desired data structure (with respect to time and space complexity) by giving the desired policies to our vector framework. An instance of the vector framework is accepted as template argument by our implementation of `leda::array`.

Interface 4. Our implementation of the interface for `leda::array` accepts the following template arguments (`V` denotes the type of the values stored, `R` the type of the realizator, `I` the type of the mutable iterator, and `J` the type of the immutable iterator).

```
1  template <typename V,
2            typename R = cphstl::vector_framework<V>,
3            typename I = cphstl::rank_iterator<R, false>,
4            typename J = cphstl::rank_iterator<R, true> >
5  class array;
```

The default realizator provides similar properties as the `libstdc++` implementation of `vector`.

3.5 Priority queue (`p_queue`)

A priority queue in LEDA (`leda::p_queue`) stores elements containing a priority and an information. Elements are ordered in the queue according to their priority. The ordering is done using a compare function, which may be given to the priority queue using the parameterized constructor. We have implemented the interface using the priority-queue framework as realizator; The code of this framework can be found in [11].

The interface of `leda::p_queue` is very similar to the interface provided by the realizator. However there were a few problems concerning the implementation which we will discuss below.

The first problem was how to compare the elements. The priority-queue framework is designed to store the priority and nothing else. If one desires to store a priority associated with an information, one needs to design a comparator for this purpose. We can store both kinds of data by letting the framework store a pair of the priority and the information, so we need to design a comparator which compares just the priority (the first field in the pair).

This means that we cannot reuse our comparator wrapper as discussed in the section on LEDA dictionary. Therefore we have refined this class template such that it can compare two pairs but it is just comparing the first field of the pairs. The code for this comparator (`stl_compare_key_value`) is shown below.

```
1  template <typename P, typename I>
2  class stl_compare_less_pair : public std::binary_function<P, I,
       bool> {
3  public:
4    stl_compare_less_pair(int (*_f)(P const&, P const&)) {
```

```
5       (*this).f = _f;
6     }
7     bool operator()(std::pair<P, I> const& a, std::pair<P, I> const&
          b) const {
8       if((*this).f(a.first, b.first) == 1) {
9         return true;
10      }
11
12      return false;
13    }
14  private:
15    int (*f)(P const&, P const&);
16  };
```

The second problem relates to the LEDA traversal mechanism. As mentioned earlier, the user can obtain pointers to the first and last element by calling `first_item()` and `last_item()` respectively. The priority-queue framework can create STL-compliant iterators which means that a user can obtain an iterator pointing to the first element and an iterator pointing to the element which is located one position past the last element. The problem here is that the priority-queue framework can only produce forward iterators (`--` not possible), which means that we cannot obtain an iterator (and thereby a pointer, since an iterator can be converted to a pointer) pointing to the last element by `--I(end())` where `I` denotes the type of the iterator. We have found no solution which allows the priority-queue framework to provide bidirectional iterators. The problem is that our priority-queue framework provides a meldable priority queue, and a meld operation may invalidate iterators.

We hope that this issue related to the priority-queue framework will be fixed in the future.

Interface 5. Our implementation of the interface of `leda::p_queue` accepts the following template arguments (where `P` denotes the type of the priorities, `V` the type of the values, `C` the type of the comparator, `R` the type of the realizator, and `J` the type of the immutable iterator[4]):

```
1  template <typename P,
2            typename V,
3            typename C = stl_compare_less_pair<P, V>,
4            typename R = cphstl::priority_queue_framework<std::pair<
                P, V>, C>,
5            typename J = cphstl::priority_queue_iterator<typename R
                ::encapsulator_type, typename R::component_type,
                true>
6  >
7  class p_queue;
```

[4] The priority-queue framework provides no support for mutable iterators, since they can violate the invariant of the data structure, similar to `dictionary`.

3.6 Macros

As mentioned, traversals in LEDA are performed using macros. LEDA provides the following macros:

- `forall_items(x, D)`: All elements in `D` are assigned to the item `x` sequentially. First `x` is assigned to the item pointing to the first element, next `x` is assigned to the item pointing to the second element, and so on. This macro applies to item-based containers.
- `forall_rev_items(x, D)`: Similar to `forall_items`, but the elements are visited (and assigned to `x`) in reverse order. This macro applies to item-based containers.
- `forall(x, L)`: Similar to `forall_items` but the elements stored in `L` are assigned to the value `x` sequentially. This macro applies to lists and sets.
- `forall_rev(x, L)`: Similar to `forall_rev_items`, `x` is just assigned to the values instead of items. This macro applies to lists and sets.

The implementation of the `forall` macro (the other macros are similar) is shown below:

```
1 #undef forall
2 #define forall(x, Q)\
3   x = Q.first_item() ? Q.inf(Q.first_item()) : x;\
4   for (auto tmp = Q.first_item(); tmp != nil; tmp = Q.next_item(tmp)
        , x = tmp ? Q.inf(tmp) : x)
```

This implementation is not accepted by contemporary C++ compilers because the `auto` keyword is not part of the current C++ standard. This keyword is part of the upcoming C++ standard, but we can already use it since gcc 4.4 has support for this feature. We need the `auto` keyword since we do not know the type of `tmp` in advance. In the original implementation by LEDA they use type casting (for example, in `forall_items` the pointer is casted to `void*`). We doubt that the casting solution will work in our setting; furthermore, that solution is not elegant at all.

4. Future work

We will briefly cover some subjects which could be included in future projects as natural successors of this work:

Better encapsulation: The CPH STL provides a high-degree of encapsulation. References to the data stored in a container are declared private, both in the actual containers and in the iterators. With the LEDA interfaces we give encapsulators to the user, which means that our encapsulation is broken. That is because the responsibility of encapsulation belongs to the containers and iterators. To resolve this problem, we need to declare the accessors of the encapsulators private. Each container and iterator should be friends of the encapsulators. We think that this is an important issue, since the encapsulation is a

very important aspect of the CPH STL. We have not implemented this mechanism yet.

Implementation of set: This structure is very similar to a dictionary. One specific detail to note is that `leda::set` has member functions for difference, intersection etc. Ideally, these should be realized as generic functions as in the STL, to be able to apply these functions to any relevant container.

Implementation of integer (multi-precision type): This issue has not been covered in this report, but there are issues like how to calculate and how to instantiate an integer, e.g. from a string. Regarding the first issue, Knuths book [14] has algorithms for addition, subtraction, multiplication (including a brief mention of Karatsuba multiplication, which is faster than ordinary multiplication) and division.

Minimizing compilation dependencies: One specific way of minimizing compilation dependencies is by using a *pointer to implementation*, also known as PIMPL or handle classes. It is described in several web documents, for example, in [18, 8]. In essence it is intended for use in situations when changes to parts of a project should not cause recompilation of other parts if these have not been changed. The idea is mostly relevant for libraries and for very large projects. It can be achieved by keeping (in the header file) only a pointer to a structure. The header file contains only a class declaration, and a pointer to an object of this class. The body of the structure is found in the implementation file. This way it is possible to recompile the latter without causing recompilation in any files using this, since the dependency has been broken by hiding the details in the implementation file. We doubt whether this technique will work in a template-based setting; therefore, a further investigation is needed.

5. Concluding remarks

The main results of this work are fully working abstractions for the containers array, dictionary, priority queue, stack and queue. The report attempts to describe how this has been achieved and to document the most difficult aspects. We think though that it can serve as a tiny starting point for the work of turning the CPH STL into a multi-interface (or multi-abstraction) library.

Acknowledgements

We would like to thank our supervisor, Jyrki Katajainen, for many enlightening and encouraging discussions—this way of learning is efficient education.

References

[1] Algorithmic Solutions, *The LEDA User Manual*, Worldwide Web Document (2008). Available at `http://www.algorithmic-solutions.info/leda_manual/MANUAL.html`.

[2] K. Beck, *Extreme programming explained — Embrace change*, Addison-Wesley (2000).

[3] Boost Community, Boost C++ libraries, Website accessible at `http://www.boost.org/` (2000–2009).

[4] British Standards Institute, *The C++ Standard: Incorporating technical corrigendum 1*, 2nd Edition, John Wiley and Sons, Ltd. (2003).

[5] F. M. Carrano and J. J. Prichard, *Data abstraction and problem solving with C++: Walls and mirrors*, 3rd Edition, Pearson Education, Inc. (2002).

[6] Computational Geometry Algorithms Library, *CGAL User and Reference Manual*, Worldwide Web Document (2009). Available at `http://www.cgal.org/Manual/last/doc_html/cgal_manual/contents.html`.

[7] Department of Computer Science, University of Copenhagen, The CPH STL, Website accessible at `http://www.cphstl.dk/` (2000–2009).

[8] B. Eckel, *Thinking in C++*, Worldwide Web Document (2009). Available at `http://www.codeguru.com/cpp/tic/`.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc. (1995).

[10] J. Katajainen, Stronger guarantees for standard-library containers, *Algorithm Engineering. Oberwolfach report 25/2007*, Mathematisches Forchungsinstitut Oberwolfach (2007), 31–35.

[11] J. Katajainen, Priority-queue framework: Programs, CPH STL Report **2009-7**, Department of Computer Science, University of Copenhagen (2009).

[12] J. Katajainen and B. Simonsen, Applying design patterns to specify the architecture of a generic program library (2008).

[13] J. Katajainen and B. Simonsen, Adaptable component frameworks: Using `vector` from the C++ standard library as an example, *Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming*, ACM (2009), 13–24.

[14] D. E. Knuth, *The art of computer programming. Vol. 2. Seminumerical algorithms*, 3rd Edition, Addison-Wesley (1998).

[15] K. Mehlhorn and S. Näher, The LEDA platform of combinatorial and geometric computing, Worldwide Web Document (1999). Available at `http://www.mpi-inf.mpg.de/~mehlhorn/LEDAbook.html`.

[16] B. Simonsen, Foundations of an adaptable container library, M. Sc. Thesis, Department of Computer Science, University of Copenhagen (2009).

[17] B. Simonsen, A framework for implementing associative containers, CPH STL Report **2009-3**, Department of Computer Science, University of Copenhagen (2009).

[18] H. Sutter, Pimpls—beauty marks you can depend on, Worldwide Web Document (1999). Available at `http://www.gotw.ca/publications/mill04.htm`.

