INFORMATIK

The LEDA User Manual (Version R 3.2)

Stefan Näher   Christian Uhrig

MPI–I–95–1-002   June 1995

FORSCHUNGSBERICHT ■ RESEARCH REPORT

ISSN: 0946 - 011X

Forschungsberichte des

Max-Planck-Instituts für Informatik

The LEDA User Manual (Version R 3.2)

Stefan Näher   Christian Uhrig

MPI–I–95–1-002   June 1995

The LEDA User Manual [1]

Version R 3.2

Stefan Näher and Christian Uhrig
Max-Planck-Institut für Informatik,
66123 Saarbrücken, Germany

[1]This research was supported by the ESPRIT II Basic Research Actions Program, under contract No. 3075 (project ALCOM)

Contents

Acknowledgement
The authors want to thank Kurt Mehlhorn, Stefan Schirra, and Christoph Burnikel for many helpful suggestions and valuable comments.

Chapter 0

Introduction

One of the major differences between combinatorial computing and other areas of computing such as statistics, numerical analysis and linear programming is the use of complex data types. Whilst the built-in types, such as integers, reals, vectors, and matrices, usually suffice in the other areas, combinatorial computing relies heavily on types like stacks, queues, dictionaries, sequences, sorted sequences, priority queues, graphs, points, segments, ... In the fall of 1988, we started a project (called **LEDA** for Library of Efficient Data types and Algorithms) to build a small, but growing library of data types and algorithms in a form which allows them to be used by non-experts. We hope that the system will narrow the gap between algorithms research, teaching, and implementation. The main features of LEDA are:

1. LEDA provides a sizable collection of data types and algorithms in a form which allows them to be used by non-experts. In the current version, this collection includes most of the data types and algorithms described in the text books of the area.

2. LEDA gives a precise and readable specification for each of the data types and algorithms mentioned above. The specifications are short (typically, not more than a page), general (so as to allow several implementations), and abstract (so as to hide all details of the implementation).

3. For many efficient data structures access by position is important. In LEDA, we use an item concept to cast positions into an abstract form. We mention that most of the specifications given in the LEDA manual use this concept, i.e., the concept is adequate for the description of many data types.

4. LEDA contains efficient implementations for each of the data types, e.g., Fibonacci heaps for priority queues, skip lists and dynamic perfect hashing for dictionaries, ...

5. LEDA contains a comfortable data type graph. It offers the standard iterations such as "for all nodes v of a graph G do" or "for all neighbors w of v do", it allows to add and delete vertices and edges and it offers arrays and matrices indexed by nodes and edges,... The data type graph allows to write programs for graph problems in a form close to the typical text book presentation.

6. LEDA is implemented by a C++ class library. It can be used with almost any C++ compiler that supports templates.
7. LEDA is available by anonymous ftp from **ftp.mpi-sb.mpg.de** in directory /pub/LEDA. The distribution contains all sources, installation instructions, a technical report, and the user manual.
8. LEDA is not in the public domain, but can be used freely for research and teaching. Information on a commercial license is available from the author or leda@mpi-sb.mpg.de.

This manual contains the specifications of all data types and algorithms currently available in LEDA. Users should be familiar with the C++ programming language (see [45] or [32]). The manual is structured as follows: In chapter one, which is a prerequisite for all other chapters, we discuss the basic concepts and notations used in LEDA. The other chapters define the data types and algorithms available in LEDA and give examples of their use. These chapters can be consulted independently from one another.

Version 3.0

The most important changes with respect to previous versions are

1. Parameterized data types are realized by C++ templates. In particular, *declare* macros used in previous versions are now obsolete and the syntax for a parameterized data type D with type parameters $T_1, \ldots, T_k$ is $D<T_1, \ldots, T_k>$ (cf. section 1.2).
2. Arbitrary data types (not only pointer and simple types) can be used as actual type parameters (cf. section 1.2).
3. For many of the parameterized data types (in the current version: dictionary, priority queue, d_array, and sortseq) there exist variants taking an additional data structure parameter for choosing a particular implementation (cf. section 1.3).
4. The LEDA memory management system can be customized for user-defined classes (cf. section 11.8).
5. The efficiency of many data types and algorithms has been improved.

See also the "Changes" file in the LEDA root directory.

Version 3.1

Many changes were made to make LEDA work with new compilers (g++ -2.6.3, Lucid C++ , Watcom C++ Sunpro C++ , ...) and on more platforms (Silicon Graphics, IBM, HP, Solaris-2.3, Linux, ...). All reported bugs of version 3.0 we were able to find and understand have been fixed (see LEDA/Fixes for a list). Several new data types and algorithms (especially in the graph and window section) have been included.

Manual Pages

All manual pages have been incorporated into the corresponding header files. There are tools (in the directory man) to extract and typeset the new user manual from these files. A postscript version of the manual is available on the ftp server.

Parameterized Data Types

The LEDA_TYPE_PARAMETER macro that had to be called for type arguments in parameterized data types when using the g++ compiler is not needed anymore for g++ versions >= 2.6.3.

Linear Orders, I/O, and Hashing

compare, *Hash*, *Read* and *Print* functions only have to be defined for type parameters of a parameterized data type if they are actually used by operations of the data type. If one of these function is called and has not been defined explicitely, a default version giving an error message is instantiated from a function template. Except for the built-in types and some basic LEDA types (*string* and *point*) there are no predefined versions of these functions any more. In particular, they are not predefined for arbitrary pointer types. You will notice the effect of this change, for instance, when calling the sort operation on a list of pointers $list\langle T*\rangle$ without a definition of a compare function for $T*$. Previous LEDA releases sorted the list by increasing addresses of the pointers in this case. In version 3.1 the program will exit with the exception "no compare function defined for $T*$". Operations based on functions *Hash*, *Read*, or *Print* show a similar behavior.

Nested Forall Loops

The limitation of previous versions that **forall**-loops (e.g. on lists) could not be nested is no longer valid.

Graphs

The distinction in directed and undirected graphs is not as strict as in previous versions. Data type *graph* represents both directed and undirected graphs. Directed and undirected graphs only differ in the definition of adjacent edges or nodes. Now, two lists of edges are associated with every node v: the list $out\_edges(v) = \{ e \in E \mid source(e) = v \}$ of edges starting in v, and the list $in\_edges(v) = \{ e \in E \mid target(e) = v \}$ of edges ending in v. A graph is either *directed* or *undirected*. In a directed graph an edge is adjacent to its source and in an undirected graph it is adjacent to its source and target. In a directed graph a node w is adjacent to a node v if there is an edge $(v,w) \in E$; in an undirected graph w is adjacent to v if there is an edge (v,w) or

(w, v) in the graph. There are iteration macros allowing to iterate over the list of starting, ending or adjacent edges (cf. 7.1 for details).

New Data Types

The old priority queue type *priority_queue*<K, I> caused a lot of confusion because of the non-standard semantics of the type parameters K and I (the meaning of *key* and *information* was exchanged compared to most text book presentations of priority queues). To eliminate this confusion we introduced a new priority queue type *p_queue*<P, I>. Now P is called the priority type of the queue.

The basic library has been extended by several numerical data types including an arbitrary length integer type (*integer*), a type of rational numbers (*rational*), and two filter types *floatf* and *real*. Together with the new types *rat_point* (points with rational coordinates) and *rat_segments* (segments with rational coordinates) they are especially useful in geometric computations. Note, that the geometric part of LEDA will be extended by more basic objects and algorithms based on exact arithmetic in the near future.

The temporarily (in LEDA-3.1c) introduced data types *node_data*, *node_stack*, *node_queue* are no longer available. Please use *node_map*, *edge_map*, *stack*<*node*> and *queue*<*node*> instead.

A list of the new data types:

- priority queues (*p_queue*<P, I>) (cf. 6.1)
- singly linked lists (*slist*<T>) (cf. 4.8)
- big integers (*integer*) (cf. 3.1)
- rational numbers (*rational*) (cf. 3.2)
- rational points (*rat_point*) (cf. 8.2)
- rational segments (*rat_segment*) (cf. 8.4)
- floating point filter (*floatf*) (cf. 3.3)
- random sources (*random_source*) (cf. 2.2)
- real numbers (*real*) (cf. 3.4)
- maps (*map*<I, E>) (cf. 5.8)
- node maps (*node_map*<T>) (cf. 7.9)
- edge maps (*edge_map*<T>) (cf. 7.10)
- node lists (*node_list*) (cf. 7.14)
- bounded node priority queues *b_node_pq*<*n*> (cf. 7.17).

Graph Generators and Algorithms

LEDA now offers more generators for different types of graphs (see 7.18 for a complete list). The planarity test *PLANAR* has been re-implemented and now has a boolean flag parameter *embed*. An embedding of the graph is computed only if $embed = true$ (the default value is $false$). A second variant of *PLANAR* computes a Kuratowsky-subgraph in the case of non-planarity. A new graph algorithm is *MIN_COST_MAX_FLOW* computing a maximal flow with minimal cost.

Windows and Panels

The window and panel data types now are based on a plain X11 implementation. New features include
- opening more than one window or panel
- positioning, displaying, and closing of panels and windows
- changing the label of windows and panels
- 16 predefined colors
- accessing colors from the X11 data base by name
- drawing arcs, arc edges, and arc arrows
- changing the default fonts,
- reading and handling X11 events
- a simple graph editor (cf. <LEDA/graph_edit.h>)

Chapter 1

Basics

1.1 A First Example

The following program can be compiled and linked with LEDA's basic library *libL.a* (cf. section 1.11). When executed it reads a sequence of strings from the standard input and then prints the number of occurrences of each string on the standard output. More examples of LEDA programs can be found throughout this manual.

```
#include <LEDA/d_array.h>
main()
{
  d_array<string,int> N(0);
  string s;
  while (cin >> s ) N[s]++;
  forall_defined(s,N) cout << s << " " << N[s] << endl;
}
```

The program above uses the parameterized data type dictionary array (*d_array*<I, E>) from the library. This is expressed by the include statement (cf. section 1.10 for more details). The specification of the data type *d_array* can be found in section 5.5. We use it also as a running example to discuss the principles underlying LEDA in sections 1.2 to 1.11.
Parameterized data types in LEDA are realized by templates, inheritance and dynamic binding. For C++ compilers not supporting templates there is still available a non-template version of LEDA using declare macros as described in [42].

1.2 Specifications

In general the specification of a LEDA data type consists of four parts: a definition of the set of objects comprising the (parameterized) abstract data type, a description of how to create an object of the data type, the definition of the operations available on the objects of the data type, and finally, information about the implementation. The four parts appear under the headers definition, creation, operations, and implementation respectively. Sometimes there is also a fifth part showing an example.

- **Definition**

  This part of the specification defines the objects (also called instances or elements) comprising the data type using standard mathematical concepts and notation.

  Example

  The generic data type dictionary array:

  An object a of type *d_array*<I, E> is an injective function from the data type I to the set of variables of data type E. The types I and E are called the index and the element type respectively, a is called a dictionary array from I to E.

  Note that the types I and E are parameters in the definition above. Any built-in, pointer, item, or user-defined class type T can be used as actual type parameter of a parameterized data type. Class types however have to provide the following operations:

a) a constructor taking no arguments	T::T()
b) a copy constructor	T::T(const T&)
c) an assignment operator	T& T::operator=(const T&)

and if required by the parameterized data type

d) an input function	void **Read**(T&, istream&)
e) an output function	void **Print**(const T&, ostream&)
f) a compare function	int **compare**(const T&, const T&)
g) a hash function	int **Hash**(const T&)

For details see sections 1.6 and 1.7.

- **Creation**

  A variable of a data type is introduced by a C++ variable declaration. For all LEDA data types variables are initialized at the time of declaration. In many cases the user has to provide arguments used for the initialization of the variable. In general a declaration

  XYZ<$t_1, \ldots, t_k$> $y(x_1, \ldots, x_\ell)$;

  introduces a variable y of the data type "XYZ<$t_1, \ldots, t_k$>" and uses the arguments $x_1, \ldots, x_\ell$ to initialize it. For example,

  d_array<*string*, *int*> $A(0)$

  introduces A as a dictionary array from strings to integers, and initializes A as follows: an injective function a from *string* to the set of unused variables of type *int* is constructed, and is assigned to A. Moreover, all variables in the range of a are initialized to 0. The reader may wonder how LEDA handles an array of infinite size. The solution is, of course, that only that part of A is explicitly stored which has been accessed already.

  For all data types, the assignment operator (=) is available for variables of that type. Note however that assignment is in general not a constant time operation, e.g., if L_1 and L_2 are variables of type *list*<T> then the assignment $L_1 = L_2$

takes time proportional to the length of the list L_2 times the time required for copying an object of type T.

Remark: For most of the complex data types of LEDA, e.g., dictionaries, lists, and priority queues, it is convenient to interpret a variable name as the name for an object of the data type which evolves over time by means of the operations applied to it. This is appropriate, whenever the operations on a data type only "modify" the values of variables, e.g., it is more natural to say an operation on a dictionary D modifies D than to say that it takes the old value of D, constructs a new dictionary out of it, and assigns the new value to D. Of course, both interpretations are equivalent. From this more object-oriented point of view, a variable declaration, e.g., *dictionary<string, int>* D, is creating a new dictionary object with name D rather than introducing a new variable of type *dictionary<string, int>*; hence the name "creation" for this part of a specification.

- **Operations**

In this section the operations of the data types are described. For each operation the description consists of two parts

1. The interface of the operation is defined using the C++ function declaration syntax. In this syntax the result type of the operation (*void* if there is no result) is followed by the operation name and an argument list specifying the type of each argument. For example,

   list_item L.insert (E x, *list_item it*, *int dir* = *after*)
   defines the interface of the insert operation on a list L of elements of type E (cf. section 4.7). Insert takes as arguments an element x of type E, a *list_item it* and an optional relative position argument *dir*. It returns a *list_item* as result.

   $E\&$ $A[I\ x]$
   defines the interface of the access operation on a dictionary array A. It takes an element x of type I as an argument and returns a variable of type E.

2. The effect of the operation is defined. Often the arguments have to fulfill certain preconditions. If such a condition is violated the effect of the operation is undefined. Some, but not all, of these cases result in error messages and abnormal termination of the program (see also section 11.9). For the insert operation on lists this definition reads:
   A new item with contents x is inserted after (if *dir* = *after*) or before (if *dir* = *before*) item *it* into L. The new item is returned.

   Precondition: item *it* must be in L.

   For the access operation on dictionary arrays the definition reads:
   returns the variable $A(x)$.

- **Implementation**

The implementation section lists the (default) data structures used to implement the data type and gives the time bounds for the operations and the space requirement. For example,

Dictionary arrays are implemented by randomized search trees ([1]). Access operations $A[x]$ take time $O(\log dom(A))$. The space requirement is $O(dom(A))$.