Table of contents

Common

leda-compare-functions.i++ 16
leda-macros.i++ 17

Interfaces

leda-array.h++ 18
leda-array.i++ 20
leda-dictionary-helpers.i++ 28
leda-dictionary.h++ 30
leda-dictionary.i++ 32
leda-p-queue.h++ 40
leda-p-queue.i++ 41
leda-queue.h++ 46
leda-queue.i++ 48
leda-stack.h++ 52
leda-stack.i++ 53

Test programs

array-smoke-test.c++ 55
dict-io-test.c++ 58
dict-word-count.c++ 58
dict-smoke-test.c++ 59
p-queue-smoke-test.c++ 62
queue-operator-test.c++ 63
queue-io-test.c++ 64
queue-smoke-test.c++ 65
stack-io-test.c++ 67
stack-smoke-test.c++ 67

Appendix A. Common

Appendix A.1 `leda-compare-functions.i++`

```
/*
  A predefined compare functor is provided for each type, for which
  operator< is defined.

  Authors: Jyrki Katajainen, Michael Neidhardt, and Bo Simonsen 2009
*/

#ifndef __LEDA_COMPARE_FUNCTIONS__
#define __LEDA_COMPARE_FUNCTIONS__

#include <functional> // defines std::binary_function
#include <tr1/type_traits> //defines std::tr1::is_convertible
#include <utility>

namespace leda {

  template <typename K, typename L = K>
  class comparator
    : public std::binary_function<K, L, int> {
  public:

    int operator()(K const& x, L const& y) const {
      static_assert(std::tr1::is_convertible<L, K>::value,
                    "Only convertible types can be compared");
      if (x < y) {
        return -1;
      }
      if (y < x) {
          return 1;
      }
      return 0;
    }
  };

  template <typename T>
  class stl_compare_less : public std::binary_function<T, T, bool> {
  public:
    stl_compare_less(int (*_f)(T const&, T const&)) {
      (*this).f = _f;
    }
    bool operator()(T const& a, T const& b) const {
      if((*this).f(a, b) == -1) {
        return true;
      }

      return false;
    }
```

```
48   private:
49     int (*f)(T const&, T const&);
50   };
51
52   template <typename P, typename I>
53   class stl_compare_less_pair : public std::binary_function<P, I,
         bool> {
54   public:
55     stl_compare_less_pair(int (*_f)(P const&, P const&)) {
56       (*this).f = _f;
57     }
58     bool operator()(std::pair<P, I> const& a, std::pair<P, I> const&
            b) const {
59       if((*this).f(a.first, b.first) == 1) {
60         return true;
61       }
62
63       return false;
64     }
65   private:
66     int (*f)(P const&, P const&);
67   };
68
69   template <typename T>
70   int compare(T const& a, T const& b) {
71     if (a < b) {
72       return -1;
73     }
74     else if(a > b) {
75       return 1;
76     }
77     return 0;
78   }
79 }
80 #endif
```

Appendix A.2 `leda-macros.i++`

```
1  /*
2    Some of the LEDA macros are defined here.
3
4    Author: Jyrki Katajainen and Bo Simonsen 2009
5  */
6
7  #ifndef __LEDA_MACROS__
8  #define __LEDA_MACROS__
9
10 /* Jyrkis implementation of forall
11
12   auto __p = Q.begin();\
13   for (x = (__p != Q.end())? *__p : x;\
14        __p != Q.end();\
```

```
15        ++__p, x = (__p != Q.end())? *__p : x)
16 */
17
18 #undef forall
19 #define forall(x, Q)\
20   x = Q.first_item() ? Q.inf(Q.first_item()) : x;\
21   for (auto tmp = Q.first_item(); tmp != nil; tmp = Q.next_item(tmp)
       , x = tmp ? Q.inf(tmp) : x)
22
23 #undef forall_defined
24 #define forall_defined(k, Q)\
25   k = Q.first_item() ? Q.key(Q.first_item()) : k;\
26   for (auto tmp = Q.first_item(); tmp != nil; tmp = Q.next_item(tmp)
       , k = tmp ? Q.key(tmp) : k)
27
28 #undef forall_items
29 #define forall_items(p, D)\
30   p = D.first_item();\
31   for (auto tmp = D.next_item(p); p != nil; p = tmp, tmp = D.
       next_item(p))
32
33 #undef forall_rev_items
34 #define forall_rev_items(p, D)\
35   p = D.last_item();\
36   for (auto tmp = D.prev(p); p != nil; p = tmp, tmp = D.prev(p))
37
38 #endif
```

Appendix B. Interfaces

Appendix B.1 `leda-array.h++`

```
1 /*
2   The interface of leda::array
3
4   Authors: Bo Simonsen 2009
5 */
6
7 #include "vector-framework.h++"
8 #include "rank-iterator.h++"
9 #include "leda-compare-functions.i++"
10
11 #include <ostream>
12 #include <sstream>
13 #include <memory>
14
15 namespace leda {
16   /* The default configuration is a regular dynamic array (as
17   ** std::vector which original implementation of leda::array is
       based on),
18   ** where the array is never constracted. */
19
```

```
20  template <typename V,
21            typename R = cphstl::vector_framework<V>,
22            typename I = cphstl::rank_iterator<R, false>,
23            typename J = cphstl::rank_iterator<R, true> >
24  class array {
25  public:
26    typedef V value_type;
27    typedef R realizator_type;
28    typedef I iterator;
29    typedef J const_iterator;
30
31    typedef typename realizator_type::concrete_iterator item;
32    typedef int size_type;
33
34    array(size_type, size_type);
35    array(size_type);
36    array();
37    array(size_type, V, V);
38    array(size_type, V, V, V);
39    array(size_type, V, V, V, V);
40
41    array(array const&);
42    array& operator=(array const&);
43
44    item first_item();
45    item last_item();
46    item next_item(item);
47    item prev_item(item);
48
49    V& inf(item);
50
51    iterator begin();
52    iterator end();
53    const_iterator begin() const;
54    const_iterator end() const;
55
56    V& get(size_type);
57    V const& get(size_type) const;
58    void set(size_type, V);
59    V& operator[](size_type);
60
61    void copy(size_type, array const&, size_type);
62    void resize(size_type, size_type);
63    void resize(size_type);
64
65    size_type low() const;
66    size_type high() const;
67    size_type size() const;
68
69    void init(V);
70    bool C_style() const;
71
```

```
72      void swap(size_type, size_type);
73
74      void sort(size_type (*)(V const&, V const&));
75      void sort(size_type, size_type);
76      void sort();
77
78      void sort(size_type (*)(V const&, V const&), size_type,
            size_type);
79      void permute();
80      void permute(size_type, size_type);
81
82      size_type binary_search(size_type (*)(V const&, V const&), V);
83      size_type binary_search(V);
84      size_type binary_locate(size_type (*)(V const&, V const&), V);
85      size_type binary_locate(V);
86
87      size_type unique();
88
89      void print(std::ostream&, char = ' ');
90      void print(char = ' ');
91      void print(char const*, char = ' ');
92
93      void read(std::istream&);
94      void read();
95      void read(char const*);
96
97    protected:
98      void realizator_resize(size_type);
99      std::pair<I, I> get_iterators(size_type, size_type);
100   private:
101     size_type s;
102     size_type e;
103     realizator_type realizator;
104
105   };
106
107   template <typename V, typename R, typename I, typename J>
108   std::istream&
109   operator>>(std::istream&, array<V, R, I, J>&);
110
111   template <typename V, typename R, typename I, typename J>
112   std::ostream&
113   operator<<(std::ostream&, array<V, R, I, J>&);
114 }
115
116 #include "leda-array.i++"
```

Appendix B.2 `leda-array.i++`

```
1 /*
2   An implementation of leda::array
3
```

```
4     Authors: Bo Simonsen 2009
5   */
6
7   namespace leda {
8
9     template<typename V, typename R, typename I, typename J>
10    array<V, R, I, J>::array(size_type a, size_type b) : s(a), e(b) {
11      (*this).realizator_resize((*this).e - (*this).s + 1);
12    }
13
14    template<typename V, typename R, typename I, typename J>
15    array<V, R, I, J>::array(size_type n) : s(0), e(n-1) {
16      (*this).realizator_resize(n);
17    }
18
19    /* I am not sure about this one */
20    template<typename V, typename R, typename I, typename J>
21    array<V, R, I, J>::array() : s(0), e(-1) {
22    }
23
24    template<typename V, typename R, typename I, typename J>
25    array<V, R, I, J>::array(size_type low, V x, V y) : s(low), e(low
          +1) {
26      (*this).realizator.insert((*this).realizator.end(), x);
27      (*this).realizator.insert((*this).realizator.end(), y);
28    }
29
30    template<typename V, typename R, typename I, typename J>
31    array<V, R, I, J>::array(size_type low, V x, V y, V w) : s(low), e
          (low+2) {
32      (*this).realizator.insert((*this).realizator.end(), x);
33      (*this).realizator.insert((*this).realizator.end(), y);
34      (*this).realizator.insert((*this).realizator.end(), w);
35    }
36
37    template<typename V, typename R, typename I, typename J>
38    array<V, R, I, J>::array(size_type low, V x, V y, V z, V w) : s(
          low), e(low+3) {
39      (*this).realizator.insert((*this).realizator.end(), x);
40      (*this).realizator.insert((*this).realizator.end(), y);
41      (*this).realizator.insert((*this).realizator.end(), z);
42      (*this).realizator.insert((*this).realizator.end(), w);
43    }
44
45    template<typename V, typename R, typename I, typename J>
46    array<V, R, I, J>::array(array const& a) : realizator(), s(a.s), e
          (a.e) {
47      (*this).realizator_resize((*this).size());
48      (*this).copy((*this).s, a, (*this).s);
49    }
50
51    template<typename V, typename R, typename I, typename J>
```

```
52  array<V, R, I, J>&
53  array<V, R, I, J>::operator=(array const& a) {
54    (*this).s = a.s;
55    (*this).e = a.e;
56    (*this).realizator_resize((*this).size());
57    (*this).copy((*this).s, a, (*this).s);
58    return (*this);
59  }
60
61  template<typename V, typename R, typename I, typename J>
62  V& array<V, R, I, J>::get(size_type x) {
63    return (*this).realizator[x - (*this).low()];
64  }
65
66  template<typename V, typename R, typename I, typename J>
67  V const& array<V, R, I, J>::get(size_type x) const {
68    return (*this).realizator[x - (*this).low()];
69  }
70
71  template<typename V, typename R, typename I, typename J>
72  void array<V, R, I, J>::set(size_type x, V e) {
73    (*this).get(x) = e;
74  }
75
76  template<typename V, typename R, typename I, typename J>
77  V& array<V, R, I, J>::operator[](size_type x) {
78    return (*this).get(x);
79  }
80
81  template<typename V, typename R, typename I, typename J>
82  void array<V, R, I, J>::copy(size_type x, array const& B,
        size_type y) {
83    while(y <= B.high()) {
84      (*this).set(x, B.get(y));
85      ++x;
86      ++y;
87    }
88  }
89
90  template<typename V, typename R, typename I, typename J>
91  void array<V, R, I, J>::resize(size_type a, size_type b) {
92    (*this).s = a;
93    (*this).e = b;
94    (*this).realizator_resize((*this).size());
95  }
96
97  template<typename V, typename R, typename I, typename J>
98  void array<V, R, I, J>::resize(size_type n) {
99    (*this).resize(0, n-1);
100 }
101
102 template<typename V, typename R, typename I, typename J>
```

```
103  typename array<V, R, I, J>::size_type
104  array<V, R, I, J>::low() const {
105      return (*this).s;
106    }
107
108  template<typename V, typename R, typename I, typename J>
109  typename array<V, R, I, J>::size_type
110  array<V, R, I, J>::high() const {
111    return (*this).e;
112  }
113
114  template<typename V, typename R, typename I, typename J>
115  typename array<V, R, I, J>::size_type
116  array<V, R, I, J>::size() const {
117    return (*this).e - (*this).s + 1;
118  }
119
120  template<typename V, typename R, typename I, typename J>
121  void array<V, R, I, J>::init(V x) {
122    for(size_type i = (*this).low(); i <= (*this).high(); ++i) {
123      (*this).set(i, x);
124    }
125  }
126
127  template<typename V, typename R, typename I, typename J>
128  bool array<V, R, I, J>::C_style() const {
129    return (*this).low() == 0;
130  }
131
132  template<typename V, typename R, typename I, typename J>
133  void array<V, R, I, J>::swap(size_type i, size_type j) {
134    std::swap((*this).get(i), (*this).get(j));
135  }
136
137  template<typename V, typename R, typename I, typename J>
138  typename array<V, R, I, J>::size_type
139  array<V, R, I, J>::unique() {
140    std::pair<I, I> p = (*this).get_iterators((*this).low(), (*this)
         .high());
141    I new_end = std::unique(p.first, p.second);
142    size_type index = (new_end - p.first) + (*this).low();
143    return index - 1;
144  }
145
146  /* Sort functions */
147
148  template<typename V, typename R, typename I, typename J>
149  void array<V, R, I, J>::sort(size_type (*cmp)(V const&, V const&))
        {
150    (*this).sort(cmp, (*this).low(), (*this).high());
151  }
152
```

```
153  template<typename V, typename R, typename I, typename J>
154  void array<V, R, I, J>::sort(size_type l, size_type h) {
155    (*this).sort(compare, l, h);
156  }
157
158  template<typename V, typename R, typename I, typename J>
159  void array<V, R, I, J>::sort() {
160      (*this).sort(compare);
161  }
162
163  template<typename V, typename R, typename I, typename J>
164  void array<V, R, I, J>::sort(size_type (*cmp)(V const&, V const&),
         size_type l, size_type h) {
165    std::pair<I, I> p = (*this).get_iterators(l, h);
166    std::sort(p.first, p.second, stl_compare_less<V>(cmp));
167  }
168
169  /* Permute functions */
170
171  template<typename V, typename R, typename I, typename J>
172  void array<V, R, I, J>::permute() {
173    (*this).permute((*this).low(), (*this).high());
174  }
175
176  template<typename V, typename R, typename I, typename J>
177  void array<V, R, I, J>::permute(size_type l, size_type h) {
178    std::pair<I, I> p = (*this).get_iterators(l, h);
179    std::random_shuffle(p.first, p.second);
180  }
181
182  /* Binary search functions */
183
184  template<typename V, typename R, typename I, typename J>
185  typename array<V, R, I, J>::size_type
186  array<V, R, I, J>::binary_search(size_type (*cmp)(V const&, V
        const&), V x) {
187    std::pair<I, I> p = (*this).get_iterators((*this).low(), (*this)
         .high());
188    I it = std::lower_bound(p.first, p.second, x,
189                            stl_compare_less<V>(cmp));
190
191    if (it == p.second) {
192      return (*this).low() - 1;
193    }
194    return (it - p.first) + (*this).low();
195  }
196
197  template<typename V, typename R, typename I, typename J>
198  typename array<V, R, I, J>::size_type
199  array<V, R, I, J>::binary_search(V x) {
200    return (*this).binary_search(compare, x);
201  }
```

```
202
203 template<typename V, typename R, typename I, typename J>
204 typename array<V, R, I, J>::size_type
205 array<V, R, I, J>::binary_locate(size_type (*cmp)(V const&, V
        const&), V x) {
206
207   std::pair<I, I> p = (*this).get_iterators((*this).low(), (*this)
          .high());
208   I it = std::upper_bound(p.first, p.second, x,
209                           stl_compare_less<V>(cmp));
210
211   if (it == p.second) {
212     return (*this).low() - 1;
213   }
214
215   size_type index = (it - p.first) - 1 + (*this).low();
216   if (cmp(x, (*this).get(index)) > 0) {
217     return (*this).low() - 1;
218   }
219
220   return index;
221 }
222
223 template<typename V, typename R, typename I, typename J>
224 typename array<V, R, I, J>::size_type
225 array<V, R, I, J>::binary_locate(V x) {
226   return (*this).binary_locate(compare, x);
227 }
228
229 /* Print functions */
230
231 template<typename V, typename R, typename I, typename J>
232 void array<V, R, I, J>::print(std::ostream& o, char space) {
233   std::stringstream sss;
234   sss << space;
235   for(size_type i = (*this).low(); i <= (*this).high(); ++i) {
236     std::stringstream ss;
237     ss << (*this).get(i);
238     o << ss.str();
239     if(i != (*this).high()) {
240       o << sss.str();
241     }
242   }
243 }
244
245 template<typename V, typename R, typename I, typename J>
246 void array<V, R, I, J>::print(char space) {
247   (*this).print(std::cout, space);
248 }
249
250 template<typename V, typename R, typename I, typename J>
251 void array<V, R, I, J>::print(char const* header, char space) {
```

```
252     std::cout << header << std::endl;
253     (*this).print(std::cout, space);
254   }
255
256   /* Read functions */
257   template<typename V, typename R, typename I, typename J>
258   void array<V, R, I, J>::read(std::istream& s) {
259     V v;
260     for(size_type i = (*this).low(); i <= (*this).high(); ++i) {
261       s >> v;
262       (*this).set(i, v);
263     }
264   }
265
266   template<typename V, typename R, typename I, typename J>
267   void array<V, R, I, J>::read() {
268     (*this).read(std::cin);
269   }
270
271   template<typename V, typename R, typename I, typename J>
272   void array<V, R, I, J>::read(const char* header) {
273     std::cout << header;
274     (*this).read();
275   }
276
277   /* Protected members */
278
279   template<typename V, typename R, typename I, typename J>
280   void array<V, R, I, J>::realizator_resize(size_type n) {
281     while((*this).realizator.size() < n) {
282       (*this).realizator.insert((*this).realizator.end(), V());
283     }
284   }
285
286   template<typename V, typename R, typename I, typename J>
287   std::pair<I, I> array<V, R, I, J>::get_iterators(size_type l,
          size_type h) {
288     size_type start_index = l - (*this).low();
289     size_type end_index = h - (*this).low();
290     I begin_it = (*this).realizator.begin();
291     I end_it = (*this).realizator.begin();
292     begin_it += start_index;
293     end_it += end_index + 1;
294     return std::pair<I, I>(begin_it, end_it);
295   }
296
297   /* Iteration - we use iterators for implementing next_item,
          prev_item
298   ** and inf. */
299
300   template<typename V, typename R, typename I, typename J>
301   typename array<V, R, I, J>::item
```

```
302 array<V, R, I, J>::first_item() {
303   return (*this).realizator.begin();
304 }
305
306 template<typename V, typename R, typename I, typename J>
307 typename array<V, R, I, J>::item
308 array<V, R, I, J>::last_item() {
309   return (*this).prev_item((*this).realizator.end());
310 }
311
312 template<typename V, typename R, typename I, typename J>
313 typename array<V, R, I, J>::item
314 array<V, R, I, J>::next_item(item a) {
315   I tmp(a);
316   ++tmp;
317   return tmp;
318 }
319
320 template<typename V, typename R, typename I, typename J>
321 typename array<V, R, I, J>::item
322 array<V, R, I, J>::prev_item(item a) {
323   I tmp(a);
324   --tmp;
325   return tmp;
326 }
327
328 template<typename V, typename R, typename I, typename J>
329 V&
330 array<V, R, I, J>::inf(item a) {
331   I tmp(a);
332   return *tmp;
333 }
334
335 /* Iterators */
336
337 template<typename V, typename R, typename I, typename J>
338 typename array<V, R, I, J>::iterator
339 array<V, R, I, J>::begin() {
340   return I((*this).realizator.begin());
341 }
342
343 template<typename V, typename R, typename I, typename J>
344 typename array<V, R, I, J>::iterator
345 array<V, R, I, J>::end() {
346   return I((*this).realizator.end());
347 }
348
349 template<typename V, typename R, typename I, typename J>
350 typename array<V, R, I, J>::const_iterator
351 array<V, R, I, J>::begin() const {
352   return J((*this).realizator.begin());
353 }
```

```
354
355   template<typename V, typename R, typename I, typename J>
356   typename array<V, R, I, J>::const_iterator
357   array<V, R, I, J>::end() const {
358     return J((*this).realizator.end());
359   }
360
361   /* operator<< and operator>> */
362
363   template <typename V, typename R, typename I, typename J>
364   std::istream&
365   operator>>(std::istream& i, array<V, R, I, J>& a) {
366     a.read(i);
367     return i;
368   }
369
370   template <typename V, typename R, typename I, typename J>
371   std::ostream&
372   operator<<(std::ostream& o, array<V, R, I, J>& a) {
373     a.print(o);
374     return o;
375   }
376
377 }
```

Appendix B.3 `leda-dictionary-helpers.i++`

```
 1 /*
 2   The tools specified are needed to make an STL realizator suitable
 3   for implementing a LEDA dictionary.
 4
 5   Authors: Jyrki Katajainen and Michael Neidhardt 2009
 6 */
 7
 8 #ifndef __LEDA_DICTIONARY_HELPERS__
 9 #define __LEDA_DICTIONARY_HELPERS__
10
11 #include <functional> // defines std::binary_function and std::
       unary_function
12 #include <utility> // defines std::pair
13 #include <tr1/type_traits> //defines std::tr1::is_convertible
14 #include "leda-compare-functions.i++" // defines leda::comparator
15
16 static const int nil = 0;
17
18 namespace leda {
19
20   // comparison between an item, i.e. (iterator, bool) pair, and nil
21
22   template <typename Pair>
23   bool operator==(Pair const& p, int const& q) {
24     if (p.second && &q == &nil) {
```

```
25      return true;
26    }
27    return false;
28  }
29
30  template <typename Pair>
31  bool operator==(int const& q, Pair const& p) {
32    return (p == q);
33  }
34
35  template <typename Pair>
36  bool operator!=(Pair const& p, int const& q) {
37    return ! (p == q);
38  }
39
40  template <typename Pair>
41  bool operator!=(int const& q, Pair const& p) {
42    return ! (p == q);
43  }
44
45  // two-way comparison support
46
47  template <typename K, typename L = K, typename C = leda::
        comparator<K, L> >
48  class two_way_compare
49    : public std::binary_function<K, L, bool> {
50  public:
51
52    two_way_compare(C function_object)
53      : compare_functor(function_object) {
54    }
55
56    bool operator()(K const& x, L const& y) const {
57      static_assert(std::tr1::is_convertible<L, K>::value,
58                    "Only convertible types can be compared");
59      return compare_functor(x, y) == -1;
60    }
61
62  private:
63
64    C compare_functor;
65
66  };
67
68  /*
69  template <typename K, typename L>
70  class two_way_compare<K, L, int T_function(K const&, L const&)>
71    : public std::binary_function<K, L, bool> {
72  public:
73
74 --- add a constructor
75
```

```
76    bool operator()(K const& x, L const& y) const {
77      static_assert(std::tr1::is_convertible<L, K>::value,
78                    "Only convertible types can be compared");
79      return T_function(x, y) == -1;
80    }
81  };
82  */
83
84  // key extractor from a pair
85
86  template <typename K, typename V>
87  class key_extractor;
88
89  template <typename K, typename I>
90  class key_extractor<K, std::pair<K, I> >
91    : public std::unary_function<std::pair<K, I>, K> {
92  public:
93    K operator()(std::pair<K, I> const& e) const {
94      return e.first;
95    }
96  };
97
98 }
99 #endif
```