1.3 Implementation Parameters

For many of the parameterized data types (in the current version: dictionary, priority queue, d_array, and sortseq) there exist variants taking an additional data structure parameter for choosing a particular implementation (cf. chapter 5.2, 5.4, 5.6 and 6.2). Since C++ does not allow to overload templates we had to use different names: the variants with an additional implementation parameters start with an underscore, e.g., _d_array<I,E,impl>. We can easily modify the example program from section 1.1 to use a dictionary array implemented by a particular data structure, e.g., skip lists ([43]), instead of the default data structure (cf. section 13.1).

```
#include <LEDA/d_array.h>
#include <LEDA/impl/skiplist.h>
main()
{
  _d_array<string,int,skiplist> N(0);
  string s;
  while (cin >> s) N[s]++;
  forall_defined(s,N) cout << s << " " << N[s] << endl;
}
```

Any type _XYZ<$T_1, \ldots, T_k$, *xyz_impl*> is derived from the corresponding "normal" parameterized type XYZ<$T_1, \ldots, T_k$>, i.e., an instance of type _XYZ<$T_1, \ldots, T_k$, *xyz_impl*> can be passed as argument to functions with a formal parameter of type XYZ<$T_1, \ldots, T_k$>&. This provides a mechanism for choosing implementations of data types in pre-compiled algorithms. See "prog/graph/dijkstra.c" for an example.

LEDA offers several implementations for each of the data types. For instance, skip lists, randomized search trees, and red-black trees for dictionary arrays. Users can also provide their own implementation. A data structure "xyz_impl" can be used as actual implementation parameter for a data type *_XYZ* if it provides a certain set of operations and uses certain virtual functions for type dependent operations (e.g. compare, initialize, copy, ...). Chapter 13 lists all data structures contained in the current version and gives the exact requirements for implementations of dictionaries, priority_queues, sorted sequences and dictionary arrays. A detailed description of the mechanism for parameterized data types and implementation parameters used in LEDA will be published soon.

1.4 Arguments

- **Optional Arguments**

  The trailing arguments in the argument list of an operation may be optional. If these trailing arguments are missing in a call of an operation the default argument values given in the specification are used. For example, if the relative position argument in the list insert operation is missing it is assumed to have the value *after*, i.e., L.insert(it, y) will insert the item $<y>$ after item *it* into L.

- **Argument Passing**

  There are two kinds of argument passing in C++ , by value and by reference. An argument x of type *type* specified by "*type* x" in the argument list of an operation or user defined function will be passed by value, i.e., the operation or function is provided with a copy of x. The syntax for specifying an argument passed by reference is "*type*& x". In this case the operation or function works directly on x (the variable x is passed not its value).

  Passing by reference must always be used if the operation is to change the value of the argument. It should always be used for passing large objects such as lists, arrays, graphs and other LEDA data types to functions. Otherwise a complete copy of the actual argument is made, which takes time proportional to its size, whereas passing by reference always takes constant time.

- **Functions as Arguments**

  Some operations take functions as arguments. For instance the bucket sort operation on lists requires a function which maps the elements of the list into an interval of integers. We use the C++ syntax to define the type of a function argument f:

  $$T \quad (*f)(T_1, T_2, \ldots, T_k)$$

  declares f to be a function taking k arguments of the data types T_1, ..., T_k, respectively, and returning a result of type T, i.e, $f : T_1 \times \ldots \times T_k \longrightarrow T$.

1.5 Overloading

Operation and function names may be overloaded, i.e., there can be different interfaces for the same operation. An example is the translate operations for points (cf. section 8).

point p.translate(*vector* v)
point p.translate(*double* α, *double* *dist*)

It can either be called with a vector as argument or with two arguments of type *double* specifying the direction and the distance of the translation.

An important overloaded function is discussed in the next section: Function *compare*, used to define linear orders for data types.

1.6 Linear Orders

Many data types, such as dictionaries, priority queues, and sorted sequences require linearly ordered parameter types. Whenever a type T is used in such a situation, e.g. in *dictionary<T,...>* the function

$$int\ \ compare(const\ T\&,\ const\ T\&)$$

must be declared and must define a linear order on the data type T.

A binary relation *rel* on a set T is called a linear order on T if for all $x, y, z \in T$:

1) x *rel* x
2) x *rel* y and y *rel* z implies x *rel* z
3) x *rel* y or y *rel* x
4) x *rel* y and y *rel* x implies $x = y$

A function *int compare(const T&, const T&)* defines the linear order *rel* on T if

$$compare(x,y)\ \begin{cases} < 0, & \text{if } x \ rel\ y \text{ and } x \neq y \\ = 0, & \text{if } x = y \\ > 0, & \text{if } y\ rel\ x \text{ and } x \neq y \end{cases}$$

For each of the simple data types *char*, *short*, *int*, *long*, *float*, *double*, *string*, and *point* a function *compare* is predefined and defines the so-called default ordering on that type. The default ordering is the usual $\leq$ - order for the built-in numerical types, the lexicographic ordering for *string*, and for *point* the lexicographic ordering of the cartesian coordinates. For all other types T there is no default ordering, and the user has to provide a *compare* function whenever a linear order on T is required.

Example: Suppose pairs of real numbers shall be used as keys in a dictionary with the lexicographic order of their components. First we declare class *pair* as the type of pairs of real numbers, then we define the I/O operations *Read* and *Print* and the lexicographic order on *pair* by writing an appropriate *compare* function.

```
class pair {
  double x;
  double y;
public:
  pair() { x = y = 0; }
  pair(const pair& p) { x = p.x; y = p.y; }
  friend void Read(pair& p, istream& is) { is >> p.x >> p.y; }
  friend void Print(const pair& p, ostream& os) { os << p.x << " " << p.y; }
  friend int compare(const pair&, const pair&);
};

int compare(const pair& p, const pair& q)
```

```
{
  if (p.x < q.x) return -1;
  if (p.x > q.x) return 1;
  if (p.y < q.y) return -1;
  if (p.y > q.y) return 1;
  return 0;
}
```

Now we can use dictionaries with key type *pair*, e.g.,

```
dictionary<pair,int> D;
```

Sometimes, a user may need additional linear orders on a data type T which are different from the order defined by *compare*, e.g., he might want to order points in the plane by the lexicographic ordering of their cartesian coordinates and by their polar coordinates. In this example, the former ordering is the default ordering for points. The user can introduce an alternative ordering on the data type *point* (cf. section 8) by defining an appropriate comparing function *int cmp*(const *point&*, const *point&*) and then calling the macro

DEFINE_LINEAR_ORDER(*point*, *cmp*, $point_1$).

After this call $point_1$ is a new data type which is equivalent to the data type *point*, with the only exception that if $point_1$ is used as an actual parameter e.g. in *dictionary<*$point_1, \ldots$*>*, the resulting data type is based on the linear order defined by *cmp*.

In general the macro call

DEFINE_LINEAR_ORDER(T, *cmp*, T_1)

introduces a new type T_1 equivalent to type T with the linear order defined by the compare function *cmp*.

In the example, we first declare a function *pol_cmp* and derive a new type *pol_point* using the DEFINE_LINEAR_ORDER macro.

```
int pol_cmp(const point& x, const point& y)
{ // lexicographic ordering on polar coordinates }

DEFINE_LINEAR_ORDER(point,pol_cmp,pol_point)
```

Now, dictionaries based on either ordering can be used.

```
dictionary<pol_point, int> D1;  // polar ordering
dictionary<point, int> D0;      // default ordering
```

Remark: We have chosen to associate a fixed linear order with most of the simple types (by predefining the function *compare*). This order is used whenever operations require a linear order on the type, e.g., the operations on a dictionary. Alternatively, we could have required the user to specify a linear order each time he uses a simple type in a situation where an ordering is needed, e.g., a user could define

$dictionary\langle point, lexicographic\_ordering, \ldots\rangle$

This alternative would handle the cases where two or more different orderings are needed more elegantly. However, we have chosen the first alternative because of the smaller implementation effort.

1.7 Hashed Types

LEDA also contains parameterized data types requiring a hash function for the actual type parameters. Examples are dictionaries implemented by hashing with chaining ($\_dictionary\langle K, I, ch\_hashing\rangle$) or hashing arrays ($h\_array\langle I, E\rangle$). Whenever a type T is used in such a context, e.g., in $h\_array\langle T, \ldots\rangle$ a function

$$int \;\; Hash(const \; T\&)$$

has to be defined that maps the elements of type T to integers. It is not required that $Hash$ is a perfect hash function, i.e., it has not to be injective. However, the performance of the underlying implementations very strongly depends on the ability of the function to keep different elements of T apart by assigning them different integers. Typically, a search operation in a hashing implementation takes time linear in the maximal size of any subset whose elements are assigned the same hash value. For each of the simple numerical data types char, short, int, long there is a predefined $Hash$ function: the identity function.

We demonstrate the use of $Hash$ and a data type based on hashing by extending the example from the previous section. Suppose we want to associate information with values of the $pair$ class by using a hashing array $h\_array\langle pair, int\rangle \; A$. We first define a hash function that assigns each pair (x, y) the integral part of the first component x

int Hash(const pair& p) { return *int(p.x)*; }

and then can use a hashing array with index type *pair*

$h\_array\langle pair, int\rangle \; A;$

1.8 Items

Many of the advanced data types in LEDA (dictionaries, priority queues, graphs, ...), are defined in terms of so-called items. An item is a container which can hold an object relevant for the data type. For example, in the case of dictionaries a *dic_item* contains a pair consisting of a key and an information. A general definition of items will be given at the end of this section.

We now discuss the role of items for the dictionary example in some detail. A popular specification of dictionaries defines a dictionary as a partial function from some type K to some other type I, or alternatively, as a set of pairs from $K \times I$, i.e., as the graph of the function. In an implementation each pair (k, i) in the dictionary is stored in some location of the memory. Efficiency dictates that the pair (k, i) cannot only be accessed through the key k but sometimes also through the location where it is stored, e.g., we might want to lookup the information i associated with key k (this involves a search in the data structure), then compute with the value i a new value i', and finally associate the new value with k. This either involves another search in the data structure or, if the lookup returned the location where the pair (k, i) is stored, can be done by direct access. Of course, the second solution is more efficient and we therefore wanted to provide it in LEDA.

In LEDA items play the role of positions or locations in data structures. Thus an object of type *dictionary*$<K, I>$, where K and I are types, is defined as a collection of items (type *dic_item*) where each item contains a pair in $K \times I$. We use $<k, i>$ to denote an item with key k and information i and require that for each $k \in K$ there is at most one $i \in I$ such that $<k, i>$ is in the dictionary. In mathematical terms this definition may be rephrased as follows: A dictionary d is a partial function from the set *dic_item* to the set $K \times I$. Moreover, for each $k \in K$ there is at most one $i \in I$ such that the pair (k, i) is in d.

The functionality of the operations

dic_item	D.lookup(K k)
I	D.inf(*dic_item* *it*)
void	D.change_inf(*dic_item* *it*, I i')

is now as follows: D.lookup(k) returns an item *it* with contents (k, i), D.inf(*it*) extracts i from *it*, and a new value i' can be associated with k by D.change_inf(*it*, i').

Let us have a look at the insert operation for dictionaries next:

dic_item D.insert(K k, I i)

There are two cases to consider. If D contains an item *it* with contents (k, i') then i' is replaced by i and *it* is returned. If D contains no such item, then a new item, i.e., an item which is not contained in any dictionary, is added to D, this item is made to contain (k, i) and is returned. In this manual (cf. section 5.1) all of this is abbreviated to

dic_item	D.insert(K k, I i)	associates the information i with the key k. If there is an item $<k, j>$ in D then j is replaced by i, else a new item $<k, i>$ is added to D. In both cases the item is returned.

We now turn to a general discussion. With some LEDA types XYZ there is an associated type *XYZ_item* of items. Nothing is known about the objects of type *XYZ_item* except that there are infinitely many of them. The only operations available on *XYZ_items* besides the one defined in the specification of type XYZ

is the equality predicate "==" and the assignment operator "=" . The objects of type XYZ are defined as sets or sequences of *XYZ_items* containing objects of some other type Z. In this situation an *XYZ_item* containing an object $z \in Z$ is denoted by <z>. A new or unused *XYZ_item* is any *XYZ_item* which is not part of any object of type XYZ.

Remark: For some readers it may be useful to interpret a *dic_item* as a pointer to a variable of type $K \times I$. The differences are that the assignment to the variable contained in a *dic_item* is restricted, e.g., the K-component cannot be changed, and that in return for this restriction the access to *dic_items* is more flexible than for ordinary variables, e.g., access through the value of the K-component is possible.

1.9 Iteration

For many data types LEDA provides iteration macros. These macros can be used to iterate over the elements of lists, sets and dictionaries or the nodes and edges of a graph. Iteration macros can be used similarly to the C++ **for** statement with the restriction that inside the body of a loop the corresponding object must not be altered. For instance, it is not allowed to delete nodes from a graph G inside the body of a **forall_nodes** loop. Examples are

for all item based data types:

forall_items(it, D) { the items of D are successively assigned to variable it }

for lists and sets:

forall(x, L) { the elements of L are successively assigned to x }

for graphs:

forall_nodes(v, G) { the nodes of G are successively assigned to v }

forall_edges(e, G) { the edges of G are successively assigned to e }

forall_adj_edges(e, v) { all edges adjacent to v are successively assigned to e }

1.10 Header Files

LEDA data types and algorithms can be used in any C++ program as described in this manual. The specifications (class declarations) are contained in header files. To use a specific data type its header file has to be included into the program. In general the header file for data type xyz is <LEDA/xyz.h>. Exceptions to this rule can be found in Tables 14.1 and 14.2.

1.11 Libraries

The implementions of all LEDA data types and algorithms are precompiled and contained in 4 libraries (libL.a, libG.a, libP.a, libWx.a) which can be linked with

C++ application programs. In the following description it is assumed that these libraries are installed in one of the systems default library directories (e.g. /usr/lib), which allows to use the "-l..." compiler option.

a) **libL.a**
is the main LEDA library, it contains the implementations of all simple data types (chapter 2), basic data types (chapter 4), dictionaries and priority queues (chapter 5 and 6). A program *prog.c* using any of these data types has to be linked with the libL.a library like this:

CC *prog.c* -lL -lm

b) **libG.a**
is the LEDA graph library. It contains the implementations of all graph data types and algorithms (chapter 7). To compile a program using any graph data type or algorithm the libG.a and libL.a library have to be used:

CC *prog.c* -lG -lL -lm

c) **libP.a**
is the LEDA library for geometry in the plane. It contains the implementations of all data types and algorithms for two-dimensional geometry (chapter 8 and 9). To compile a program using geometric data types or algorithms the libP.a, libG.a, libL.a and maths libraries have to be used:

CC *prog.c* -lP -lG -lL -lm

d) **libWx.a**
is the LEDA library for graphic windows under the X11 window system. Application programs using data type *window* (cf. section 10.2) have to be linked with this library:

CC *prog.c* -lP -lG -lL -lWx -lX11 -lm

Note that the libraries must be given in the order -lP -lG -lL and that the window library (-lWx) has to appear after the plane library (-lP).

Chapter 2

Simple Data Types

2.1 Strings (string)

1. Definition

An instance s of the data type *string* is a sequence of characters (type *char*). The number of characters in the sequence is called the length of s. A string of length zero is called the empty string. Strings can be used wherever a C++ *char∗* string can be used.

Strings differ from the C++ type *char∗* in several aspects: parameter passing by value and assignment works properly (i.e., the value is passed or assigned and not a pointer to the value) and *strings* offer many additional operations.

2. Creation

string s;

introduces a variable s of type *string*. s is initialized with the empty string.

string s(*char* ∗ p);

introduces a variable s of type *string*. s is initialized with a copy of the C++ string p.

string s(*char* ∗ *format*, ...);

introduces a variable s of type *string*. s is initialized with the string produced by printf(*format*,. . .).

string s(*char* c);

introduces a variable s of type *string*. s is initialized with the one-character string "c".

3. Operations

int	s.length()	returns the length of string s.
char&	s [*int* i]	returns the character at position i. *Precondition*: $0 \le i \le s$.length()-1.
string	s (*int* i, *int* j)	returns the substring of s starting at position $\max(0, i)$ and ending at position $\min(j, s$.length()$-1)$. If $\min(j, s$.length()$-1) < \max(0, i)$ then the empty string is returned.
string	s.head(*int* i)	returns the first i characters of s.
string	s.tail(*int* i)	returns the last i characters of s.
int	s.pos(*string* $s1$, *int* i)	returns the minimum j such that $j \ge i$ and s_1 is a substring of s starting at position j (returns -1 if no such j exists).
int	s.pos(*string* $s1$)	returns pos$(s1, 0)$.
string	s.insert(*int* i, *string* $s1$)	returns $s(0, i-1) + s_1 + s(i, s$.length()$-1)$.
string	s.replace(*string* $s1$, *string* $s2$, *int* $i = 1$)	returns the string created from s by replacing the i-th occurrence of s_1 in s by s_2.
string	s.replace(*int* i, *int* j , *string* $s1$)	returns the string created from s by replacing $s(i, j)$ by s_1. *Precondition*: $i \le j$.
string	s.replace(*int* i, *string* $s1$)	returns the string created from s by replacing $s[i]$ by s_1.
string	s.replace_all(*string* $s1$, *string* $s2$)	returns the string created from s by replacing all occurrences of s_1 in s by s_2. *Precondition*: The occurrences of s_1 in s do not overlap (it's hard to say what the function returns if the precondition is violated.).
string	s.del(*string* $s1$, *int* $i = 1$)	returns s.replace$(s_1, "", i)$.
string	s.del(*int* i , *int* j)	returns s.replace$(i, j, "")$.
string	s.del(*int* i)	returns s.replace$(i, "")$.

string	s.del_all(*string* $s1$)	returns s.replace_all(s_1,"").
void	s.read(*istream&* I, *char delim* $=$ ' ')	reads characters from input stream I into s until the first occurrence of character *delim*.
void	s.read(*char delim* $=$ ' ')	read(*cin*,*delim*).
void	s.read_line(*istream&* I)	read(I,'\n').
void	s.read_line()	read_line(*cin*).
string	$x+y$	returns the concatenation of x and y.
string&	$s \mathrel{+}= x$	appends x to s and returns a reference to s.
bool	$x == y$	true iff x and y are equal.
bool	$x \mathrel{!}= y$	true iff x and y are not equal.
bool	$x < y$	true iff x is lexicographically smaller than y.
bool	$x > y$	true iff x is lexicographically greater than y.
bool	$x <= y$	returns $(x < y) \mid\mid (x == y)$.
bool	$x >= y$	returns $(x > y) \mid\mid (x == y)$.
istream&	*istream&* I >> & s	read(I,' ').
ostream&	*ostream&* O << s	writes string s to the output stream O.

4. Implementation

Strings are implemented by C++ character vectors. All operations involving the search for a pattern $s1$ in a string s take time $O(s.lenght() * s1.length())$, [ ] takes constant time and all other operations on a string s take time $O(s.length())$.

2.2 Random Sources (random_source)

1. Definition

An instance of type *random_source* is a random source. It allows to generate uniformly distributed random bits, characters, integers, and doubles. It can be in either of two modes: In bit mode it generates a random bit string of some given length p ($1 \leq p \leq 31$) and in integer mode it generates a random integer in some given range $[low..high]$ ($low \leq high < low + 2^{31}$). The mode can be changed any time, either globally or for a single operation. The output of the random source can be converted to a number of formats (using standard conversions).

2. Creation

random_source S;

creates an instance S of type *random_source*, puts it into bit mode, and sets the precision to 31.

random_source $S(int\ p)$;

creates an instance S of type *random_source*, puts it into bit mode, and sets the precision to p ($1 \leq p \leq 31$).

random_source $S(int\ low,\ int\ high)$;

creates an instance S of type *random_source*, puts it into integer mode, and sets the range to $[low..high]$.

3. Operations

void	S.set_seed(*int s*)	resets the seed of the random number generator to s.
void	S.set_range(*int low*, *int high*)	sets the mode to integer mode and changes the range to $[low..high]$.
void	S.set_precision(*int p*)	sets the mode to bit mode and changes the precision to p bits.
random_source&	S >> *char& x*	extracts a character x of default precision or range and returns S, i.e., it first generates an unsigned integer of the desired precision or in the desired range and then converts it to a character (by standard conversion).
random_source&	S >> *unsigned char& x*	extracts an unsigned character x of default precision or range and returns S.

random_source&	S >> *int&* x	extracts an integer x of default precision or range and returns S.
random_source&	S >> *unsigned int&* x	extracts an unsigned integer x of default precision or range and returns S.
random_source&	S >> *double&* x	extracts a real number x in $[0,1]$, i.e, $u/(2^{31}-1)$ where u is a random integer in $[0..2^{31}-1]$, and returns S.
random_source&	S >> *bool&* b	extracts a random boolean value (true or false).
unsigned	S.get()	returns an unsigned integer of maximal precision (31 bits).
int	S ()	returns an integer x.
int	S (*int prec*)	returns an integer x of supplied precision *prec*.
int	S (*int low*, *int high*)	returns an integer x from the supplied range [*low..high*].

2.3 Real-Valued Vectors (vector)

1. Definition

An instance of the data type *vector* is a vector of real variables.

2. Creation

vector v;

creates an instance v of type *vector*; v is initialized to the zero-dimensional vector.

vector v(*int d*);

creates an instance v of type *vector*; v is initialized to the zero vector of dimension d.

vector v(*double a*, *double b*);

creates an instance v of type *vector*; v is initialized to the two-dimensional vector (a, b).

vector v(*double a*, *double b*, *double c*);

creates an instance v of type *vector*; v is initialized to the three-dimensional vector (a, b, c).

3. Operations

int	v.dim()	returns the dimension of v.
double	v.length()	returns the Euclidean length of v.
double	v.angle(*vector w*)	returns the angle between v and w.
double&	v [*int i*]	returns i-th component of v. *Precondition*: $0 \leq i \leq v$.dim()-1.
vector	$v + v1$	Addition. *Precondition*: v.dim() = $v1$.dim().
vector	$v - v1$	Subtraction. *Precondition*: v.dim() = $v1$.dim().
double	$v * v1$	Scalar multiplication. *Precondition*: v.dim() = $v1$.dim().
vector	v * *double r*	Componentwise multiplication with double r.
bool	$v == w$	Test for equality.
bool	v != w	Test for inequality.
ostream&	*ostream& O* << v	writes v componentwise to the output stream O.
istream&	*istream& I* >> & v	reads v componentwise from the input stream I.

4. Implementation

Vectors are implemented by arrays of real numbers. All operations on a vector v take time $O(v.dim())$, except for dim and [ ] which take constant time. The space requirement is $O(v.dim())$.

2.4 Real-Valued Matrices (matrix)

1. Definition

An instance of the data type *matrix* is a matrix of double variables.

2. Creation

matrix M(*int* $n = 0$, *int* $m = 0$);

creates an instance M of type *matrix*, M is initialized to the $n \times m$ - zero matrix.

3. Operations

int	M.dim1()	returns n, the number of rows of M.
int	M.dim2()	returns m, the number of columns of M.
vector&	M.row(*int* i)	returns the i-th row of M (an m-vector). *Precondition*: $0 \le i \le n-1$.
vector	M.col(*int* i)	returns the i-th column of M (an n-vector). *Precondition*: $0 \le i \le m-1$.
matrix	M.trans()	returns M^T ($m \times n$ - matrix).
matrix	M.inv()	returns the inverse matrix of M. *Precondition*: M.det() $\neq 0$.
double	M.det()	returns the determinant of M. *Precondition*: M is quadratic.
vector	M.solve(*vector* b)	returns vector x with $M \cdot x = b$. *Precondition*: M.dim1() $=$ M.dim2() $=$ b.dim() and M.det() $\neq 0$.
double&	M (*int* i, *int* j)	returns $M_{i,j}$. *Precondition*: $0 \le i \le n-1$ and $0 \le j \le m-1$.
matrix	$M + M1$	Addition. *Precondition*: M.dim1() $=$ $M1$.dim1() and M.dim2() $=$ $M1$.dim2().
matrix	$M - M1$	Subtraction. *Precondition*: M.dim1() $=$ $M1$.dim1() and M.dim2() $=$ $M1$.dim2().
matrix	$M * M1$	Multiplication. *Precondition*: M.dim2() $=$ $M1$.dim1().
vector	M * *vector vec*	Multiplication with vector. *Precondition*: M.dim2() $=$ *vec*.dim().

matrix	*M* $*$ *double x*	Multiplication with double x.
ostream&	*ostream& O* $<<$ *M*	writes matrix M row by row to the output stream O.
istream&	*istream& I* $>>$ & *M*	reads matrix M row by row from the input stream I.

4. Implementation

Data type *matrix* is implemented by two-dimensional arrays of double numbers. Operations det, solve, and inv take time $O(n^3)$, dim1, dim2, row, and col take constant time, all other operations take time $O(nm)$. The space requirement is $O(nm)$.

Chapter 3

Number Types

3.1 Integers of Arbitrary Length (integer)

1. Definition

An instance *a* of the data type *integer* is an integer number of arbitrary length.

2. Creation

integer *a*;

creates an instance *a* of type *integer* and initializes it with zero.

integer *a*(*int* *n*);

creates an instance *a* of type *integer* and initializes it with the value of *n*.

integer *a*(*unsigned int* *i*);

creates an instance *a* of type *integer* and initializes it with the value of *i*.

integer *a*(*double* *x*);

creates an instance *a* of type *integer* and initializes it with the integral part of *x*.

3. Operations

The arithmetic operations $+$, $-$, $*$, $/$, $+=$, $-=$, $*=$, $/=$, $-$(unary), $++$, $--$, the modulus operation (%, % =), bitwise AND (&, & =), bitwise OR (|, | =), the complement (˜), the shift operations (<<, >>), the comparison operations $<$, $<=$, $>$, $>=$, $==$, $!=$ and the stream operations all are available.

int	*a*.length()	returns the number of bits of the representation of *a*.

bool	*a*.islong()	returns whether *a* fits in the data type *long*.
bool	*a*.iszero()	returns whether *a* is equal to zero.
long	*a*.tolong()	returns a *long* number which is initialized with the value of *a*. *Precondition*: *a*.islong() is *true*.
double	*a*.todouble()	returns a *double* number which is initialized with the value of *a*. *Precondition*: *a* fits in the range of a *double*.
integer	*a*.sqrt()	returns the largest *integer* which is not larger than the squareroot of *a*.

Non-member functions

integer	abs(*integer a*)	returns the absolute value of *a*.
integer	gcd(*integer a*, *integer b*)	returns the greatest common divisor of *a* and *b*.
int	sign(*integer a*)	returns the sign of *a*.
int	log(*integer a*)	returns the logarithm of *a* to the basis 2.

4. Implementation

An *integer* is essentially implemented by a vector *vec* of *unsigned long* numbers. The sign and the size are stored in extra variables. Some time critical functions are also implemented in sparc assembler code.

3.2 Rational Numbers (rational)

1. Definition

An instance q of type *rational* is a rational number where the numerator and the denominator are both of type *integer*.

2. Creation

rational q;

creates an instance q of type *rational*.

rational q(*double* x);

creates an instance q of type *rational* and initializes it with the value of x.

rational q(*int* n);

creates an instance q of type *rational* and initializes it with the value of n.

rational q(*int* m, *int* n);

creates an instance q of type *rational* and initializes its numerator with m and its denominator with n.

rational q(*integer* a);

creates an instance q of type *rational* and initializes it with the value of a.

rational q(*integer* a, *integer* b);

creates an instance q of type *rational* and initializes its numerator with a and its denominator with b.

3. Operations

The arithmetic operations $+$, $-$, $*$, $/$, $+=$, $-=$, $*=$, $/=$, $-$(unary), $++$, $--$, the comparison operations $<$, $<=$, $>$, $>=$, $==$, $!=$ and the stream operations are all available.

integer	q.numerator()	returns the numerator of q.
integer	q.denominator()	returns the denominator of q.
rational&	q.simplify(*integer* a)	simplifies q by a. *Precondition*: a divides the numerator and the denominator of q.
rational&	q.normalize()	normalizes q.

void	*q*.negate()	negates *q*.
void	*q*.invert()	inverts *q*.
rational	*q*.inverse()	returns the inverse of *q*.

Non-member functions

int	sign(*rational q*)	returns the sign of *q*.
rational	abs(*rational q*)	returns the absolute value of *q*.
rational	sqr(*rational q*)	returns the square of *q*.
rational	pow(*rational q*, *int n*)	returns the *n*-th power of *q*.
rational	pow(*rational q*, *integer a*)	returns the *a*-th power of *q*.
integer	trunc(*rational q*)	returns the *integer* with the next smaller absolute value.
integer	floor(*rational q*)	returns the next smaller *integer*.
integer	ceil(*rational q*)	returns the next bigger *integer*.
integer	round(*rational q*)	rounds *q* to the nearest *integer*.

4. Implementation

A *rational* is implemented by two *integer* numbers which represent the numerator and the denominator. The sign is represented by the sign of the numerator.