Appendix B.4 `leda-dictionary.h++`

```
1 /*
2   leda::dictionary is an API for a dictionary that is realized by a
3   realizator from the STL.
4
5   Authors: Jyrki Katajainen, Michael Neidhardt, and Bo Simonsen 2009
6 */
7
8 #ifndef __LEDA_DICTIONARY__
9 #define __LEDA_DICTIONARY__
10
11 #include <iostream> // defines standard streams
12 #include <map> // defines std::map
13 #include <utility> // defines std::pair
14 #include "leda-dictionary-helpers.i++" // defines leda::
     two_way_compare and nil
15 #include "leda-macros.i++" // defines LEDA macros
16
17 #include "stl-map.h++"
18 #include "tree.h++"
19 #include "node-iterator.h++"
20
21 namespace leda {
22
23   template <
24     typename K,
```

```
25    typename V,
26    typename C = leda::stl_compare_less<K>,
27    typename R = cphstl::tree<K, std::pair<K const, V>, cphstl::
         unnamed::key_functor<K const, V>, C>,
28    typename I = cphstl::node_iterator<typename R::node_type, false
         >,
29    typename J = cphstl::node_iterator<typename R::node_type, true>
30  >
31  class dictionary {
32  public:
33
34    // types
35
36    typedef K key_type;
37    typedef V inf_type;
38    typedef C comparator_type;
39    typedef I iterator;
40    typedef J const_iterator;
41    typedef typename R::node_type* item;
42    typedef typename R::node_type const* const_item;
43    // structors
44
45    explicit dictionary();
46    dictionary(int (*)(K const&, K const&));
47    dictionary(dictionary const&);
48    ~dictionary();
49    dictionary& operator=(dictionary const&);
50
51    // items
52
53    item first_item();
54    const_item first_item() const;
55    item last_item();
56    const_item last_item() const;
57    item next_item(item);
58    const_item next_item(const_item) const;
59    item prev_item(item);
60    const_item prev_item(const_item) const;
61
62    // iterators
63
64    iterator begin();
65    const_iterator begin() const;
66    iterator end();
67    const_iterator end() const;
68
69    // accessors
70
71    int size() const;
72    bool empty() const;
73    K const& key(item) const;
74    V const& inf(item) const;
```

```
75      V const& operator[](const_item) const;
76
77      // modifiers
78
79      V& operator[](item);
80      item insert(K const&, V const&);
81      item lookup(K const&);
82      inf_type access(K const&);
83      void del(K const&);
84      void del_item(item);
85      bool defined(K const&);
86      void undefine(K const&);
87      void change_inf(item, V const&);
88      void clear();
89
90      // operators
91
92      template <typename KK, typename VV, typename CC, typename RR,
            typename II, typename JJ>
93      friend std::istream& operator>>(std::istream&, dictionary<KK, VV
            , CC, RR, II, JJ>&);
94
95      template <typename KK, typename VV, typename CC, typename RR,
            typename II, typename JJ>
96      friend std::ostream& operator<<(std::ostream&, dictionary<KK, VV
            , CC, RR, II, JJ> const&);
97
98    protected:
99
100     typedef R realizator_type;
101     realizator_type realizator;
102
103   };
104
105 }
106
107 #include "leda-dictionary.i++" // implements leda::dictionary
108 #endif
```

Appendix B.5 `leda-dictionary.i++`

```
1  /*
2    An implementation of leda::dictionary
3
4    Authors: Jyrki Katajainen, Michael Neidhardt, and Bo Simonsen 2009
5  */
6
7  #include <iostream> // defines standard streams
8  #include <iterator> // defines std::iostream_iterator and std::
       ostream_iterator
9  #include <stdexcept> // defines standard exception hierarchy
10 #include <utility> // defines std::pair
```

```
11
12 namespace leda {
13
14   // default constructor
15
16   template <typename K, typename V, typename C, typename R, typename
         I, typename J>
17   dictionary<K, V, C, R, I, J>::dictionary()
18     : realizator(comparator_type(compare), typename R::
          allocator_type()) {
19   }
20
21   // parametrized constructor
22   template <typename K, typename V, typename C, typename R, typename
         I, typename J>
23   dictionary<K, V, C, R, I, J>::dictionary(int (*cmp)(K const&, K
        const&))
24     : realizator(comparator_type(cmp), typename R::allocator_type())
           {
25   }
26
27   // copy constructor
28   template <typename K, typename V, typename C, typename R, typename
         I, typename J>
29   dictionary<K, V, C, R, I, J>::dictionary(dictionary<K, V, C, R, I,
         J> const& other)
30     : realizator(other.realizator) {
31   }
32
33   // destructor
34   template <typename K, typename V, typename C, typename R, typename
         I, typename J>
35   dictionary<K, V, C, R, I, J>::~dictionary() {
36   }
37
38   // operator=
39   template <typename K, typename V, typename C, typename R, typename
         I, typename J>
40   dictionary<K, V, C, R, I, J>&
41   dictionary<K, V, C, R, I, J>::operator=(dictionary const& other) {
42     (*this).realizator = other.realizator;
43     return *this;
44   }
45
46   // first
47   template <typename K, typename V, typename C, typename R, typename
         I, typename J>
48   typename dictionary<K, V, C, R, I, J>::item
49   dictionary<K, V, C, R, I, J>::first_item() {
50     return (*this).begin();
51   }
52
```

```
53  template <typename K, typename V, typename C, typename R, typename
        I, typename J>
54  typename dictionary<K, V, C, R, I, J>::const_item
55  dictionary<K, V, C, R, I, J>::first_item() const {
56    return (*this).begin();
57  }
58
59  // last
60  template <typename K, typename V, typename C, typename R, typename
        I, typename J>
61  typename dictionary<K, V, C, R, I, J>::item
62  dictionary<K, V, C, R, I, J>::last_item() {
63    if ((*this).empty()) {
64      return nil;
65    }
66    return (*(*this).realizator.end()).predecessor();
67  }
68
69  template <typename K, typename V, typename C, typename R, typename
        I, typename J>
70  typename dictionary<K, V, C, R, I, J>::const_item
71  dictionary<K, V, C, R, I, J>::last_item() const {
72    if ((*this).empty()) {
73      return nil;
74    }
75    return (*(*this).realizator.end()).predecessor();
76  }
77
78  // succ
79  template <typename K, typename V, typename C, typename R, typename
        I, typename J>
80  typename dictionary<K, V, C, R, I, J>::item
81  dictionary<K, V, C, R, I, J>::next_item(item p) {
82    if (p == nil || (*p).successor() == (*this).realizator.end()) {
83      return nil;
84    }
85    return (*p).successor();
86  }
87
88  template <typename K, typename V, typename C, typename R, typename
        I, typename J>
89  typename dictionary<K, V, C, R, I, J>::const_item
90  dictionary<K, V, C, R, I, J>::next_item(const_item p) const {
91    if (p == nil || (*p).successor() == (*this).realizator.end()) {
92      return nil;
93    }
94    return (*p).successor();
95  }
96
97  // prev
98  template <typename K, typename V, typename C, typename R, typename
        I, typename J>
```

```
99  typename dictionary<K, V, C, R, I, J>::item
100 dictionary<K, V, C, R, I, J>::prev_item(item p) {
101   if(p == nil || (*p).predecessor() == (*this).realizator.end()) {
102     return nil;
103   }
104   return (*p).predecessor();
105 }
106 template <typename K, typename V, typename C, typename R, typename
        I, typename J>
107 typename dictionary<K, V, C, R, I, J>::const_item
108 dictionary<K, V, C, R, I, J>::prev_item(const_item p) const {
109   if(p == nil) {
110     return p;
111   }
112   if((*p).predecessor() == (*this).realizator.end()) {
113     return nil;
114   }
115   return (*p).predecessor();
116 }
117
118 // begin
119 template <typename K, typename V, typename C, typename R, typename
        I, typename J>
120 typename dictionary<K, V, C, R, I, J>::iterator
121 dictionary<K, V, C, R, I, J>::begin() {
122   return iterator((*this).realizator.begin());
123 }
124
125 template <typename K, typename V, typename C, typename R, typename
        I, typename J>
126 typename dictionary<K, V, C, R, I, J>::const_iterator
127 dictionary<K, V, C, R, I, J>::begin() const {
128   return const_iterator((*this).realizator.begin());
129 }
130
131 // end
132 template <typename K, typename V, typename C, typename R, typename
        I, typename J>
133 typename dictionary<K, V, C, R, I, J>::iterator
134 dictionary<K, V, C, R, I, J>::end() {
135   return iterator((*this).realizator.end());
136 }
137
138 template <typename K, typename V, typename C, typename R, typename
        I, typename J>
139 typename dictionary<K, V, C, R, I, J>::const_iterator
140 dictionary<K, V, C, R, I, J>::end() const {
141   return const_iterator((*this).realizator.end());
142 }
143
144 // size
```

```
145 template <typename K, typename V, typename C, typename R, typename
        I, typename J>
146 int
147 dictionary<K, V, C, R, I, J>::size() const {
148   return static_cast<int>((*this).realizator.size());
149 }
150
151 // empty
152  template <typename K, typename V, typename C, typename R,
        typename I, typename J>
153 bool dictionary<K, V, C, R, I, J>::empty() const {
154   return (*this).realizator.size() == 0;
155 }
156
157 // key
158 template <typename K, typename V, typename C, typename R, typename
        I, typename J>
159 K const&
160 dictionary<K, V, C, R, I, J>::key(item p) const {
161   return (*p).content().first;
162 }
163
164 // inf
165 template <typename K, typename V, typename C, typename R, typename
        I, typename J>
166 V const&
167 dictionary<K, V, C, R, I, J>::inf(item p) const {
168   return (*p).content().second;
169 }
170
171 // operator[]
172 template <typename K, typename V, typename C, typename R, typename
        I, typename J>
173 V const&
174 dictionary<K, V, C, R, I, J>::operator[](const_item p) const {
175   return (*p).content().second;
176 }
177
178 template <typename K, typename V, typename C, typename R, typename
        I, typename J>
179 V&
180 dictionary<K, V, C, R, I, J>::operator[](item p) {
181   return (*p).content().second;
182 }
183
184 // insert
185 template <typename K, typename V, typename C, typename R, typename
        I, typename J>
186 typename dictionary<K, V, C, R, I, J>::item
187 dictionary<K, V, C, R, I, J>::insert(K const& key, V const& info)
       {
188    std::pair<item, bool> return_value =
```

```
189      (*this).realizator.insert(typename R::value_type(key, info));
190    if (! return_value.second) { // key was already there
191      (*this).change_inf(return_value.first, info);
192    }
193    return return_value.first;
194  }
195
196  // lookup
197  template <typename K, typename V, typename C, typename R, typename
         I, typename J>
198  typename dictionary<K, V, C, R, I, J>::item
199  dictionary<K, V, C, R, I, J>::lookup(K const& key) {
200    item p = (*this).realizator.lower_bound(key);
201    typedef typename R::key_compare key_compare;
202    key_compare less = (*this).realizator.key_comp();
203    if (p == (*this).realizator.end() || less(key, (*p).content().
          first)) {
204      return nil;
205    }
206    return p;
207  }
208
209  // access
210  template <typename K, typename V, typename C, typename R, typename
         I, typename J>
211  V
212  dictionary<K, V, C, R, I, J>::access(K const& key) {
213    item p = (*this).realizator.lower_bound(key);
214    typedef typename R::key_compare key_compare;
215    key_compare less = (*this).realizator.key_comp();
216    if (p == (*this).realizator.end() || less(key, (*p).content().
          first)) {
217      assert(false); // precondition failed: no such item
218    }
219    return (*p).content().second;
220  }
221
222  // del
223  template <typename K, typename V, typename C, typename R, typename
         I, typename J>
224  void
225  dictionary<K, V, C, R, I, J>::del(K const& key) {
226    item p = (*this).realizator.lower_bound(key);
227    typedef typename R::key_compare key_compare;
228    key_compare less = (*this).realizator.key_comp();
229    if (p != (*this).realizator.end() && ! less(key, (*p).content().
          first)) {
230      (*this).realizator.erase(p);
231    }
232  }
233
234  // del_item
```

```
235 template <typename K, typename V, typename C, typename R, typename
        I, typename J>
236 void
237 dictionary<K, V, C, R, I, J>::del_item(item p) {
238   (*this).realizator.erase(p);
239 }
240
241 // defined
242 template <typename K, typename V, typename C, typename R, typename
        I, typename J>
243 bool
244 dictionary<K, V, C, R, I, J>::defined(K const& key) {
245   item p = (*this).realizator.lower_bound(key);
246   if (p == (*this).realizator.end()) {
247     return false;
248   }
249   return true;
250 }
251
252 // undefine
253 template <typename K, typename V, typename C, typename R, typename
        I, typename J>
254 void
255 dictionary<K, V, C, R, I, J>::undefine(K const& key) {
256   (*this).del(key);
257 }
258
259 // change_inf
260 template <typename K, typename V, typename C, typename R, typename
        I, typename J>
261 void
262 dictionary<K, V, C, R, I, J>::change_inf(item p, V const& info) {
263   if (p != (typename R::node_type*) nil) {
264     (*p).content().second = info;
265   }
266 }
267
268 // clear
269 template <typename K, typename V, typename C, typename R, typename
        I, typename J>
270 void
271 dictionary<K, V, C, R, I, J>::clear() {
272   typedef typename dictionary<K, V, C, R, I, J>::iterator iterator
          ;
273   iterator p = (*this).begin();
274   while (p != (*this).end()) {
275     iterator q = p;
276     ++q;
277     (*this).realizator.erase(p);
278     p = q;
279   }
280 }
```

```
281
282  // operator>>
283  template <typename K, typename V, typename C, typename R, typename
          I, typename J>
284  std::istream&
285  operator>>(std::istream& s, dictionary<K, V, C, R, I, J>& d) {
286    char c;
287    K key;
288    I info;
289    while (1) {
290      s >> c;
291      if(s.fail()) {
292        break;
293      }
294      if (c != '(') {
295        throw std::runtime_error("stream corrupted");
296      }
297      s >> key;
298      if (! s) {
299        throw std::runtime_error("stream corrupted");
300      }
301      s >> c;
302      if (c != ',') {
303        throw std::runtime_error("stream corrupted");
304      }
305      s >> info;
306      if (! s) {
307        throw std::runtime_error("stream corrupted");
308      }
309      s >> c;
310      if (c != ')') {
311        throw std::runtime_error("stream corrupted");
312      }
313      (void) d.insert(key, info);
314    }
315    return s;
316  }
317
318  // operator<<
319  template <typename K, typename V, typename C, typename R, typename
          I, typename J>
320  std::ostream&
321  operator<<(std::ostream& s, dictionary<K, V, C, R, I, J> const& d)
          {
322    typedef typename dictionary<K, V, C, R, I, J>::const_iterator
          const_iterator;
323    for (const_iterator t = d.begin(); t != d.end(); ++t) {
324      s << "(" << (*t).first << ", " << (*t).second << ")";
325      s << std::endl;
326    }
327    return s;
328  }
```

```
329
330 }
```