3.3 A Floating Point Filter (floatf)

1. Definition

The type *floatf* provides a clean and efficient way to approximately compute with large integers. Consider an expression E with integer operands and operators $+, -$, and $*$, and suppose that we want to determine the sign of E. In general, the integer arithmetic provided by our machines does not suffice to evaluate E since intermediate results might overflow. Resorting to arbitrary precision integer arithmetic is a costly process. An alternative is to evaluate the expression using floating point arithmetic, i.e., to convert the operands to doubles and to use floating-point addition, subtraction, and multiplication. Of course, only an approximation $\tilde{E}$ of the true value E is computed. However, $\tilde{E}$ might still be able to tell us something about the sign of E. If $\tilde{E}$ is far away from zero (the forward error analysis carried out in the next section gives a precise meaning to "far away") then the signs of $\tilde{E}$ and E agree and if $\tilde{E}$ is zero then we may be able to conclude under certain circumstances that E is zero. Again, forward error analysis can be used to say what 'certain circumstances' are. The type *floatf* encapsulates this kind of approximate integer arithmetic. Any integer (= object of type *integer*) can be converted to a *floatf*; *floatf*s can be added, subtracted, multiplied, and their sign can be computed: for any *floatf* x the function $Sign(x)$ returns either the sign of x (-1 if $x < 0$, 0 if $x = 0$, and $+1$ if $x > 0$) or the special value NO_IDEA. If x approximates X, i.e., X is the integer value obtained by an exact computation, then $Sign(x)! = NO\_IDEA$ implies that $Sign(x)$ is actually the sign of X if $Sign(x) = NO\_IDEA$ then no claim is made about the sign of X.

2. Creation

floatf x;

introduces a variable x of type *floatf* and initializes it with zero.

floatf $x(integer\ i)$;

introduces a variable x of type *floatf* and initializes it with integer i.

3. Operations

floatf	$a + b$	Addition.
floatf	$a - b$	Subtraction.
floatf	$a * b$	Multiplication.
int	Sign(*floatf* f)	as described above.

4. Implementation

A *floatf* is represented by a double (its value) and an error bound. An operation on *floatf*s performs the corresponding operation on the values and also computes the error bound for the result. For this reason the cost of a *floatf* operation is about four times the cost of the corresponding operation on doubles. The rules used to compute the error bounds are described in ([40]).

5. Example

see [40] for an application in a sweep line algorithm.

3.4 Real Numbers (real)

1. Definition

An instance x of the data type *real* is an algebraic real number. There are many ways to construct a *real*: either by conversion from *double*, *integer* or *rational* or by applying one of the arithmetic operators $+, -, *, /$ or $\sqrt{\ }$ to *real* numbers. One may test the sign of a *real* number or compare two *real* numbers by any of the comparison relations $=, \neq, <, \leq, >$ and $\geq$. The outcome of such a test is *exact*. There is also a non-standard version of the sign function: the call $x.sign(integer\ q)$ computes the sign of x under the precondition that $|x| \leq 2^{-q}$ implies $x = 0$. This version of the sign function allows the user to assist the data type in the computation of the sign of x, cf. the example below.

One can ask for *double* approximations of a real number x. The calls $x.todouble()$ and $x.get\_double\_error()$ return *doubles* $xnum$ and $xerr$ such that $|xnum - x| \leq |xnum| * xerr$. Note that $xerr = \infty$ is possible. There are also several functions to compute more accurate approximations of *reals*. The call $x.get\_precision()$ returns an *integer* $xerr$ such that the internal approximation x_num satisfies $|xnum - x| \leq 2^{-xerr}$. The user may set a bound on $xerr$. More precisely, after the call $x.improve(integer\ q)$ the data type guarantees $xerr \leq 2^{-q}$.

2. Creation

real x;

introduces a variable x of type *real* and initializes it to zero.

real $x(double\ y)$;

introduces a variable x of type *real* and initializes it to the value of y.

real $x(int\ n)$;

introduces a variable x of type *real* and initializes it to the value of n.

real $x(integer\ a)$;

introduces a variable x of type *real* and initializes it to the value of a.

real $x(rational\ q)$;

introduces a variable x of type *real* and initializes it to the value of q

3. Operations

The arithmetic operations $+$, $-$, $*$, $/$, $+=$, $-=$, $*=$, $/=$, $-$(unary), the comparison operations $<$, $<=$, $>$, $>=$, $==$, $!=$ and the stream operations all are available.

real	x.sqrt(*real*)	squareroot operation.
int	x.sign()	returns -1 if (the exact value of) $x < 0$, 1 if $x > 0$, and 0 if $x = 0$.
int	x.sign(*integer a*)	as above. Precondition: if $\lvert x\rvert \leq 2^{-a}$ then $x = 0$.
void	x.improve(*integer a*)	(re-)computes the approximation of x such that its final quality is bounded by a, i.e., $x.get\_precision() \geq a$ after the call $x.improve(a)$.
void	x.compute_in(*long k*)	(re-)computes the approximation of x; each numerical operation is carried out with k binary places.
void	x.compute_up_to(*long k*)	(re-)computes an approximation of x such that the error of the approximation lies in the k-th binary place.
double	x.todouble()	returns the current double approximation of x.
double	x.get_double_error()	returns the quality of the current double approximation of x, i.e., $\lvert x - x.todouble()\rvert \leq x.get\_double\_error() * \lvert x.todouble()\rvert$.
integer	x.get_precision()	returns the quality of the current internal approximation x.*num* of x, i.e., $\lvert x - x.num\rvert \leq 2^{-x.get\_precision()}$.

Non-member functions

real	fabs(*real& x*)	absolute value of x.
real	sq(*real x*)	square of x.
real	hypot(*real x*, *real y*)	euclidean distance of vector (x,y) to the origin.
real	powi(*real x*, *int n*)	n-th power of x.

4. Implementation

A *real* is represented by the expression which defines it and a *double* approximation $\hat{x}$ together with a relative error bound ϵ_x. The arithmetic operators $+, -, *, /, \sqrt{\ }$ take constant time. When the sign of a *real* number needs to be determined, the data type first computes a number q, if not already given as an argument to *sign*, such that $|x| \leq q$ implies $x = 0$. The bound q is computed as described in [37]. The data type then computes an internal approximation *xnum* for x with error bound $xerr \leq q$. The sign of *xnum* is then returned as the sign of x.

Two shortcuts are used to speed up the computation of the sign. Firstly, if the *double* approximation already suffices to determine the sign, then no further approximation is computed at all. Secondly, the internal approximation is first computed only with small precision. The precision is then doubled until either the sign can be decided (i.e., if $xerr < |xnum|$) or the full precision q is reached.

5. Example

Examples can be found in [9].

Chapter 4

Basic Data Types

4.1 One Dimensional Arrays (array)

1. Definition

An instance A of the parameterized data type *array<E>* is a mapping from an interval $I = [a..b]$ of integers, the index set of A, to the set of variables of data type E, the element type of A. $A(i)$ is called the element at position i.

2. Creation

array<E> $A(int\ a,\ int\ b)$;

creates an instance A of type *array<E>* with index set $[a..b]$.

array<E> $A(int\ n)$;

creates an instance A of type *array<E>* with index set $[0..n-1]$.

array<E> A;

creates an instance A of type *array<E>* with empty index set.

3. Operations

E&	$A\ [int\ x]$	returns $A(x)$. *Precondition:* $a \leq x \leq b$.
int	A.low()	returns the minimal index a.
int	A.high()	returns the maximal index b.
void	A.sort(*int* (**cmp*)(*const E&*, *const E&*))	sorts the elements of A, using function *cmp* to compare two elements, i.e., if $(in_a, \ldots, in_b)$ and $(out_a, \ldots, out_b)$ denote the values of the variables $(A(a), \ldots, A(b))$ before and after the call of sort, then $cmp(out_i, out_j) \leq 0$ for $i \leq j$ and there is a permutation π of $[a..b]$ such that $out_i = in_{\pi(i)}$ for $a \leq i \leq b$.

void	A.sort()	sorts the elements of A according to the linear order of the element type E. *Precondition*: A linear order on E must have been defined by $compare(const E\&, const E\&)$.
void	A.sort(*int* (**cmp*)(*const E&*, *const E&*), *int l*, *int h*)	sorts sub-array $A[l..h]$ using compare function *cmp*.
void	A.sort(*int l*, *int h*)	sorts sub-array $A[l..h]$ using the linear order on E.
void	A.permute()	the elemens of A are randomly permuted.
void	A.permute(*int l*, *int h*)	the elements of $A[l..h]$ are randomly permuted.
int	A.binary_search(*int* (**cmp*)(*const E&*, *const E&*), *E x*)	performs a binary search for x. Returns i with $A[i] = x$ if x in A, A.low()-1 otherwise. Function *cmp* is used to compare two elements. *Precondition*: A must be sorted according to *cmp*.
int	A.binary_search(*E x*)	performs a binary search for x using the default linear order on E. *Precondition*: A must be sorted.
void	A.read(*istream& I*)	reads $b - a + 1$ objects of type E from the input stream I into the array A using the overloaded *Read* function (cf. section 1.5).
void	A.read()	calls A.read(*cin*) to read A from the standard input stream *cin*.
void	A.read(*string s*)	As above, uses string s as prompt.
void	A.print(*ostream& O*, *char space* = ' ')	prints the contents of array A to the output stream O using the overloaded *Print* function (cf. section 1.5) to print each element. The elements are separated by the character *space*.
void	A.print(*char space* = ' ')	calls A.print(*cout*, *space*) to print A on the standard output stream *cout*.
void	A.print(*string s*, *char space* = ' ')	As above, uses string s as header.

4. Implementation

Arrays are implemented by C++ vectors. The access operation takes time $O(1)$, the sorting is realized by quicksort (time $O(n \log n)$) and the binary_search operation takes time $O(\log n)$, where $n = b - a + 1$. The space requirement is $O(|I| * sizeof(E))$.

4.2 Two Dimensional Arrays (array2)

1. Definition

An instance A of the parameterized data type *array2<E>* is a mapping from a set of pairs $I = [a..b] \times [c..d]$, called the index set of A, to the set of variables of data type E, called the element type of A, for two fixed intervals of integers $[a..b]$ and $[c..d]$. $A(i,j)$ is called the element at position (i,j).

2. Creation

array2<E> A(*int a*, *int b*, *int c*, *int d*);

creates an instance A of type *array2<E>* with index set $[a..b] \times [c..d]$.

array2<E> A(*int n*, *int m*);

creates an instance A of type *array2<E>* with index set $[0..n-1] \times [0..m-1]$.

3. Operations

E&	A (*int i*, *int j*)	returns $A(i,j)$. *Precondition*: $a \le i \le b$ and $c \le j \le d$.
int	A.low1()	returns a.
int	A.high1()	returns b.
int	A.low2()	returns c.
int	A.high2()	returns d.

4. Implementation

Two dimensional arrays are implemented by C++ vectors. All operations take time $O(1)$, the space requirement is $O(|I| * sizeof(E))$.

4.3 Stacks (stack)

1. Definition

An instance S of the parameterized data type *stack<E>* is a sequence of elements of data type E, called the element type of S. Insertions or deletions of elements take place only at one end of the sequence, called the top of S. The size of S is the length of the sequence, a stack of size zero is called the empty stack.

2. Creation

stack<E> S;

creates an instance S of type *stack<E>*. S is initialized with the empty stack.

3. Operations

E	S.top()	returns the top element of S. *Precondition*: S is not empty.
void	S.push(E x)	adds x as new top element to S.
E	S.pop()	deletes and returns the top element of S. *Precondition*: S is not empty.
int	S.size()	returns the size of S.
bool	S.empty()	returns true if S is empty, false otherwise.
void	S.clear()	makes S the empty stack.

4. Implementation

Stacks are implemented by singly linked linear lists. All operations take time $O(1)$, except clear which takes time $O(n)$, where n is the size of the stack.

4.4 Queues (queue)

1. Definition

An instance Q of the parameterized data type *queue<E>* is a sequence of elements of data type E, called the element type of Q. Elements are inserted at one end (the rear) and deleted at the other end (the front) of Q. The size of Q is the length of the sequence; a queue of size zero is called the empty queue.

2. Creation

queue<E> Q;

creates an instance Q of type *queue<E>*. Q is initialized with the empty queue.

3. Operations

E	Q.top()	returns the front element of Q. *Precondition*: Q is not empty.
E	Q.pop()	deletes and returns the front element of Q. *Precondition*: Q is not empty.
void	Q.append(E x)	appends x to the rear end of Q.
int	Q.size()	returns the size of Q.
bool	Q.empty()	returns true if Q is empty, false otherwise.
void	Q.clear()	makes Q the empty queue.

4. Implementation

Queues are implemented by singly linked linear lists. All operations take time $O(1)$, except clear which takes time $O(n)$, where n is the size of the queue.

4.5 Bounded Stacks (b_stack)

1. Definition

An instance S of the parameterized data type *b_stack<E>* is a stack (see section 4.3) of bounded size.

2. Creation

b_stack<E> $S(int\ n)$;

creates an instance S of type *b_stack<E>* that can hold up to n elements. S is initialized with the empty stack.

3. Operations

E	S.top()	returns the top element of S. *Precondition*: S is not empty.
E	S.pop()	deletes and returns the top element of S. *Precondition*: S is not empty.
void	S.push(E x)	adds x as new top element to S. *Precondition*: S.size() $< n$.
void	S.clear()	makes S the empty stack.
int	S.size()	returns the size of S.
bool	S.empty()	returns true if S is empty, false otherwise.

4. Implementation

Bounded stacks are implemented by C++ vectors. All operations take time $O(1)$. The space requirement is $O(n)$.

4.6 Bounded Queues (b_queue)

1. Definition

An instance Q of the parameterized data type *b_queue<E>* is a queue (see section 4.4) of bounded size.

2. Creation

b_queue<E> $Q(int\ n)$;

creates an instance Q of type *b_queue<E>* that can hold up to n elements. Q is initialized with the empty queue.

3. Operations

E	Q.top()	returns the front element of Q. *Precondition*: Q is not empty.
E	Q.pop()	deletes and returns the front element of Q. *Precondition*: Q is not empty.
void	Q.append(E& x)	appends x to the rear end of Q. *Precondition*: Q.size()$< n$.
void	Q.clear()	makes Q the empty queue.
int	Q.size()	returns the size of Q.
bool	Q.empty()	returns true if Q is empty, false otherwise.

4. Implementation

Bounded queues are implemented by circular arrays. All operations take time $O(1)$. The space requirement is $O(n)$.

4.7 Linear Lists (list)

1. Definition

An instance L of the parameterized data type *list<E>* is a sequence of items (*list_item*). Each item in L contains an element of data type E, called the element type of L. The number of items in L is called the length of L. If L has length zero it is called the empty list. In the sequel $<x>$ is used to denote a list item containing the element x and $L[i]$ is used to denote the contents of list item i in L.