Appendix B.6 `leda-p-queue.h++`

```
1  /*
2    The interface of leda::p_queue.
3
4    Authors: Bo Simonsen 2009
5  */
6
7  #include "leda-compare-functions.i++"
8  #include "priority-queue-framework.h++"
9  #include "priority-queue-iterator.h++"
10
11 #include <stdexcept>
12
13 namespace leda {
14
15   template <typename P,
16             typename V,
17             typename C = stl_compare_less_pair<P, V>,
18             typename R = cphstl::priority_queue_framework<std::pair<
                   P, V>, C>,
19             typename J = cphstl::priority_queue_iterator<typename R
                   ::encapsulator_type,
20               typename R::component_type, true>
21   >
22   class p_queue {
23   public:
24     typedef P prio_type;
25     typedef V inf_type;
26     typedef C comparator_type;
27     typedef R realizator_type;
28     typedef J const_iterator;
29     typedef int size_type;
30
31     typedef typename R::encapsulator_type encapsulator_type;
32     typedef typename R::allocator_type allocator_type;
33     typedef encapsulator_type* item;
34     typedef encapsulator_type const* const_item;
35   private:
36     typedef typename allocator_type::template rebind<
         encapsulator_type>::other encapsulator_allocator_type;
37   public:
38     p_queue();
39     p_queue(int (*cmp)(P const&, P const&));
40     ~p_queue();
41
42     p_queue(p_queue const&);
43     p_queue& operator=(p_queue const&);
44
```

```
45      P const& prio(const_item) const;
46      V const& inf(const_item) const;
47
48      V& operator[](item);
49      item insert(P const&, V const&);
50
51      item find_min();
52      P del_min();
53
54      void del_item(item);
55      void change_inf(item, V const&);
56      void decrease_p(item, P const&);
57
58      size_type size();
59      bool empty();
60      void clear();
61
62      item first_item();
63      item last_item();
64      item next_item(item);
65      item prev_item(item);
66
67      const_iterator begin() const;
68      const_iterator end() const;
69
70    protected:
71      void destroy(item);
72      item create(P const&, V const&);
73
74    private:
75      R realizator;
76    };
77
78  }
79
80  #include "leda-p-queue.i++"
```

Appendix B.7 `leda-p-queue.i++`

```
1   /*
2     The implementation of leda::p_queue.
3
4     Authors: Bo Simonsen 2009
5   */
6
7   namespace leda {
8
9     /* Protected methods, destroy and create. Should be in a factory
          class,
10    ** such that the STL interface and LEDA interface can share these
          */
11
```

```
12  template <typename P, typename V, typename C, typename R, typename
        J>
13  void p_queue<P, V, C, R, J>::destroy(item it) {
14    encapsulator_allocator_type allocator = (*this).realizator.
          get_allocator();
15
16    (*it).~encapsulator_type();
17    allocator.deallocate(it, 1);
18  }
19
20  template <typename P, typename V, typename C, typename R, typename
        J>
21  typename p_queue<P, V, C, R, J>::item
22  p_queue<P, V, C, R, J>::create(P const& p, V const& i) {
23    encapsulator_allocator_type allocator = (*this).realizator.
          get_allocator();
24
25    item it = allocator.allocate(1);
26    try {
27      new (it) encapsulator_type(std::pair<P, V>(p, i), allocator);
28    }
29    catch (...) {
30      (*this).destroy(it);
31      throw;
32    }
33    return it;
34  }
35
36  /* *structors */
37
38  template <typename P, typename V, typename C, typename R, typename
        J>
39  p_queue<P, V, C, R, J>::p_queue()
40    : realizator(comparator_type(compare)) {
41  }
42
43  template <typename P, typename V, typename C, typename R, typename
        J>
44  p_queue<P, V, C, R, J>::p_queue(int (*cmp)(P const&, P const&))
45    : realizator(comparator_type(cmp)) {
46  }
47
48  template <typename P, typename V, typename C, typename R, typename
        J>
49  p_queue<P, V, C, R, J>::~p_queue() {
50    (*this).clear();
51  }
52
53  /* Note: These two member functions are not exception-safe */
54  template <typename P, typename V, typename C, typename R, typename
        J>
```

```
55  p_queue<P, V, C, R, J>::p_queue(p_queue const& p) : realizator(p.
       realizator.get_comparator()) {
56    for(J it = p.begin(); it != p.end(); ++it) {
57      (*this).insert(p.prio(it), p.inf(it));
58    }
59  }
60
61  template <typename P, typename V, typename C, typename R, typename
        J>
62  p_queue<P, V, C, R, J>&
63  p_queue<P, V, C, R, J>::operator=(p_queue const& p) {
64    /* Delete the old realizator */
65    (*this).clear();
66
67    for(J it = p.begin(); it != p.end(); ++it) {
68      (*this).insert(p.prio(it), p.inf(it));
69    }
70    return (*this);
71  }
72
73
74  /* Modifiers and accessors */
75
76  template <typename P, typename V, typename C, typename R, typename
        J>
77  P const& p_queue<P, V, C, R, J>::prio(const_item it) const {
78    return (*it).element().first;
79  }
80
81  template <typename P, typename V, typename C, typename R, typename
        J>
82  V const& p_queue<P, V, C, R, J>::inf(const_item it) const {
83    return (*it).element().second;
84  }
85
86  template <typename P, typename V, typename C, typename R, typename
        J>
87  V& p_queue<P, V, C, R, J>::operator[](item it) {
88    return (*it).element().second;
89  }
90
91  template <typename P, typename V, typename C, typename R, typename
        J>
92  typename p_queue<P, V, C, R, J>::item
93  p_queue<P, V, C, R, J>::insert(P const& p, V const& i) {
94    item tmp = (*this).create(p, i);
95    (*this).realizator.insert(tmp);
96    return tmp;
97  }
98
99  template <typename P, typename V, typename C, typename R, typename
        J>
```

```
100 typename p_queue<P, V, C, R, J>::item
101 p_queue<P, V, C, R, J>::find_min() {
102   return (*this).realizator.top();
103 }
104
105 template <typename P, typename V, typename C, typename R, typename
        J>
106 P p_queue<P, V, C, R, J>::del_min() {
107   item tmp = (*this).realizator.top();
108   (*this).realizator.extract(tmp);
109   P p = (*this).prio(tmp);
110   (*this).destroy(tmp);
111   return p;
112 }
113
114 template <typename P, typename V, typename C, typename R, typename
        J>
115 void p_queue<P, V, C, R, J>::del_item(item it) {
116   (*this).realizator.extract(it);
117   (*this).destroy(it);
118 }
119
120 template <typename P, typename V, typename C, typename R, typename
        J>
121 void p_queue<P, V, C, R, J>::change_inf(item it, V const& i) {
122   (*it).element().second = i;
123 }
124
125 template <typename P, typename V, typename C, typename R, typename
        J>
126 void p_queue<P, V, C, R, J>::decrease_p(item it, P const& p) {
127   (*this).realizator.increase(it, std::pair<P, V>(p, (*this).inf(
          it)));
128 }
129
130 template <typename P, typename V, typename C, typename R, typename
        J>
131 typename p_queue<P, V, C, R, J>::size_type
132 p_queue<P, V, C, R, J>::size() {
133   return (*this).realizator.size();
134 }
135
136 template <typename P, typename V, typename C, typename R, typename
        J>
137 bool p_queue<P, V, C, R, J>::empty() {
138   return (*this).size() == 0;
139 }
140
141 template <typename P, typename V, typename C, typename R, typename
        J>
142 void p_queue<P, V, C, R, J>::clear() {
143   while(!(*this).empty()) {
```

```
144       (*this).destroy((*this).realizator.extract());
145     }
146   }
147
148   /* LEDA Iterators */
149
150   template <typename P, typename V, typename C, typename R, typename
          J>
151   typename p_queue<P, V, C, R, J>::item
152   p_queue<P, V, C, R, J>::first_item() {
153     return (*this).realizator.begin();
154   }
155
156   template <typename P, typename V, typename C, typename R, typename
          J>
157   typename p_queue<P, V, C, R, J>::item
158   p_queue<P, V, C, R, J>::last_item() {
159     return (*this).prev_item((*this).realizator.end());
160   }
161
162   template <typename P, typename V, typename C, typename R, typename
          J>
163   typename p_queue<P, V, C, R, J>::item
164   p_queue<P, V, C, R, J>::next_item(item i) {
165     J tmp(i);
166     ++tmp;
167     return (item) static_cast<const_item>(tmp);
168   }
169
170   template <typename P, typename V, typename C, typename R, typename
          J>
171   typename p_queue<P, V, C, R, J>::item
172   p_queue<P, V, C, R, J>::prev_item(item i) {
173     throw std::domain_error("Backward iteration is not possible");
174   }
175
176   /* STL iterators */
177
178   template <typename P, typename V, typename C, typename R, typename
          J>
179   typename p_queue<P, V, C, R, J>::const_iterator
180   p_queue<P, V, C, R, J>::begin() const {
181     return (*this).realizator.begin();
182   }
183
184   template <typename P, typename V, typename C, typename R, typename
          J>
185   typename p_queue<P, V, C, R, J>::const_iterator
186   p_queue<P, V, C, R, J>::end() const {
187     return (*this).realizator.end();
188   }
189
```

```
190
191 }
```