2. Creation

list<E> L;

creates an instance L of type *list<E>* and initializes it to the empty list.

3. Operations

3.1 Access Operations

int	L.length()	returns the length of L.
int	L.size()	returns L.length().
bool	L.empty()	returns true if L is empty, false otherwise.
list_item	L.first()	returns the first item of L.
list_item	L.last()	returns the last item of L.
list_item	L.succ(*list_item it*)	returns the successor item of item *it*, nil if *it* = L.last(). *Precondition*: *it* is an item in L.
list_item	L.pred(*list_item it*)	returns the predecessor item of item *it*, nil if *it* = L.first(). *Precondition*: *it* is an item in L.
list_item	L.cyclic_succ(*list_item it*)	returns the cyclic successor of item *it*, i.e., L.first() if *it* = L.last(), L.succ(*it*) otherwise.
list_item	L.cyclic_pred(*list_item it*)	returns the cyclic predecessor of item *it*, i.e, L.last() if *it* = L.first(), L.pred(*it*) otherwise.
list_item	L.search(E x)	returns the first item of L that contains x, nil if x is not an element of L. *Precondition*: compare has to be defined for type E.
E	L.contents(*list_item it*)	returns the contents $L[it]$ of item *it*. *Precondition*: *it* is an item in L.

E	*L*.inf(*list_item it*)	returns *L*.contents(*it*).
E	*L*.head()	returns the first element of *L*, i.e. the contents of *L*.first(). *Precondition*: *L* is not empty.
E	*L*.tail()	returns the last element of *L*, i.e. the contents of *L*.last(). *Precondition*: *L* is not empty.
int	*L*.rank(*E* x)	returns the rank of x in L, i.e. its first position in L as an integer from $[1\ldots\lvert L\rvert]$ (0 if x is not in L).

3.2 Update Operations

list_item	*L*.push(*E* x)	adds a new item <x> at the front of L and returns it (L.insert(x, L.first$(), before$)).
list_item	*L*.append(*E* x)	appends a new item <x> to L and returns it (L.insert(x, L.last$(), after$)).
list_item	*L*.insert(*E a*, *list_item l*, *int dir* = 0)	inserts a new item <x> after (if $dir = after$) or before (if $dir = before$) item *it* into L and returns it (here *after* and *before* are predefined *int* constants). *Precondition*: *it* is an item in L.
E	*L*.pop()	deletes the first item from L and returns its contents. *Precondition*: L is not empty.
E	*L*.Pop()	deletes the last item from L and returns its contents. *Precondition*: L is not empty.
E	*L*.del_item(*list_item it*)	deletes the item *it* from L and returns its contents $L[it]$. *Precondition*: *it* is an item in L.
void	*L*.assign(*list_item it*, *E* x)	makes x the contents of item *it*. *Precondition*: *it* is an item in L.
void	*L*.conc(*list<E>& L*1)	appends list L_1 to list L and makes L_1 the empty list. *Precondition*: $L \neq L_1$
void	*L*.split(*list_item it*, *list<E>& L*1, *list<E>& L*2)	splits L at item *it* into lists $L1$ and $L2$. More precisely, if $it \neq nil$ and $L = x_1, \ldots, x_{k-1}, it, x_{k+1}, \ldots, x_n$ then $L1 = x_1, \ldots, x_{k-1}$ and $L2 = it, x_{k+1}, \ldots, x_n$. If $it = nil$ then $L1$ is made empty and $L2$ a copy of L. Finally L is made empty if it is not identical to $L1$ or $L2$. *Precondition*: *it* is an item of L or *nil*.

void	*L*.split(*list_item it*, *list<E>& L1*, *list<E>& L2*, *int dir*)	splits *L* at item *it* into lists *L*1 and *L*2. Item *it* becomes the last item of *L*1 if *dir* == 0 and the first item of *L*2 otherwise. *Precondition*: *it* is an item of *L*.
void	*L*.sort(*int* (**cmp*)(*const E&*, *const E&*))	sorts the items of *L* using the ordering defined by the compare function $cmp : E \times E \longrightarrow int$, with $cmp(a,b) \begin{cases} < 0, & \text{if } a < b \\ = 0, & \text{if } a = b \\ > 0, & \text{if } a > b \end{cases}$ More precisely, if $(in_1, \ldots, in_n)$ and $(out_1, \ldots, out_n)$ denote the values of *L* before and after the call of sort, then $cmp(L[out_j], L[out_{j+1}]) \leq 0$ for $1 \leq j < n$ and there is a permutation π of $[1..n]$ such that $out_i = in_{\pi_i}$ for $1 \leq i \leq n$.
void	*L*.sort()	sorts the items of *L* using the default ordering of type *E*, i.e., the linear order defined by function *int* compare(*const E&*, *const E&*).
list_item	*L*.min()	returns the item with the minimal contents with respect to the default linear order of type *E*.
list_item	*L*.min(*int* (**cmp*)(*const E&*, *const E&*))	returns the item with the minimal contents with respect to the linear order defined by compare function *cmp*.
list_item	*L*.max()	returns the item with the maximal contents with respect to the default linear order of type *E*.
list_item	*L*.max(*int* (**cmp*)(*const E&*, *const E&*))	returns the item with the maximal contents with respect to the linear order defined by compare function *cmp*.
void	*L*.apply(*void* (**f*)(*E& x*))	for all items <*x*> in *L* function *f* is called with argument *x* (passed by reference).
void	*L*.bucket_sort(*int i*, *int j*, *int* (**f*)(*const E&*))	sorts the items of *L* using bucket sort, where $f : E \longrightarrow int$ with $f(x) \in [i..j]$ for all elements *x* of *L*. The sort is stable, i.e., if $f(x) = f(y)$ and <*x*> is before <*y*> in *L* then <*x*> is before <*y*> after the sort.

void	L.permute()	the items of L are randomly permuted.
void	L.clear()	makes L the empty list.

3.3 Input and Output

void	L.read(*istream& I*, *char delim* = '\n')	reads a sequence of objects of type E terminated by the delimiter *delim* from the input stream I using the overloaded *Read* function (section 1.5). L is made a list of appropriate length and the sequence is stored in L.
void	L.read(*char delim* = '\n')	calls L.read(*cin*, *delim*) to read L from the standard input stream *cin*.
void	L.read(*string s*, *char delim* = '\n')	As above, but uses string s as a prompt.
void	L.print(*ostream& O*, *char space* = ' ')	prints the contents of list L to the output stream O using the overload *Print* function (cf. section 1.5) to print each element. The elements are separated by the character *space*.
void	L.print(*char space* = ' ')	calls L.print(*cout*, *space*) to print L on the standard output stream *cout*.
void	L.print(*string s*, *char space* = ' ')	As above, but uses string s as a header.

3.4 Iterators

Each list L has a special item called the iterator of L. There are operations to read the current value or the contents of this iterator, to move it (setting it to its successor or predecessor) and to test whether the end (head or tail) of the list is reached. If the iterator contains a *list_item* $\neq$ *nil* we call this item the position of the iterator. Iterators are used to implement iteration statements on lists.

void	L.set_iterator(*list_item it*)	assigns item *it* to the iterator. *Precondition*: *it* is in L or *it* = nil.
void	L.init_iterator()	assigns nil to the iterator.
list_item	L.get_iterator()	returns the current value of the iterator.
list_item	L.move_iterator(*int dir*)	moves the iterator to its successor (predecessor) if *dir* = *forward* (*backward*) and to the first (last) item if the iterator is undefined (= nil), returns the value of the iterator.

bool	L.current_element(E& x)	if the iterator is defined ($\neq$ nil) its contents is assigned to x and true is returned else false is returned.
bool	L.next_element(E& x)	calls L.move_iterator(*forward*) and then returns L.current_element(x).
bool	L.prev_element(E& x)	calls L.move_iterator(*backward*) and then returns L.current_element(x).

3.5 Operators

list<E>&	L = $L1$	The assignment operator makes L a copy of list L_1. More precisely if L_1 is the sequence of items $x_1, x_2, \ldots, x_n$ then L is made a sequence of items $y_1, y_2, \ldots, y_n$ with $L[y_i] = L_1[x_i]$ for $1 \leq i \leq n$.
E&	L [*list_item it*]	returns a reference to the contents of *it*.

3.6 Iterations Macros

forall_items(it, L) { "the items of L are successively assigned to it" }

forall(x, L) { "the elements of L are successively assigned to x" }

4. Implementation

The data type list is realized by doubly linked linear lists. All operations take constant time except for the following operations: search and rank take linear time $O(n)$, bucket_sort takes time $O(n + j - i)$ and sort takes time $O(n \cdot c \cdot \log n)$ where c is the time complexity of the compare function. n is always the current length of the list.

4.8 Singly Linked Lists (slist)

1. Definition

An instance L of the parameterized data type *slist<E>* is a sequence of items (*slist_item*). Each item in L contains an element of data type E, called the element type of L. The number of items in L is called the length of L. If L has length zero it is called the empty list. In the sequel $<x>$ is used to denote a list item containing the element x and $L[i]$ is used to denote the contents of list item i in L.

2. Creation

slist<E> L;

creates an instance L of type *slist<E>* and initializes it to the empty list.

slist<E> $L(E\ x)$;

creates an instance L of type *slist<E>* and initializes it to the one-element list $<x>$.

3. Operations

int	L.length()	returns the length of L.
int	L.size()	returns L.length().
bool	L.empty()	returns true if L is empty, false otherwise.
slist_item	L.first()	returns the first item of L.
slist_item	L.last()	returns the last item of L.
slist_item	L.succ(*slist_item it*)	returns the successor item of item *it*, nil if *it* = L.last(). *Precondition*: *it* is an item in L.
slist_item	L.cyclic_succ(*slist_item it*)	returns the cyclic successor of item *it*, i.e., L.first() if *it* = L.last(), L.succ(*it*) otherwise.
E	L.contents(*slist_item it*)	returns the contents $L[it]$ of item *it*. *Precondition*: *it* is an item in L.
E	L.inf(*slist_item it*)	returns L.contents(*it*). *Precondition*: *it* is an item in L.
E	L.head()	returns the first element of L, i.e. the contents of L.first(). *Precondition*: L is not empty.
E	L.tail()	returns the last element of L, i.e. the contents of L.last(). *Precondition*: L is not empty.

slist_item	L.push(E x)	adds a new item <x> at the front of L and returns it.
slist_item	L.append(E x)	appends a new item <x> to L and returns it.
slist_item	L.insert(E x, *slist_item* it)	inserts a new item <x> after item *it* into L and returns it. *Precondition*: *it* is an item in L.
E	L.pop()	deletes the first item from L and returns its contents. *Precondition*: L is not empty.
void	L.conc(*slist*<E>& $L1$)	appends list L_1 to list L and makes L_1 the empty list. *Precondition*: $L \mathrel{!}= L_1$.
E&	L [*slist_item* it]	returns a reference to the contents of *it*.
slist_item	L $+=$ E x	appends a new item <x> to L and returns it.

4.9 Sets (set)

1. Definition

An instance S of the parameterized data type *set<E>* is a collection of elements of the linearly ordered type E, called the element type of S. The size of S is the number of elements in S, a set of size zero is called the empty set.

2. Creation

set<E> S;

creates an instance S of type *set<E>* and initializes it to the empty set.

3. Operations

void	S.insert(E x)	adds x to S.
void	S.del(E x)	deletes x from S.
bool	S.member(E x)	returns true if x in S, false otherwise.
E	S.choose()	returns an element of S. *Precondition*: S is not empty.
bool	S.empty()	returns true if S is empty, false otherwise.
int	S.size()	returns the size of S.
void	S.clear()	makes S the empty set.

Iteration

forall(x, S) { "the elements of S are successively assigned to x" }

4. Implementation

Sets are implemented by randomized search trees [1]. Operations insert, del, member take time $O(\log n)$, empty, size take time $O(1)$, and clear takes time $O(n)$, where n is the current size of the set.

4.10 Integer Sets (int_set)

1. Definition

An instance S of the data type *int_set* is a subset of a fixed interval $[a..b]$ of the integers.

2. Creation

int_set $S(int\ a,\ int\ b)$;

creates an instance S of type *int_set* for elements from $[a..b]$ and initializes it to the empty set.

int_set $S(int\ n)$;

creates an instance S of type *int_set* for elements from $[0..n-1]$ and initializes it to the empty set.

3. Operations

void	S.insert(*int* x)	adds x to S. *Precondition*: $a \le x \le b$.
void	S.del(*int* x)	deletes x from S. *Precondition*: $a \le x \le b$.
int	S.member(*int* x)	returns true if x in S, false otherwise. *Precondition*: $a \le x \le b$.
void	S.clear()	makes S the empty set.
int_set&	$S\ =\ S1$	assignment.
int_set	$S\ \|\ S1$	returns the union of S and $S1$
int_set	$S\ \&\ S1$	returns the intersection of S and $S1$
int_set	$\tilde{}\,S$	returns the complement of S.

4. Implementation

Integer sets are implemented by bit vectors. Operations insert, delete, member, empty, and size take constant time. clear, intersection, union and complement take time $O(b-a+1)$.

4.11 Partitions (partition)

1. Definition

An instance P of the data type *partition* consists of a finite set of items (*partition_item*) and a partition of this set into blocks.

2. Creation

partition P;

creates an instance P of type *partition* and initializes it to the empty partition.

3. Operations

partition_item	P.make_block()	returns a new *partition_item* *it* and adds the block $\{it\}$ to partition P.
partition_item	P.find(*partition_item* p)	returns a canonical item of the block that contains item p, i.e., if P.same_block(p, q) then P.find(p) = P.find(q). *Precondition*: p is an item in P.
bool	P.same_block(*partition_item* p, *partition_item* q)	returns true if p and q belong to the same block of partition P. *Precondition*: p and q are items in P.
void	P.union_blocks(*partition_item* p, *partition_item* q)	unites the blocks of partition P containing items p and q. *Precondition*: p and q are items in P.

4. Implementation

Partitions are implemented by the union find algorithm with weighted union and path compression (cf. [48]). Any sequence of n make_block and $m \geq n$ other operations takes time $O(m\alpha(m, n))$.

5. Example

Spanning Tree Algorithms (cf. section 7.20).

4.12 Dynamic Collections of Trees (tree_collection)

1. Definition

An instance D of the parameterized data type *tree_collection<I>* is a collection of vertex disjoint rooted trees, each of whose vertices has a double-valued cost and contains an information of type I, called the information type of D.

2. Creation

tree_collection<I> D;

creates an instance D of type *tree_collection<I>*, initialized with the empty collection.

3. Operations

d_vertex	D.maketree(I x)	adds a new tree to D containing a single vertex v with cost zero and information x, and returns v.
I	D.inf(*d_vertex* v)	returns the information of vertex v.
d_vertex	D.findroot(*d_vertex* v)	returns the root of the tree containing v.
d_vertex	D.findcost(*d_vertex* v, *double&* x)	sets x to the minimum cost of a vertex on the tree path from v to findroot(v) and returns the last vertex (closest to the root) on this path of cost x.
void	D.addcost(*d_vertex* v, *double* x)	adds double number x to the cost of every vertex on the tree path from v to findroot(v).
void	D.link(*d_vertex* v, *d_vertex* x)	combines the trees containing vertices v and w by adding the edge (v,w). (We regard tree edges as directed from child to parent.) *Precondition*: v and w are in different trees and v is a root.
void	D.cut(*d_vertex* v)	divides the tree containing vertex v into two trees by deleting the edge out of v. *Precondition*: v is not a tree root.

4. Implementation

Dynamic collections of trees are implemented by partitioning the trees into vertex disjoint paths and representing each path by a self-adjusting binary tree (see [48]). All operations take amortized time $O(\log n)$ where n is the number of maketree operations.

Chapter 5

Dictionaries

5.1 Dictionaries (dictionary)

1. Definition

An instance D of the parameterized data type *dictionary*$<K, I>$ is a collection of items (*dic_item*). Every item in D contains a key from the linearly ordered data type K, called the key type of D, and an information from the data type I, called the information type of D. The number of items in D is called the size of D. A dictionary of size zero is called the empty dictionary. We use $<k, i>$ to denote an item with key k and information i (i is said to be the information associated with key k). For each $k \in K$ there is at most one $i \in I$ with $<k, i> \in D$.

2. Creation

dictionary$<K, I>$ D;

creates an instance D of type *dictionary*$<K, I>$ and initializes it with the empty dictionary.

3. Operations

K	D.key(*dic_item it*)	returns the key of item *it*. *Precondition*: *it* is an item in D.
I	D.inf(*dic_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in D.
dic_item	D.insert(K k, I i)	associates the information i with the key k. If there is an item $<k, j>$ in D then j is replaced by i, else a new item $<k, i>$ is added to D. In both cases the item is returned.
dic_item	D.lookup(K k)	returns the item with key k (nil if no such item exists in D).
I	D.access(K k)	returns the information associated with key k. *Precondition*: there is an item with key k in D.

void	*D*.del(*K* *k*)	deletes the item with key *k* from *D* (null operation, if no such item exists).
void	*D*.del_item(*dic_item* *it*)	removes item *it* from *D*. *Precondition*: *it* is an item in *D*.
void	*D*.change_inf(*dic_item* *it*, *I* *i*)	makes *i* the information of item *it*. *Precondition*: *it* is an item in *D*.
void	*D*.clear()	makes *D* the empty dictionary.
int	*D*.size()	returns the size of *D*.
bool	*D*.empty()	returns true if *D* is empty, false otherwise.

4. Implementation

Dictionaries are implemented by randomized search trees [1]. Operations insert, lookup, del_item, del take time $O(\log n)$, key, inf, empty, size, change_inf take time $O(1)$, and clear takes time $O(n)$. Here n is the current size of the dictionary. The space requirement is $O(n)$.

5. Example

We count the number of occurrences of each string in a sequence of strings.

```
#include <LEDA/dictionary.h>

main()

{ dictionary<string,int> D;

  string s;

  dic_item it;

  while (cin >> s)

  { it = D.lookup(s);

    if (it==nil) D.insert(s,1);

    else D.change_inf(it,D.inf(it)+1);

  }

  forall_items(it,D) cout << D.key(it) << " : " <<  D.inf(it) << endl;

}
```

5.2 Dictionaries with Implementation Parameter (_dictionary)

1. Definition

An instance of type *_dictionary*<*K*, *I*, *impl*> is a dictionary implemented by data type *impl*. *impl* must be one of the dictionary implementations listed in section 13.1.1 or a user defined data structure fulfilling the specification given in section 13.2.1. Note that depending on the actual implementation *impl* the key type *K* must either be linearly ordered or hashed.

Example
Using a dictionary implemented by skiplists to count the number of occurrences of the elements in a sequence of strings.

```
#include <LEDA/_dictionary.h>
#include <LEDA/impl/skiplist.h>
main()
{
  _dictionary<string,int,skiplist> D;
  string s;
  dic_item it;
  while (cin >> s)
  { it = D.lookup(s);
    if (it==nil) D.insert(s,1);
    else D.change_inf(it,D.inf(it)+1);
  }
  forall_items(it,D) cout << D.key(it) << " : " << D.inf(it) << endl;
}
```

5.3 Sorted Sequences (sortseq)

1. Definition

An instance S of the parameterized data type *sortseq*<K, I> is a sequence of items (*seq_item*). Every item contains a key from the linearly ordered data type K, called the key type of S, and an information from data type I, called the information type of S. The number of items in S is called the size of S. A sorted sequence of size zero is called empty. We use <k, i> to denote a *seq_item* with key k and information i (called the information associated with key k). For each $k \in K$ there is at most one item <k, i> $\in S$.

The linear order on K may be time-dependent, e.g., in an algorithm that sweeps an arrangement of lines by a vertical sweep line we may want to order the lines by the y-coordinates of their intersections with the sweep line. However, whenever an operation (except for reverse_items) is applied to a sorted sequence S, the keys of S must form an increasing sequence according to the currently valid linear order on K. For operation reverse_items this must hold after the execution of the operation.

2. Creation

sortseq<K, I> S;

creates an instance S of type *sortseq*<K, I> and initializes it to the empty sorted sequence.

3. Operations

K	S.key(*seq_item it*)	returns the key of item *it*. *Precondition*: *it* is an item in S.
I	S.inf(*seq_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in S.
seq_item	S.lookup(K k)	returns the item with key k (nil if no such item exists in S).
seq_item	S.insert(K k, I i)	associates information i with key k: If there is an item <k, j> in S then j is replaced by i, else a new item <k, i> is added to S. In both cases the item is returned.
seq_item	S.insert_at(*seq_item it*, K k, I i)	Like insert(k, i), the item *it* gives the position of the item <k, i> in the sequence. *Precondition*: *it* is an item in S with either key(*it*) is maximal with key(*it*) $\leq k$ or key(*it*) is minimal with key(*it*) $\geq k$.
seq_item	S.locate_succ(K k)	returns the item <k', i> in S such that k' is minimal with $k' \geq k$ (nil if no such item exists).

seq_item	S.locate_pred(K k)	returns the item <k', i> in S such that k' is maximal with $k' \leq k$ (nil if no such item exists).
seq_item	S.locate(K k)	returns S.locate_succ(K k).
seq_item	S.succ(*seq_item it*)	returns the successor item of *it*, i.e., the item <k, i> in S such that k is minimal with $k > key(it)$ (nil if no such item exists). *Precondition*: *it* is an item in S.
seq_item	S.pred(*seq_item it*)	returns the predecessor item of *it*, i.e., the item <k, i> in S such that k is maximal with $k < key(it)$ (nil if no such item exists). *Precondition*: *it* is an item in S.
seq_item	S.min()	returns the item with minimal key (nil if S is empty).
seq_item	S.max()	returns the item with maximal key (nil if S is empty).
void	S.del(K k)	removes the item with key k from S(null operation if no such item exists).
void	S.del_item(*seq_item it*)	removes the item *it* from S. *Precondition*: *it* is an item in S.
void	S.change_inf(*seq_item it*, I i)	makes i the information of item *it*. *Precondition*: *it* is an item in S.
void	S.reverse_items(*seq_item a*, *seq_item b*)	the subsequence of S from a to b is reversed. *Precondition*: a appears before b in S.
void	S.split(*seq_item it*, *sortseq<K, I>& S1*, *sortseq<K, I>& S2*)	splits S at item *it* into sequences S_1 and S_2 and makes S empty. More precisely, if $S = x_1, \ldots, x_{k-1}, it, x_{k+1}, \ldots, x_n$ then $S_1 = x_1, \ldots, x_{k-1}, it$ and $S_2 = x_{k+1}, \ldots, x_n$. *Precondition*: *it* is an item in S.
sortseq<K, I>&	S.conc(*sortseq<K, I>& S1*)	appends S_1 to S, makes S_1 empty and returns S.*Precondition*: S.key(S.max()) $<$ S_1.key(S_1.min()).
void	S.clear()	makes S the empty sorted sequence.
int	S.size()	returns the size of S.
bool	S.empty()	returns true if S is empty, false otherwise.

4. Implementation

Sorted sequences are implemented by (2,4)-trees. Operations lookup, locate, insert, del, split, conc take time $O(\log n)$, operations succ, pred, max, min, key, inf, insert_at and del_item take time $O(1)$. Clear takes time $O(n)$ and reverse_items $O(\ell)$, where ℓ is the length of the reversed subsequence. The space requirement is $O(n)$. Here n is the current size of the sequence.

5. Example

We use a sorted sequence to list all elements in a sequence of strings lying lexicographically between two given search strings. We first read a sequence of strings terminated by "stop" and then a pair of search strings. We output all strings that lie lexicographically between the two search strings (inclusive).

```
#include <LEDA/sortseq.h>

main()

{

 sortseq<string,int> S;

 string s1,s2;

 while ( cin >> s1 &&  s1 != "stop" )  S.insert(s1,0);

 while ( cin >> s1 >> s2 )

 { seq_item start = S.locate_succ(s1);

   seq_item stop  = S.locate_pred(s2);

   if (S.key(start) <= S.key(stop))

   { for (seq_item it = start; it != stop; it = S.succ(it))

      cout << S.key(it) << endl;

   }

  }

}
```

5.4 Sorted Sequences with Implementation Parameter (_sortseq)

1. Definition

An instance of type *_sortseq*<K, I, *impl*> is a sorted sequence implemented by data type *impl*. *impl* must be one of the sorted sequence implementations listed in section 13.1.1 or a user defined data structure fulfilling the specification given in section 13.2.3. Note that the key type K must be linearly ordered.

Example
Using a sorted sequence implemented by skiplists to list all elements in a sequence of strings lying lexicographically between two given search strings.

```
#include <LEDA/_sortseq.h>

#include <LEDA/impl/skiplist.h>

main()

{

 _sortseq<string,int,skiplist> S;

 string s1,s2;

 while ( cin >> s1 &&  s1 != "stop" )  S.insert(s1,0);

 while ( cin >> s1 >> s2 )

 { seq_item start = S.locate(s1);

   seq_item stop  = S.locate(s2);

   for (seq_item it = start; it != stop; it = S.succ(it))

      cout << S.key(it) << endl;

  }

}
```

5.5 Dictionary Arrays (d_array)

1. Definition

An instance A of the parameterized data type *d_array*<I, E> (dictionary array) is an injective mapping from the linearly ordered data type I, called the index type of A, to the set of variables of data type E, called the element type of A. We use $A(i)$ to denote the variable with index i.

2. Creation

d_array<I, E> $A(E\ x)$;

creates an injective function a from I to the set of unused variables of type E, assigns x to all variables in the range of a and initializes A with a.

3. Operations

E&	$A\ [I\ i]$	returns the variable $A(i)$.
bool	A.defined($I\ i$)	returns true if $i \in dom(A)$, false otherwise; here $dom(A)$ is the set of all $i \in I$ for which $A[i]$ has already been executed.
void	A.undefine($I\ i$)	removes i from $dom(A)$.

Iteration

forall_defined(i, A) { "the elements from $dom(A)$ are successively assigned to i" }

4. Implementation

Dictionary arrays are implemented by randomized search trees [1]. Access operations $A[i]$ take time $O(\log dom(A))$. The space requirement is $O(dom(A))$.

5. Example

Program 1: We use a dictionary array to count the number of occurrences of the elements in a sequence of strings.

```
#include <LEDA/d_array.h>

main()

{

  d_array<string,int> N(0);

  string s;

  while (cin >> s) N[s]++;

  forall_defined(s,N) cout << s << "  " << N[s] << endl;
}
```

Program 2: We use a *d_array<string, string>* to realize an english/german dictionary.

```
#include <LEDA/d_array.h>

main()

{

  d_array<string,string> dic;

  dic["hello"] = "hallo";

  dic["world"] = "Welt";

  dic["book"]  = "Buch";

  dic["key"]   = "Schluessel";

  string s;

  forall_defined(s,dic) cout << s << "  " << dic[s] << endl;

}
```

5.6 Dictionary Arrays with Implementation Parameter (_d_array)

1. Definition

An instance of type *_d_array<I, E, impl>* is a dictionary array implemented by data type *impl*. *impl* must be one of the dictionary implementations listed in section 13.1.1 or a user defined data structure fulfilling the specification given in section 13.2.1. Note that depending on the actual implementation *impl* the index type *I* must either be linearly ordered or hashed.

Example
Using a dictionary array implemented by hashing with chaining (*ch_hash*) to count the number of occurences of the elements in a sequence of strings.

```
#include <LEDA/_d_array.h>

#include <LEDA/impl/ch_hash.h>

//we first have to define a hash function for strings

int Hash(const string& x) { return (x.length() > 0) ? x[0] : 0; }

main()

{

  _d_array<string,int,ch_hash> N(0);

  string s;

  while (cin >> s) N[s]++;

  forall_defined(s,N) cout << s << "  " << N[s] << endl;

}
```

5.7 Hashing Arrays (h_array)

1. Definition

An instance A of the parameterized data type *h_array<I, E>* (hashing array) is an injective mapping from a hashed data type I (cf. section 1.7), called the index type of A, to the set of variables of arbitrary type E, called the element type of A. We use $A(i)$ to denote the variable indexed by i.

2. Creation

h_array<I, E> $A(E\ x)$;

creates an injective function a from I to the set of unused variables of type E, assigns x to all variables in the range of a and initializes A with a.

3. Operations

E&	$A\ [I\ i]$	returns the variable $A(i)$
bool	A.defined($I\ i$)	returns true if $i \in dom(A)$, false otherwise; here $dom(A)$ is the set of all $i \in I$ for which $A[i]$ has already been executed.

forall_defined(i, A) { "the elements from $dom(A)$ are successively assigned to i" }

4. Implementation

Hashing arrays are implemented by hashing with chaining. Access operations take expected time $O(1)$. In many cases, hashing arrays are more efficient than dictionary arrays (cf. 5.5).

5.8 Maps (map)

1. Definition

An instance M of the parameterized data type *map*<I,E> is an injective mapping from the data type I, called the index type of M, to the set of variables of data type E, called the element type of M. I must be a pointer, item, or handle type or the type int. We use $M(i)$ to denote the variable indexed by i.