Appendix B.8 `leda-queue.h++`

```
1 /*
2   leda::queue is an API for a queue that conforms to LEDA's
3   requirements; it is otherwise identical to its original
        counterpart,
4   except that it has one additional type parameter: a realizator
        type.
5
6   Authors: Jyrki Katajainen, Michael Neidhardt, and Bo Simonsen
        2008, 2009
7 */
8
9 #ifndef __LEDA_QUEUE__
10 #define __LEDA_QUEUE__
11
12 #include <iostream> // defines std::istream and std::ostream
13 #include "doubly-linked-list.h++"
14 #include "node-iterator.h++"
15
16 namespace leda {
17
18   template <
19     typename V,
20     typename R = cphstl::doubly_linked_list<V>,
21     typename I = cphstl::node_iterator<typename R::encapsulator_type
        , false>,
22     typename J = cphstl::node_iterator<typename R::encapsulator_type
        , true>
23   >
24   class queue {
25   public:
26
27     // types
28
29     typedef V value_type;
30     typedef R realizator_type;
31     typedef I iterator;
32     typedef J const_iterator;
33
34     typedef typename R::encapsulator_type* item;
35     typedef typename R::encapsulator_type const* const_item;
36     // structors
37
38     explicit queue();
39     queue(queue<V, R, I, J> const&);
40     ~queue();
41     queue<V, R, I, J>& operator=(queue<V, R, I, J> const&);
42
```

```
43     // items
44
45     const_item first_item() const;
46     item first_item();
47     const_item last_item() const;
48     item last_item();
49     const_item next_item(const_item) const;
50     item next_item(item);
51
52     // iterators
53
54     const_iterator begin() const;
55     iterator begin();
56     const_iterator end() const;
57     iterator end();
58
59     // accessors
60
61     int size() const;
62     int length() const;
63     bool empty() const;
64     V const& top() const;
65     V const& operator[](const_item) const;
66
67     // modifiers
68
69     V& operator[](item);
70     void append(V const&);
71     void push(V const&);
72     V pop();
73     void clear();
74
75     // operators
76
77     template <typename U, typename S, typename K, typename L>
78     friend std::istream& operator>>(std::istream& , queue<U, S, K, L
           >&);
79
80     template <typename U, typename S, typename K, typename L>
81     friend std::ostream& operator<<(std::ostream&, queue<U, S, K, L>
            const&);
82
83   protected:
84
85     realizator_type kernel;
86
87   };
88 }
89
90 #include "leda-queue.i++" // implements leda::queue
91
92 #endif
```

Appendix B.9 `leda-queue.i++`

```
1  /*
2    An implementation of leda::queue
3
4    Authors: Jyrki Katajainen, Michael Neidhardt, and Bo Simonsen
         2008, 2009
5  */
6
7  #include <iostream> // defines std::istream and std::ostream
8  #include <iterator> // defines std::istream_iterator and std::
       ostream_iterator
9  #include "leda-macros.i++" // defines leda macros
10
11 namespace leda {
12
13   // default constructor
14
15   template <typename V, typename R, typename I, typename J>
16   queue<V, R, I, J>::queue() : kernel() {
17   }
18
19   // copy constructor
20
21   template <typename V, typename R, typename I, typename J>
22   queue<V, R, I, J>::queue(queue<V, R, I, J> const& q) : kernel(q.
         kernel) {
23   }
24
25   // destructor
26
27   template <typename V, typename R, typename I, typename J>
28   queue<V, R, I, J>::~queue() {
29   }
30
31   // operator=
32
33   template <typename V, typename R, typename I, typename J>
34   queue<V, R, I, J>&
35   queue<V, R, I, J>::operator=(queue<V, R, I, J> const& q) {
36     (*this).kernel = q.kernel;
37     return *this;
38   }
39
40   // first
41
42   template <typename V, typename R, typename I, typename J>
43   typename queue<V, R, I, J>::const_item
44   queue<V, R, I, J>::first_item() const {
45     return (*this).kernel.begin();
46   }
47
```

```
48   template <typename V, typename R, typename I, typename J>
49   typename queue<V, R, I, J>::item
50   queue<V, R, I, J>::first_item() {
51     return (*this).kernel.begin();
52   }
53
54   // last
55
56   template <typename V, typename R, typename I, typename J>
57   typename queue<V, R, I, J>::const_item
58   queue<V, R, I, J>::last_item() const {
59     return (*this).kernel.end();
60   }
61
62   template <typename V, typename R, typename I, typename J>
63   typename queue<V, R, I, J>::item
64   queue<V, R, I, J>::last_item() {
65     return (*this).kernel.end();
66   }
67
68   // succ
69
70   template <typename V, typename R, typename I, typename J>
71   typename queue<V, R, I, J>::const_item
72   queue<V, R, I, J>::next_item(const_item p) const {
73     return (*p).successor();
74   }
75
76   template <typename V, typename R, typename I, typename J>
77   typename queue<V, R, I, J>::item
78   queue<V, R, I, J>::next_item(item p) {
79     return (*p).successor();
80   }
81
82   // size
83
84   template <typename V, typename R, typename I, typename J>
85   int
86   queue<V, R, I, J>::size() const {
87     return static_cast<int>((*this).kernel.size());
88   }
89
90   // length
91
92   template <typename V, typename R, typename I, typename J>
93   //typename queue<V, R, I, J>::size_type
94   int
95   queue<V, R, I, J>::length() const {
96     return (*this).size();
97   }
98
99   // empty
```

```
100
101   template <typename V, typename R, typename I, typename J>
102   bool
103   queue<V, R, I, J>::empty() const {
104     return (*this).size() == 0;
105   }
106
107   // top
108
109   template <typename V, typename R, typename I, typename J>
110   V const&
111   queue<V, R, I, J>::top() const {
112     return (*this)[(*this).kernel.begin()];
113   }
114
115   // operator[]
116
117   template <typename V, typename R, typename I, typename J>
118   V const&
119   queue<V, R, I, J>::operator[](const_item p) const {
120     return (*p).content();
121   }
122
123   template <typename V, typename R, typename I, typename J>
124   V&
125   queue<V, R, I, J>::operator[](item p) {
126     return (*p).content();
127   }
128
129   // append: inserts an element at the back of the queue
130
131   template <typename V, typename R, typename I, typename J>
132   void
133   queue<V, R, I, J>::append(V const& v) {
134     (*this).kernel.push_back(v);
135   }
136
137   // push: inserts an element at the front of the queue
138
139   template <typename V, typename R, typename I, typename J>
140   void
141   queue<V, R, I, J>::push(V const& v) {
142     (*this).kernel.push_front(v);
143   }
144
145   // pop
146
147   template <typename V, typename R, typename I, typename J>
148   V
149   queue<V, R, I, J>::pop() {
150     V element = (*this)[(*this).kernel.begin()];
151     (*this).kernel.erase((*this).kernel.begin());
```

```
152     return element;
153   }
154 
155   // clear
156 
157   template <typename V, typename R, typename I, typename J>
158   void
159   queue<V, R, I, J>::clear() {
160     while ((*this).kernel.size() != 0) {
161       (*this).kernel.erase((*this).kernel.begin());
162     }
163   }
164 
165   // operator>>
166 
167   template <typename V, typename R, typename I, typename J>
168   std::istream&
169   operator>>(std::istream& s, queue<V, R, I, J>& q) {
170     std::istream_iterator<V> input(s);
171     std::istream_iterator<V> eof;
172     while (input != eof) {
173       V element = *input;
174       q.push(element);
175       ++input;
176     }
177     return s;
178   }
179 
180   // operator<<
181 
182   template <typename V, typename R, typename I, typename J>
183   std::ostream&
184   operator<<(std::ostream& s, queue<V, R, I, J> const& q) {
185     queue<V, R, I, J> r;
186     typename queue<V, R, I, J>::const_item i = q.first_item();
187     for (int n = 0; n != q.size(); ++n) {
188       r.push(q[i]);
189       i = q.next_item(i);
190     }
191 
192     typename queue<V, R, I, J>::const_item j = r.first_item();
193     std::ostream_iterator<V> output(s, " ");
194     for (int n = 0; n != r.size(); ++n) {
195       *output = r[j];
196       ++output;
197       j = r.next_item(j);
198     }
199     s << std::endl;
200     return s;
201   }
202 
203 }
```

Appendix B.10 `leda-stack.h++`

```
1 /*
2   leda::stack is an API for a stack that conforms to LEDA's
3   requirements; it is otherwise identical to its original
       counterpart,
4   except that it has one additional type parameter: a realizator
       type.
5
6   Authors: Jyrki Katajainen, Michael Neidhardt, and Bo Simonsen
       2008, 2009
7 */
8
9 #ifndef __LEDA_STACK__
10 #define __LEDA_STACK__
11
12 #include <iostream> // defines std::istream and std::ostream
13 #include "doubly-linked-list.h++" // defines cphstl::
     doubly_linked_list
14
15 namespace leda {
16
17   template <typename V,
18             typename R = cphstl::doubly_linked_list<V> >
19   class stack {
20   public:
21
22     // structors
23
24     explicit stack();
25     stack(stack<V, R> const&);
26     ~stack();
27     stack<V, R>& operator=(stack<V, R> const&);
28
29     // accessors
30
31     int size() const;
32     bool empty() const;
33     V const& top();
34
35     // modifiers
36
37     void push(V const&);
38     V pop();
39     void clear();
40
41     // operators
42
43     template <typename U, typename S>
44     friend std::istream& operator>>(std::istream&, stack<U, S>&);
45
46     template <typename U, typename S>
```

```
47     friend std::ostream& operator<<(std::ostream&, stack<U, S> const
           &);
48
49   protected:
50
51     typedef R realizator_type;
52     realizator_type kernel;
53
54   };
55 }
56
57 #include "leda-stack.i++" // implements leda::stack
58
59 #endif
```