2. Creation

map<I,E> M;

creates an injective function a from I to the set of unused variables of type E, initializes all variables in the range of a using the default constructor of type E and assigns a to M.

map<I,E> $M(E\ x)$;

creates an injective function a from I to the set of unused variables of type E, assigns x to all variables in the range of a and initializes M with a.

3. Operations

E&	$M\ [I\ i]$	returns the variable $M(i)$.
bool	M.defined($I\ i$)	returns true if $i \in dom(M)$, false otherwise; here $dom(M)$ is the set of all $i \in I$ for which $M[i]$ has already been executed.

4. Implementation

Maps are implemented by hashing with chaining and table doubling. Access operations $M[i]$ take expected time $O(1)$.

5.9 Persistent Dictionaries (p_dictionary)

1. Definition

An instance D of the parameterized data type *p_dictionary*<K, I> is a set of items (type *p_dic_item*). Every item in D contains a key from the linearly ordered data type K, called the key type of D, and an information from data type I, called the information type of D. The number of items in D is called the size of D. A dictionary of size zero is called empty. We use $<k, i>$ to denote an item with key k and information i (i is said to be the information associated with key k). For each $k \in K$ there is at most one item $<k, i> \in D$.

The difference between dictionaries (cf. section 5.1) and persistent dictionaries lies in the fact that update operations performed on a persistent dictionary D do not change D but create and return a new dictionary D'. For example, D.del(k) returns the dictionary D' containing all items it of D with key(it) $\neq k$. Also, an assignment $D1 = D2$ does not assign a copy of $D2$ (with new items) to $D1$ but the value of $D2$ itself.

2. Creation

p_dictionary<K, I> D;

creates an instance D of type *p_dictionary*<K, I> and initializes D to an empty persistent dictionary.

3. Operations

K	D.key(*p_dic_item it*)	returns the key of item it. *Precondition*: $it \in D$.
I	D.inf(*p_dic_item it*)	returns the information of item it. *Precondition*: $it \in D$.
p_dic_item	D.lookup(K k)	returns the item with key k (nil if no such item exists in D).
p_dictionary<K, I>	D.del(K k)	returns $\{x \in D \mid key(x) \neq k\}$.
p_dictionary<K, I>	D.del_item(*p_dic_item it*)	returns $\{x \in D \mid x \neq it\}$.
p_dictionary<K, I>	D.insert(K k, I i)	returns D.del(k) $\cup$ $\{<k, i>\}$.
p_dictionary<K, I>	D.change_inf(*p_dic_item it*, I i)	returns D.del_item(it) $\cup$ $\{<k, i>\}$, where $k = key(it)$. *Precondition*: $it \in D$.
int	D.size()	returns the size of D.
bool	D.empty()	returns true if D is empty, false otherwise.

4. Implementation

Persistent dictionaries are implemented by leaf oriented persistent red black trees. Operations insert, lookup, del_item, del take time $O(\log^2 n)$, key, inf, empty, size, change_inf and clear take time $O(1)$. The space requirement is $O(1)$ for each update operation.

Chapter 6

Priority Queues

6.1 Priority Queues (p_queue)

1. Definition

An instance Q of the parameterized data type *p_queue*<P, I> is a collection of items (type *pq_item*). Every item contains a priority from a linearly ordered type P and an information from an arbitrary type I. P is called the priority type of Q and I is called the information type of Q. The number of items in Q is called the size of Q. If Q has size zero it is called the empty priority queue. We use <p, i> to denote a *pq_item* with priority p and information i.

2. Creation

p_queue<P, I> Q;

creates an instance Q of type *p_queue*<P, I> and initializes it with the empty priority queue.

3. Operations

P	Q.prio(*pq_item it*)	returns the priority of item *it*. *Precondition*: *it* is an item in Q.
I	Q.inf(*pq_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in Q.
pq_item	Q.insert(P x, I i)	adds a new item <x, i> to Q and returns it.
pq_item	Q.find_min()	returns an item with minimal information (nil if Q is empty).
P	Q.del_min()	removes the item *it* = Q.find_min() from Q and returns the priority of it. *Precondition*: Q is not empty.
void	Q.del_item(*pq_item it*)	removes the item *it* from Q. *Precondition*: *it* is an item in Q.

void	Q.change_inf(*pq_item it*, $I\ i$)	makes i the new information of item *it*. *Precondition*: *it* is an item in Q.
void	Q.decrease_p(*pq_item it*, $P\ x$)	makes x the new priority of item *it*. *Precondition*: *it* is an item in Q and x is not larger then *prio*(*it*).
int	Q.size()	returns the size of Q.
bool	Q.empty()	returns true, if Q is empty, false otherwise.
void	Q.clear()	makes Q the empty priority queue.

4. Implementation

Priority queues are implemented by Fibonacci heaps [22]. Operations insert, del_item, del_min take time $O(\log n)$, find_min, decrease_p, prio, inf, empty take time $O(1)$ and clear takes time $O(n)$, where n is the size of Q. The space requirement is $O(n)$.

5. Example

Dijkstra's Algorithm (cf. section 12.1)

6.2 Priority Queues with Implementation Parameter (_p_queue)

1. Definition

An instance of type *_p_queue*<P, I, *impl*> is a priority queue implemented by data type *impl*. *impl* must be one of the priority queue implementations listed in section 13.1.2 or a user defined data structure fulfilling the specification given in section 13.2.2. Note that the priority type P must linearly ordered.

6.3 Old-Style Priority Queues (priority_queue)

1. Definition

An instance Q of the parameterized data type *priority_queue*<K, I> is a collection of items (type *pq_item*). Every item contains a key from type K and an information from the linearly ordered type I. K is called the key type of Q and I is called the information type of Q. The number of items in Q is called the size of Q. If Q has size zero it is called the empty priority queue. We use <k, i> to denote a *pq_item* with key k and information i.

The type *priority_queue*<K, I> is identical to the type *p_queue* except that the meanings of K and I are interchanged. We now believe that the semantics of *p_queue* is the more natural one and keep *priority_queue*<K, I> only for historical reasons. We recommend to use *p_queue* instead.

2. Creation

priority_queue<K, I> Q;

creates an instance Q of type *priority_queue*<K, I> and initializes it with the empty priority queue.

3. Operations

K	Q.key(*pq_item it*)	returns the key of item *it*. *Precondition*: *it* is an item in Q.
I	Q.inf(*pq_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in Q.
pq_item	Q.insert(K k, I i)	adds a new item <k, i> to Q and returns it.
pq_item	Q.find_min()	returns an item with minimal information (nil if Q is empty).
K	Q.del_min()	removes the item $it = Q$.find_min() from $Q$and returns the key of *it*. *Precondition*: Q is not empty.
void	Q.del_item(*pq_item it*)	removes the item *it* from Q. *Precondition*: *it* is an item in Q.
void	Q.change_key(*pq_item it*, K k)	makes k the new key of item *it*. *Precondition*: *it* is an item in Q.
void	Q.decrease_inf(*pq_item it*, I i)	makes i the new information of item *it*. *Precondition*: *it* is an item in Q and i is not larger then $inf(it)$.
int	Q.size()	returns the size of Q.
bool	Q.empty()	returns true, if Q is empty, false otherwise
void	Q.clear()	makes Q the empty priority queue.

4. Implementation

Priority queues are implemented by Fibonacci heaps [22]. Operations insert, del_item, del_min take time $O(\log n)$, find_min, decrease_inf, key, inf, empty take time $O(1)$ and clear takes time $O(n)$, where n is the size of Q. The space requirement is $O(n)$.

5. Example

Dijkstra's Algorithm (cf. section 12.1)

6.4 Bounded Priority Queues (b_priority_queue)

1. Definition

An instance Q of the parameterized data type *b_priority_queue<K>* is a priority_queue (cf. section 6.1) whose information type is a fixed interval $[a..b]$ of integers.

2. Creation

b_priority_queue<K> $Q(int\ a,\ int\ b)$;

creates an instance Q of type *b_priority_queue<K>* with information type $[a..b]$ and initializes it with the empty priority queue.

3. Operations

See section 6.1.

4. Implementation

Bounded priority queues are implemented by arrays of linear lists. Operations insert, find_min, del_item, decrease_inf, key, inf, and empty take time $O(1)$, del_min (= del_item for the minimal element) takes time $O(d)$, where d is the distance of the minimal element to the next bigger element in the queue (= $O(b-a)$ in the worst case). clear takes time $O(b-a+n)$ and the space requirement is $O(b-a+n)$, where n is the current size of the queue.

Chapter 7

Graphs and Related Data Types

7.1 Graphs (graph)

1. Definition

An instance G of the data type *graph* consists of a list V of nodes and a list E of edges (*node* and *edge* are item types). A pair of nodes $(v, w) \in V \times V$ is associated with every edge $e \in E$; v is called the *source* of e and w is called the *target* of e. Two lists of edges are associated with every node v: the list $out\_edges(v) = \{e \in E \mid source(e) = v\}$ of edges starting in v, and the list $in\_edges(v) = \{e \in E \mid target(e) = v\}$ of edges ending in v. Distinct graphs have disjoint node and edge sets. A graph with empty node list is called *empty*.

A graph is either *directed* or *undirected*; the main difference between directed and undirected edges is the definition of *adjacent*. Undirected graphs are the subject of section 7.3. In a directed graph an edge is adjacent to its source and in an undirected graph it is adjacent to its source and target. In a directed graph a node w is adjacent to a node v if there is an edge $(v, w) \in E$; in an undirected graph w is adjacent to v if there is an edge (v, w) or (w, v) in the graph. The *adjacency list* of a node v is the list of edges adjacent to v; more precisely, for directed graphs the adjacency list of v is equal to $out\_edges(v)$ and for undirected graphs it is the concatenation of $out\_edges(v)$ and $in\_edges(v)$.

The value of a variable of type node is either the node of some graph, or the special value nil (which is distinct from all nodes), or is undefined (before the first assignment to the variable). A corresponding statement is true for the variables of type edge.

2. Creation

graph G;

creates an object G of type *graph* and initializes it to the empty directed graph.

3. Operations

a) Access operations

int	*G*.outdeg(*node v*)	returns the outdegree of node v, i.e., the number of edges starting at v ($\|out\_edges(v)\|$).
int	*G*.indeg(*node v*)	returns the indegree of node v, i.e., the number of edges ending at v ($\|in\_edges(v)\|$).
int	*G*.degree(*node v*)	returns the degree of node v, i.e., the number of edges starting or ending at v.
node	*G*.source(*edge e*)	returns the source node of edge e.
node	*G*.target(*edge e*)	returns the target node of edge e.
node	*G*.opposite(*node v*, *edge e*)	returns a node of edge e different from v (returns v if e has source and target equal to v) .
int	*G*.number_of_nodes()	returns the number of nodes in G.
int	*G*.number_of_edges()	returns the number of edges in G.
list<edge>	*G*.all_edges()	returns the list E of all edges of G.
list<node>	*G*.all_nodes()	returns the list V of all nodes of G.
list<edge>	*G*.adj_edges(*node v*)	returns the list of all edges adjacent to v.
list<edge>	*G*.in_edges(*node v*)	returns the list of all edges ending at v.
list<node>	*G*.adj_nodes(*node v*)	returns the list of all nodes adjacent to v.
node	*G*.first_node()	returns the first node in V.
node	*G*.last_node()	returns the last node in V.
node	*G*.choose_node()	returns a node of G (nil if G is empty).
node	*G*.succ_node(*node v*)	returns the successor of node v in V (nil if it does not exist).
node	*G*.pred_node(*node v*)	returns the predecessor of node v in V (nil if it does not exist).
edge	*G*.first_edge()	returns the first edge in E.
edge	*G*.last_edge()	returns the last edge in E.
edge	*G*.choose_edge()	returns an edge of G (nil if G has no edges).
edge	*G*.succ_edge(*edge e*)	returns the successor of edge e in E (nil if it does not exist).
edge	*G*.pred_edge(*edge e*)	returns the predecessor of edge e in E (nil if it does not exist).

edge	G.first_adj_edge(*node v*)	returns the first edge in the adjacency list of v.
edge	G.last_adj_edge(*node v*)	returns the last edge in the adjacency list of v.
edge	G.adj_succ(*edge e*)	returns the successor of edge e in the adjacency list of node *source*(e) (nil if it does not exist).
edge	G.adj_pred(*edge e*)	returns the predecessor of edge e in the adjacency list of node *source*(e) (nil if it does not exist).
edge	G.cyclic_adj_succ(*edge e*)	returns the cyclic successor of edge e in the adjacency list of node *source*(e).
edge	G.cyclic_adj_pred(*edge e*)	returns the cyclic predecessor of edge e in the adjacency list of node *source*(e).
edge	G.first_in_edge(*node v*)	returns the first edge of *in_edges*(v).
edge	G.last_in_edge(*node v*)	returns the last edge of *in_edges*(v).
edge	G.in_succ(*edge e*)	returns the successor of edge e in *in_edges*(*target*(e)) (nil if it does not exist).
edge	G.in_pred(*edge e*)	returns the predecessor of edge e in *in_edges*(*target*(e)) (nil if it does not exist).
edge	G.cyclic_in_succ(*edge e*)	returns the cyclic successor of edge e in *in_edges*(*target*(e)) (nil if it does not exist).
edge	G.cyclic_in_pred(*edge e*)	returns the cyclic predecessor of edge e in *in_edges*(*target*(e)) (nil if it does not exist).