Appendix B.11 `leda-stack.i++`

```
1  /*
2    An implementation of leda::stack
3
4    Authors: Jyrki Katajainen, Michael Neidhardt, and Bo Simonsen 2009
5  */
6
7  #include <cassert> // defines assert macro
8  #include <iostream> // defines std::istream and std::ostream
9  #include <iterator> // defines std::istream_iterator and std::
       ostream_iterator
10
11 namespace leda {
12
13   // default constructor
14
15   template <typename V, typename R>
16   stack<V, R>::stack() : kernel() {
17   }
18
19   // copy constructor
20
21   template <typename V, typename R>
22   stack<V, R>::stack(stack<V, R> const& s) : kernel(s.kernel) {
23   }
24
25   // destructor
26
27   template <typename V, typename R>
28   stack<V, R>::~stack() {
29   }
30
31   // operator=
32
33   template <typename V, typename R>
34   stack<V, R>&
```

```
35  stack<V, R>::operator=(stack<V, R> const& s) {
36    (*this).kernel = s.kernel;
37    return *this;
38  }
39
40  // size
41
42  template <typename V, typename R>
43  int
44  stack<V, R>::size() const {
45    return static_cast<int>((*this).kernel.size());
46  }
47
48  // empty
49
50  template <typename V, typename R>
51  bool
52  stack<V, R>::empty() const {
53    return (*this).size() == 0;
54  }
55
56  // top
57
58  template <typename V, typename R>
59  V const&
60  stack<V, R>::top() {
61    assert(! (*this).empty());
62    return (*(*this).kernel.begin()).content();
63  }
64
65  // push
66
67  template <typename V, typename R>
68  void
69  stack<V, R>::push(V const& v) {
70    (*this).kernel.insert((*this).kernel.begin(), v);
71  }
72
73  // pop: deletes and returns the top element
74
75  template <typename V, typename R>
76  V
77  stack<V, R>::pop() {
78    assert(! (*this).empty());
79    V element = (*(*this).kernel.begin()).content();
80    (*this).kernel.erase((*this).kernel.begin());
81    return element;
82  }
83
84  // clear
85
86  template <typename V, typename R>
```

```
87    void
88    stack<V, R>::clear() {
89      while ((*this).kernel.size() != 0) {
90        (*this).kernel.erase((*this).kernel.begin());
91      }
92    }
93
94    // operator>>
95
96    template <typename V, typename R>
97    std::istream&
98    operator>>(std::istream& s, stack<V, R>& t) {
99      std::istream_iterator<V> input(s);
100     std::istream_iterator<V> eof;
101     while (input != eof) {
102       V element = *input;
103       t.push(element);
104       ++input;
105     }
106     return s;
107   }
108
109   // operator<<
110
111   template <typename V, typename R>
112   std::ostream&
113   operator<<(std::ostream& s, stack<V, R> const& t) {
114     stack<V, R> u(t);
115     stack<V, R> v;
116     while (! u.empty()) {
117       v.push(u.pop());
118     }
119     std::ostream_iterator<V> output(s, " ");
120     while (! v.empty()) {
121       *output = v.pop();
122       ++output;
123     }
124     s << std::endl;
125     return s;
126   }
127 }
```

Appendix C. Test programs

Appendix C.1 array-smoke-test.c++

```
1 /*
2   A smoke test for leda::array
3
4   Authors: Bo Simonsen 2009
5 */
6
```

```
7  #include <cassert>
8
9  #ifdef STL
10 #include "leda-array.h++"
11 #else
12 #include "LEDA/core/array.h"
13 #endif
14
15 template <typename V>
16 int test() {
17   leda::array<V> a1;
18   assert(a1.size() == 0);
19   leda::array<V> a2(10, 19);
20   assert(a2.size() == 10);
21   leda::array<V> a3(10);
22   assert(a3.size() == 10);
23   leda::array<V> a4(2, 1, 2);
24   assert(a4.size() == 2);
25   assert(a4[2] == 1);
26   assert(a4[3] == 2);
27   assert(a4.low() == 2);
28   assert(a4.high() == 3);
29   leda::array<V> a5(10, 1, 2, 3);
30   assert(a5.size() == 3);
31   assert(a5[10] == 1);
32   assert(a5[11] == 2);
33   assert(a5[12] == 3);
34   assert(a5.low() == 10);
35   assert(a5.high() == 12);
36   leda::array<V> a6(20, 1, 2, 3, 4);
37   assert(a6.size() == 4);
38   assert(a6[20] == 1);
39   assert(a6[21] == 2);
40   assert(a6[22] == 3);
41   assert(a6[23] == 4);
42   assert(a6.low() == 20);
43   assert(a6.high() == 23);
44
45   a1.resize(100);
46   a1.init(5);
47   for(int i = 0; i < 100; ++i) {
48     assert(a1.get(i) == 5);
49   }
50
51   /*a6.swap(23, 20);
52   assert(a6[23] == 1);
53   assert(a6[20] == 4);*/
54
55   a6.permute();
56   assert(a6.low() == 20);
57   assert(a6.high() == 23);
58
```

```
59  a6.print('\n');
60
61  for(int i=0; i <= 20; ++i) {
62    a1[i] = i;
63  }
64  a1.permute(0, 20);
65  a1.sort();
66  a1.unique();
67  std::cout << "A1" << std::endl;
68  for(int i=0; i <= 20; ++i) {
69    std::cout << "x: " << a1.get(i) << std::endl;
70  }
71  a1.resize(0, 20);
72
73  assert(a1.binary_search(5) == 5);
74  std::cout << a1.binary_search(11) << std::endl;
75  assert(a1.binary_search(11) == 11);
76  assert(a1.binary_search(20) == 20);
77
78  leda::array<int> a7(0,1,5,5,7);
79  std::cout << a7.binary_search(5) << std::endl;
80
81  a7.print("Test", '\n');
82
83  leda::array<int> a8(5);
84  a8.init(5);
85  /*std::cin >> a8;
86  std::cout << a8;*/
87  for(leda::array<int>::item it = a8.first_item();
88      it != a8.last_item();
89      it = a8.next_item(it)) {
90      std::cout << a8.inf(it) << std::endl;
91  }
92
93  leda::array<int> a9(a8);
94  std::lexicographical_compare(a9.begin(), a9.end(), a8.begin(), a8.
        end());
95  assert(a8.low() == a9.low());
96  assert(a8.high() == a9.high());
97  for(int i=a8.low(); i < a8.high(); ++i) {
98    assert(a8[i] == a9[i]);
99  }
100
101 leda::array<int> a10;
102 a10 = a9;
103 std::lexicographical_compare(a9.begin(), a9.end(), a10.begin(),
        a10.end());
104 assert(a10.low() == a9.low());
105 assert(a10.high() == a9.high());
106 for(int i=a10.low(); i < a10.high(); ++i) {
107   assert(a10[i] == a9[i]);
108 }
```

```
109
110
111 }
112
113 int main() {
114   test<int>();
115   //test<float>();
116
117 }
```

Appendix C.2 `dict-io-test.c++`

```
1  /*
2    A specific test for read and write routines of leda::dictionary
3
4    Authors: Jyrki Katajainen 2009
5  */
6
7  #include <iostream> // defines standard streams
8  #include "leda-dictionary.h++" // defines leda::dictionary
9
10 template <typename K, typename I>
11 void dictionary_test() {
12   leda::dictionary<K, I> d;
13
14   try {
15     std::cin >> d;
16     std::cout << d;
17   }
18   catch(...) {
19     std::cout << "Error!" << std::endl;
20   }
21 }
22
23 int main(int, char**) {
24   dictionary_test<int, int>();
25   return 0;
26 }
```

Appendix C.3 `dict-word-count.c++`

```
1  /*
2    The original version of this word-count program was taken from the
3    LEDA sources.
4
5    Author: Jyrki Katajainen 2009
6  */
7
8  #include <cassert> // defines assert macro
9  #include <iostream> // defines standard streams
10 //#include <LEDA/core/string.h> // defines leda::string
11 #include <string>
```

```
12 #ifdef STL
13 #include <leda-dictionary.h++> // defines leda::dictionary
14 #else
15 #include <LEDA/core/dictionary.h> // defines leda::dictionary
16 #include <LEDA/core/string.h>
17 using leda::string;
18 #endif
19
20 int main() {
21   typedef leda::dictionary<string, int> dictionary;
22   dictionary D;
23   string s;
24
25   while (std::cin >> s) {
26     dictionary::item p = D.lookup(s);
27     if (p == nil) {
28       D.insert(s, 1);
29     }
30     else {
31       D.change_inf(p, D.inf(p) + 1);
32     }
33   }
34
35   dictionary::iterator q;
36   for (q = D.begin(); q != D.end(); ++q) {
37     std::cout << (*q).first << ": " << (*q).second << std::endl;
38   }
39   std::cout << std::endl;
40
41   dictionary::item r = D.first_item();
42   forall_rev_items(r, D) {
43     std::cout <<  D.key(r) << ": " << D.inf(r) << std::endl;
44     D.del_item(r);
45   }
46   std::cout << std::endl;
47   std::cout << D.size() << std::endl;
48   assert(D.size() == 0);
49
50   return 0;
51 }
```

Appendix C.4 ***dict-smoke-test.c++***

```
1 /*
2   A smoke test for leda::dictionary
3
4   Authors: Jyrki Katajainen and Michael Neidhardt 2009
5
6   nil: does not work!
7
8 */
9
```

```
10 #ifdef STL
11 #include "leda-compare-functions.i++" // defines leda::compare
12 #include "leda-dictionary.h++" // defines leda::dictionary
13 #else
14 #include <limits.h>
15 #include "LEDA/core/dictionary.h"
16 #endif
17 #include <cassert>
18
19 using namespace leda;
20
21 template <typename K, typename I>
22 void dictionary_test() {
23   typedef typename dictionary<K, I>::item dic_item;
24   typedef typename dictionary<K, I>::iterator dic_iterator;
25
26   assert(compare(K(1), K(2)) == -1);
27   assert(compare(K(2), K(1)) == 1);
28   assert(compare(K(2), K(2)) == 0);
29
30   dictionary<K, I> d1;
31   assert(d1.empty() && d1.size() == 0);
32
33   dictionary<K, I> d2(d1);
34   assert(d2.empty() && d2.size() == 0);
35   d2.insert(K(1), I(1));
36   assert( !d2.empty() && d2.size() == 1);
37   d2.insert(K(2), I(1));
38   assert( !d2.empty() && d2.size() == 2);
39   d2.insert(K(2), I(1));
40   assert( !d2.empty() && d2.size() == 2);
41
42   dictionary<K, I> d3;
43   d3 = d2;
44   assert( !d3.empty() && d3.size() == d2.size());
45   dictionary<K, I> d4(d3);
46   assert( !d4.empty() && d4.size() == d3.size());
47
48   dic_item t = d1.lookup(K(0));
49   assert(t == nil);
50
51   d1.insert(K(1), I(2));
52   d1.insert(K(2), I(2));
53   d1.insert(K(1), I(1));
54   d1.insert(K(3), I(3));
55   d1.insert(K(4), I(4));
56
57   assert(! d1.empty() && d1.size() == 4);
58
59   t = d1.lookup(K(1));
60   assert(t != nil);
61   assert(d1.key(t) == K(1));
```

```
62    t = d1.next_item(t);
63    assert(d1.inf(t) == I(2));
64    assert(d1[t] == I(2));
65
66    t = d1.lookup(K(2));
67    //  assert(t != nil && d1[t] == K(2));
68
69    K n = 1;
70    for (dic_iterator p = d1.begin(); p != d1.end(); ++p, ++n) {
71      assert(d1.access(K(n)) == I(n));
72    }
73
74    d1.del(K(1));
75    assert(d1.size() == 3);
76    //  assert(d1.lookup(K(1)) == nil);
77
78    t = d1.lookup(K(4));
79    d1.del_item(t);
80    assert(d1.size() == 2);
81    //  assert(d1.lookup(K(4)) == nil);
82
83    assert(d1.defined(K(3)));
84    d1.undefine(K(3));
85    d1.undefine(K(3));
86    assert(! d1.defined(K(3)));
87    assert(d1.size() == 1);
88
89    d1.insert(K(99), I(1));
90    d1.clear();
91    assert(d1.empty() && d1.size() == 0);
92
93    d1.insert(K(1), I(10));
94    d1.insert(K(2), I(20));
95    d1.insert(K(3), I(30));
96    d1.insert(K(4), I(40));
97
98  #if 0
99    {
100     I x;
101     K i = 1;
102     forall(x, d1) {
103       assert(i * 10 == x);
104       ++i;
105     }
106   }
107
108   {
109     I i = 1;
110     K k;
111     forall_defined(k, d1) {
112       assert(i == k);
113       i++;
```

```
114       }
115     }
116     forall_items(t, d1) {
117       assert(K(10) * d1.key(t) == d1.inf(t));
118     }
119 #endif
120 
121     dic_item a = d1.first_item();
122     a = d1.next_item(a);
123     d1.change_inf(a, I(5));
124 }
125 
126 int tmp() {
127     dictionary<int, char> d;
128     d.insert(5, 'b');
129     std::cout << d.access(5) << std::endl;
130 }
131 
132 int main(int argc, char** argv) {
133     //dictionary_test<int, int>();
134     //dictionary_test<int, char>();
135     tmp();
136 
137     return 0;
138 }
```

Appendix C.5 *p-queue-smoke-test.c++*

```
1  /*
2    A smoke test for leda::p_queue
3  
4    Authors: Bo Simonsen 2009
5  */
6  
7  #include "leda-p-queue.h++"
8  
9  int main() {
10   leda::p_queue<int, char> Q;
11 
12   assert(Q.empty() == true);
13   Q.insert(5, 'a');
14   assert(Q.empty() == false);
15   assert(Q.size() == 1);
16   Q.insert(7, 'b');
17   assert(Q.size() == 2);
18   Q.insert(1, 'c');
19   assert(Q.size() == 3);
20   Q.insert(12, 'z');
21   assert(Q.size() == 4);
22 
23   assert(Q[Q.find_min()] == 'c');
24 
```

```
25   Q.del_min();
26   assert(Q[Q.find_min()] == 'a');
27   assert(Q.size() == 3);
28   Q.del_min();
29   assert(Q.size() == 2);
30   assert(Q.prio(Q.find_min()) == 7);
31   Q.decrease_p(Q.find_min(), 5);
32   assert(Q.prio(Q.find_min()) == 5);
33   Q.change_inf(Q.find_min(), 'z');
34   assert(Q[Q.find_min()] == 'z');
35
36   // Iteration don't work yet, I'll not accept last_item() to be
         done in
37   // O(n). It should be done in O(1)!
38
39   /*for(leda::p_queue<int, char>::item it = Q.first_item();
40       it != Q.last_item();
41       it = Q.next_item(it)) {
42     std::cout << Q.prio(it) << std::endl;
43   }*/
44
45   Q.insert(10, 'b');
46
47   for(leda::p_queue<int, char>::const_iterator it = Q.begin();
48       it != Q.end();
49       ++it) {
50     std::cout << (*it).first << std::endl;
51   }
52
53   leda::p_queue<int, char> Q2(Q);
54   assert(Q2.begin() != Q2.end());
55   assert(Q2.size() == Q.size());
56   std::lexicographical_compare(Q.begin(), Q.end(), Q2.begin(), Q2.
         end());
57
58   leda::p_queue<int, char> Q3;
59   Q3 = Q2;
60   assert(Q3.begin() != Q3.end());
61   assert(Q2.size() == Q3.size());
62   std::lexicographical_compare(Q3.begin(), Q3.end(), Q2.begin(), Q2.
         end());
63
64   Q.clear();
65   assert(Q.size() == 0);
66   assert(Q.empty() == true);
67
68
69
70 }
```

Appendix C.6 `queue-operator-test.c++`

```
1  /*
2    Testing push and operator>> for leda::queue
3
4    Authors: Jyrki Katajainen and Michael Neidhardt 2009
5  */
6
7  #include <cassert> // defines assert macro
8  #include <iostream> // defines standard streams
9  //#include <LEDA/core/string.h> // defines leda::string
10 #ifdef STL
11 #include <leda-queue.h++> // defines leda::queue
12 #else
13 #include <LEDA/core/queue.h> // defines leda::queue
14 #endif
15
16 int main(int argc, char** argv) {
17   leda::queue<int> Q;
18   int i;
19
20   while (std::cin >> i) {
21     Q.push(i);
22   }
23
24   std::cout << "Size: " << Q.size() << "\n";
25   std::cout << "Contents: " << Q << "\n";
26 }
```

Appendix C.7 `queue-io-test.c++`

```
1  /*
2    A specific test for read and write routines of leda::queue
3
4    Authors: Jyrki Katajainen and Michael Neidhardt 2009
5  */
6
7  #include <iostream> // defines standard streams
8  #include "leda-queue.h++" // defines leda::queue
9
10 template <typename V>
11 void test_queue() {
12   leda::queue<V> s;
13
14   std::cin >> s;
15   std::cout << s;
16 }
17
18 #ifndef ELEMENT
19 #define ELEMENT int
20 #endif
21
22 int main(int, char**) {
23   test_queue<ELEMENT>();
```

```
24   return 0;
25 }
```

Appendix C.8 `queue-smoke-test.c++`

```
1  /*
2    A smoke test for leda::queue
3
4    Authors: Jyrki Katajainen and Michael Neidhardt 2008, 2009
5  */
6
7  #include <cassert> // defines assert macro
8  #include "leda-queue.h++" // define leda::queue
9
10 // no equality operator in leda::queue
11
12 template<typename V>
13 bool operator==(leda::queue<V> const& q, leda::queue<V> const& r) {
14   if (q.size() != r.size()) {
15     return false;
16   }
17   typename leda::queue<V>::const_item i = q.first_item();
18   typename leda::queue<V>::const_item j = r.first_item();
19   for (int n = 0; n != q.size(); ++n) {
20     if (q[i] != r[j]) {
21       return false;
22     }
23     q.next_item(i);
24     r.next_item(j);
25   }
26   return true;
27 }
28
29 template <typename V>
30 void test_queue() {
31
32   // test default constructor
33
34   leda::queue<V> q;
35   assert(q.empty());
36   assert(q.size() == 0);
37
38   // test append
39
40   q.append(V(1));
41   assert(q.size() == 1);
42
43   // test copy constructor
44
45   leda::queue<V> r(q);
46   assert(r.size() == 1);
47
```

```
48    // test append
49
50    for (int i = 2; i <= 10; ++i) {
51      r.append(V(i));
52    }
53
54    // test assignment
55
56    leda::queue<V> s = r;
57    assert(s.size() == 10);
58
59    // test iterator operations
60
61    assert(r == s);
62
63    // test pop
64
65    while (r.size() != 0) {
66      assert(r.pop() == s.pop());
67      assert(r.size() == s.size());
68    }
69
70    // test top
71
72    V element = q.top();
73    assert(element == V(1));
74    V const constant = q.top();
75    assert(constant == V(1));
76
77    // test size and length
78
79    int n = s.size();
80    s.push(V(9));
81    assert(s.top() == V(9) && s.size() == n + 1);
82    assert(s.size() == s.length());
83
84    // test empty
85
86    leda::queue<V> t;
87    t.append(V(1));
88    t.append(V(2));
89    t.append(V(3));
90    assert(! t.empty() && t.size() == 3);
91    t.clear();
92    assert(t.empty() && t.size() == 0 && t.length() == 0);
93  }
94
95  int main(int, char**) {
96    test_queue<int>();
97    test_queue<char>();
98    return 0;
99  }
```

Appendix C.9 `stack-io-test.c++`

```
/*
  A specific test for read and write routines of leda::stack

  Authors: Jyrki Katajainen and Michael Neidhardt 2009
*/

#include <iostream> // defines standard streams
#include "leda-stack.h++" // defines leda::stack

template <typename V>
void test_stack() {
  typedef leda::stack<V> stack;

  stack s;

  std::cin >> s;
  std::cout << s;
}

#ifndef ELEMENT
#define ELEMENT int
#endif

int main(int, char**) {
  test_stack<ELEMENT>();
  return 0;
}
```

Appendix C.10 `stack-smoke-test.c++`

```
/*
  A smoke test for leda::stack

  Authors: Jyrki Katajainen and Michael Neidhardt 2008, 2009
*/

#include <cassert> // defines assert macro
#include "leda-stack.h++" // defines leda::stack

// no equality operator in leda::stack

template<typename V>
bool operator==(leda::stack<V> const& s, leda::stack<V> const& t) {
  if (s.size() != t.size()) {
    return false;
  }
  leda::stack<V> a(s);
  leda::stack<V> b(t);
  while (! a.empty()) {
    if (a.pop() != b.pop()) {
```

```
21        return false;
22      }
23    }
24    return true;
25  }
26
27  template <typename V>
28  void test_stack() {
29
30    // some random tests first
31
32    leda::stack<V> s;
33    for (int i = 1; i <= 10; ++i) {
34      s.push(V(i));
35    }
36    assert(s.size() == 10);
37
38    s.clear();
39    assert(s.size() == 0);
40
41    for (int i = 1; i <= 10; ++i) {
42      s.push(V(i));
43    }
44    while (! s.empty()) {
45      V const& r = s.top();
46      V e = s.pop();
47      assert(r == e);
48    }
49    assert(s.size() == 0);
50
51    leda::stack<V> t;
52    leda::stack<V> u;
53    assert(t.size() == 0);
54    assert(u.empty());
55    t.push(V(1));
56    assert(t.size() == 1);
57    u.push(V(1));
58
59    // test copy constructor
60
61    leda::stack<V> v(u);
62    assert(u == v);
63
64    // test assignment
65
66    t = u;
67    assert(t == u);
68
69    // test clear
70
71    t.clear();
72    assert(t.empty());
```

```
73
74   // test top
75
76   for (V i = 1; i <= 10; ++i) {
77     t.push(V(i));
78   }
79   V e = t.top();
80   assert(e == V(10));
81
82   // test pop
83
84   e = t.pop();
85   assert(t.top() == V(9));
86 }
87
88 int main(int, char**) {
89   test_stack<int>();
90   test_stack<char>();
91   return 0;
92 }
```