b) Update operations

node	G.new_node()	adds a new node to G and returns it.
edge	G.new_edge(*node v*, *node w*)	adds a new edge (v, w) to G by appending it to *out_edges*(v) and to *in_edges*(w), and returns it.
edge	G.new_edge(*edge e*, *node w*, *int dir* = *after*)	adds a new edge (*source*(e), w) to G by inserting it before (*dir* = *before*) or after (*dir* = *after*) edge e into *out_edges*(*source*(e)) and appending it to *in_edges*(w), and returns it. Here *before* and *after* are predefined integer constants.
edge	G.new_edge(*edge e*1, *edge e*2, *int d*1 = *after*, *int d*2 = *after*)	

		adds a new edge (*source*(*e*1), *target*(*e*2)) to *G* by inserting it before (if *d*1 = *before*) or after (if *d*1 = *after*) edge *e*1 into *out_edges*(*source*(*e*1)) and before (if *d*2 = *before*) or after (if *d*2 = *after*) edge *e*2 into *in_edges*(*target*(*e*2)), and returns it.
void	*G*.hide_edge(*edge e*)	removes edge *e* from *out_edges*(*source*(*e*)) and from *in_edges*(*target*(*e*)), but leaves it in the list of all edges *E*.
void	*G*.restore_edge(*edge e*)	re-inserts *e* into *out_edges*(*source*(*e*)) and into *in_edges*(*target*(*e*)). *Precondition*: *e* must have been removed by a call of hide_edge before.
void	*G*.del_node(*node v*)	deletes node *v* and all edges starting or ending at *v* from *G*.
void	*G*.del_edge(*edge e*)	deletes the edge *e* from *G*.
void	*G*.del_all_nodes()	deletes all nodes from *G*.
void	*G*.del_all_edges()	deletes all edges from *G*.
edge	*G*.rev_edge(*edge e*)	reverses the edge $e = (v, w)$ by removing it from *G* and inserting the edge (w, v) into *G*; returns the reversed edge.
graph&	*G*.rev()	all edges in *G* are reversed.
void	*G*.sort_nodes(*int* (**cmp*)(*const node&*, *const node&*))	
		the nodes of *G* are sorted according to the ordering defined by the comparing function *cmp*. Subsequent executions of forall_nodes step through the nodes in this order. (cf. TOPSORT1 in section 12.1).
void	*G*.sort_edges(*int* (**cmp*)(*const edge&*, *const edge&*))	
		the edges of *G* and all out_edges lists (but not the in_edges lists) are sorted according to the ordering defined by the comparing function *cmp*. Subsequent executions of forall_edges step through the edges in this order. (cf. TOPSORT1 in section 12.1).
void	*G*.sort_nodes(*node_array*<*T*> *A*)	
		the nodes of *G* are sorted according to the entries of node_array *A* (cf. section 7.7) *Precondition*: *T* must be linearly ordered.

void	*G*.sort_edges(*edge_array<T> A*)	the edges of *G* are sorted according to the entries of edge_array *A* (cf. section 7.8) *Precondition*: *T* must be linearly ordered.
list<edge>	*G*.insert_reverse_edges()	for every edge (v, w) in *G* the reverse edge (w, v) is inserted into *G*. Returns the list of all inserted edges.
void	*G*.make_undirected()	make *G* undirected.
void	*G*.make_directed()	make *G* directed.
bool	*G*.is_directed()	returns true if *G* directed.
bool	*G*.is_undirected()	returns true if *G* undirected.
void	*G*.clear()	makes *G* the empty graph.

c) Iterators

With the adjacency list of every node v a list iterator, called the adjacency iterator of v, is associated (cf. section 4.7). There are operations to initialize, move, and read these iterators.

void	*G*.init_adj_iterator(*node v*)	sets the adjacency iterator of v to undefined.
bool	*G*.next_adj_edge(*edge& e*, *node v*)	moves the adjacency iterator of v forward by one edge (to the first item of the adjacency list of v if it was undefined) and returns *G*.current_adj_edge(e, v).
bool	*G*.current_adj_edge(*edge& e*, *node v*)	if the adjacency iterator of v is defined then the corresponding edge is assigned to e and true is returned, otherwise, false is returned.
bool	*G*.next_adj_node(*node& w*, *node v*)	if *G*.next_adj_edge(e, v) = true then *target*(e) is assigned to w and true is returned else false is returned.
bool	*G*.current_adj_node(*node& w*, *node v*)	if *G*.current_adj_edge(e, v) = true then *target*(e) is assigned to w and true is returned, else false is returned.
void	*G*.reset()	sets all iterators in *G* to undefined.

d) I/O Operations

void	G.write(*ostream*& O = *cout*)	writes G to the output stream O.
void	G.write(*string* s)	writes G to the file with name s.
int	G.read(*istream*& I = *cin*)	reads a graph from the input stream I and assigns it to G.
int	G.read(*string* s)	reads a graph from the file with name s and assigns it to G.
void	G.print_node(*node* v, *ostream*& O = *cout*)	prints node v on the output stream O.
void	G.print_edge(*edge* e, *ostream*& O = *cout*)	prints edge e on the output stream O. If G is directed e is represented by an arrow pointing from source to target. If G is undirected e is printed as an undirected line segment.
void	G.print(*string* s, *ostream*& O = *cout*)	pretty-prints G with header line s on the output stream O.
void	G.print(*ostream*& O)	pretty-prints G on the output stream O.
void	G.print()	pretty-prints G on the standard ouput stream *cout*.

e) Iteration

forall_nodes(v, G)
{ "the nodes of G are successively assigned to v" }

forall_edges(e, G)
{ "the edges of G are successively assigned to e" }

Forall_nodes(v, G)
{ "the nodes of G are successively assigned to v in reverse order" }

Forall_edges(e, G)
{ "the edges of G are successively assigned to e in reverse order" }

forall_out_edges(e, w)
{ "the edges of *out_edges*(w) are successively assigned to e" }

forall_in_edges(e, w)
{ "the edges of *in_edges*(w) are successively assigned to e" }

forall_inout_edges(e, w)
{ "the edges of *out_edges*(w) and *in_edges*(w) are successively assigned to e" }

forall_adj_edges(e, w)
{ "the edges adjacent to node w are successively assigned to e" }

forall_adj_nodes(v, w)
{ "the nodes adjacent to node w are successively assigned to v" }

4. Implementation

Graphs are implemented by doubly linked adjacency lists. Most operations take constant time, except for all_nodes, all_edges, del_all_nodes, del_all_edges, clear, write, and read which take time $O(n+m)$, where n is the current number of nodes and m is the current number of edges. The space requirement is $O(n+m)$.

7.2 Parameterized Graphs (GRAPH)

1. Definition

A parameterized graph G is a graph whose nodes and edges contain additional (user defined) data. Every node contains an element of a data type *vtype*, called the node type of G and every edge contains an element of a data type *etype* called the edge type of G. We use $<v, w, y>$ to denote an edge (v, w) with information y and $<x>$ to denote a node with information x.

All operations defined on instances of the data type *graph* are also defined on instances of any parameterized graph type *GRAPH<vtype, etype>*. For parameterized graphs there are additional operations to access or update the information associated with its nodes and edges. Instances of a parameterized graph type can be used wherever an instance of the data type *graph* can be used, e.g., in assignments and as arguments to functions with formal parameters of type *graph&*. If a function f(*graph&* G) is called with an argument Q of type *GRAPH<vtype, etype>* then inside f only the basic graph structure of Q (the adjacency lists) can be accessed. The node and edge entries are hidden. This allows the design of generic graph algorithms, i.e., algorithms accepting instances of any parametrized graph type as argument.

2. Creation

GRAPH<vtype, etype> G;

creates an instance G of type *GRAPH<vtype, etype>* and initializes it to the empty graph.

3. Operations

vtype	G.inf(*node* v)	returns the information of node v.

etype	*G*.inf(*edge e*)	returns the information of edge *e*.
void	*G*.assign(*node v*, *vtype x*)	makes *x* the information of node *v*.
void	*G*.assign(*edge e*, *etype x*)	makes *x* the information of edge *e*.
node	*G*.new_node(*vtype x*)	adds a new node <*x*> to *G* and returns it.
node	*G*.new_node()	adds a new node <*vdef*> to *G* and returns it. Here, *vdef* is the default value of type *vtype*.
edge	*G*.new_edge(*node v*, *node w*, *etype a*)	adds a new edge <*v*, *w*, *a*> to *G* by appending it to the adjacency list of *v* and the in_edges list of *w* and returns it.
edge	*G*.new_edge(*node v*, *node w*)	adds a new edge <*v*, *w*, *edef*> to *G* by appending it to the adjacency list of *v* and the in_edges list of *w* and returns it. Here, *edef* is the default value of type *etype*.
edge	*G*.new_edge(*edge e*, *node w*, *etype a*)	adds a new edge <*source*(*e*), *w*, *a*> to *G* by appending it to the adjacency list of *source*(*e*) and the in-list of *w* and returns it.
edge	*G*.new_edge(*edge e*, *node w*)	adds a new edge <*source*(*e*), *w*, *edef*> to *G* by appending it to the adjacency list of *source*(*e*) and the in_edges list of *w* and returns it. Here, *edef* is the default value of type *etype*.
edge	*G*.new_edge(*edge e*, *node w*, *etype a*, *int dir*)	adds a new edge <*source*(*e*), *w*, *a*> to *G* by inserting it after (*dir* = *after*) or before (*dir* = *before*) edge *e* into the adjacency list of *source*(*e*) and appending it to the in_edges list of *w*. Returns the new edge.
void	*G*.sort_nodes()	the nodes of *G* are sorted according to their contents. *Precondition*: *vtype* is linearly ordered.
void	*G*.sort_edges()	the edges of *G* are sorted according to their contents. *Precondition*: *etype* is linearly ordered.
void	*G*.write(*string fname*)	writes *G* to the file with name *fname*. The output functions *Print*(*vtype*, *ostream*) and *Print*(*etype*, *ostream*) (cf. section 1.6) must be defined.

int	*G*.read(*string fname*)	reads *G* from the file with name *fname*. The input functions *Read*(*vtype*, *istream*) and *Read*(*etype*, *istream*) (cf. section 1.6) must be defined. Returns error code 1 if file *fname* does not exist 2 if graph is not of type *GRAPH*<*vtype*, *etype*> 3 if file *fname* does not contain a graph 0 otherwise.

Operators

vtype&	*G* [*node v*]	returns a reference to *G*.inf(*v*).
etype&	*G* [*edge e*]	returns a reference to *G*.inf(*e*).

4. Implementation

Parameterized graphs are derived from directed graphs. All additional operations for manipulating the node and edge entries take constant time.

7.3 Undirected Graphs (ugraph)

1. Definition

An instance G of the data type *ugraph* is an undirected graph as defined in section 7.1.

2. Creation

ugraph U;

creates an instance U of type *ugraph* and initializes it to the empty undirected graph.

3. Operations

see section 7.1.

4. Implementation

see section 7.1.

7.4 Parameterized Ugraphs (UGRAPH)

1. Definition

A parameterized undirected graph G is an undirected graph whose nodes and contain additional (user defined) data (cf. 7.2). Every node contains an element of a data type *vtype*, called the node type of G and every edge contains an element of a data type *etype* called the edge type of G.

UGRAPH<vtype, etype> U;

creates an instance U of type *UGRAPH<vtype, etype>* and initializes it to the empty undirected graph.

3. Operations

see section 7.2.

4. Implementation

see section 7.2. .

7.5 Planar Maps (planar_map)

1. Definition

An instance M of the data type *planar_map* is the combinatorial embedding of a planar graph, i.e., M is bidirected (for every edge (v,w) of M the reverse edge (w,v) is also in M) and there is a planar embedding of M such that for every node v the ordering of the edges in the adjacency list of v corresponds to the counter-clockwise ordering of these edges around v in the embedding.

Planar maps make use of the item type *face* in addition to nodes and edges.

2. Creation

planar_map M(*graph* G);

creates an instance M of type *planar_map* and initializes it to the planar map represented by the directed graph G.
Precondition: G represents a bidirected planar map, i.e. for every edge (v,w) in G the reverse edge (w,v) is also in G and there is a planar embedding of G such that for every node v the ordering of the edges in the adjacency list of v corresponds to the counter-clockwise ordering of these edges around v in the embedding.

3. Operations

list<face>	M.all_faces()	returns the list of all faces of M.
list<face>	M.adj_faces(*node* v)	returns the list of all faces of M adjacent to node v in counter-clockwise order.
face	M.adj_face(*edge* e)	returns the face of M to the right of e.
list<node>	M.adj_nodes(*face* f)	returns the list of all nodes of M adjacent to face f in clockwise order.
list<edge>	M.adj_edges(*face*)	returns the list of all edges of M bounding face f in clockwise order.
edge	M.reverse(*edge* e)	returns the reversal of edge e in M.
edge	M.first_face_edge(*face* f)	returns the first edge of face f in M.
edge	M.succ_face_edge(*edge* e)	returns the successor edge of e in face M.adj_face(e) i.e., the next edge in clockwise order.
edge	M.pred_face_edge(*edge* e)	returns the predecessor edge of e in face f, i.e., the next edge in counter-clockwise order.
edge	M.new_edge(*edge* $e1$, *edge* $e2$)	

		inserts the edge $e = (source(e_1), source(e_2))$ and its reversal into M and returns e. *Precondition*: e_1 and e_2 are bounding the same face F. The operation splits F into two new faces.
void	M.del_edge(*edge e*)	deletes the edge e from M. The two faces adjacent to e are united to one face.
edge	M.split_edge(*edge e*)	splits edge $e = (v, w)$ and its reversal $r = (w, v)$ into edges (v, u), (u, w), (w, u), and (u, v). Returns the edge (u, w).
node	M.new_node(*list<edge> el*)	splits the face bounded by the edges in *el* by inserting a new node u and connecting it to all source nodes of edges in *el*. *Precondition*: all edges in *el* bound the same face.
node	M.new_node(*face f*)	splits face f into triangles by inserting a new node u and connecting it to all nodes of f. Returns u.
list<edge>	M.triangulate()	triangulates all faces of M by inserting new edges. The list of inserted edges is returned.
int	M.straight_line_embedding(*node_array<int>& x*, *node_array<int>& y*)	computes a straight line embedding for M with integer coordinates $(x[v], y[v])$ in the range $0 \ldots 2(n-1)$ for every node v of M, and returns the maximal used coordinate.

Iteration

forall_faces(f, M) { "the faces of M are successively assigned to f" }

forall_adj_edges(e, f)
{ "the edges adjacent to face f are successively assigned to e" }

forall_adj_faces(f, v)
{ "the faces adjacent to node v are successively assigned to f" }

4. Implementation

Planar maps are implemented by parameterized directed graphs. All operations take constant time, except for new_edge and del_edge which take time $O(f)$ where f is the number of edges in the created faces and triangulate and straight_line_embedding which take time $O(n)$ where n is the current size (number of edges) of the planar map.

7.6 Parameterized Planar Maps (PLANAR_MAP)

1. Definition

A parameterized planar map M is a planar map whose nodes and faces contain additional (user defined) data. Every node contains an element of a data type *vtype*, called the node type of M and every face contains an element of a data type *ftype* called the face type of M. All operations of the data type *planar_map* are also defined for instances of any parameterized planar_map type. For parameterized planar maps there are additional operations to access or update the node and face entries.

2. Creation

PLANAR_MAP<vtype, ftype> M(*GRAPH<vtype, ftype>* G);

creates an instance M of type *PLANAR_MAP<vtype, ftype>* and initializes it to the planar map represented by the parameterized directed graph G. The node entries of G are copied into the corresponding nodes of M and every face f of M is assigned the information of one of its bounding edges in G.
Precondition: G represents a planar map.

3. Operations

vtype	M.inf(*node* v)	returns the information of node v.
ftype	M.inf(*face* f)	returns the information of face f.
vtype&	M [*node* v]	returns a reference to the information of node v.
ftype&	M [*face* f]	returns a reference to the information of face f.
void	M.assign(*node* v, *vtype* x)	makes x the information of node v.
void	M.assign(*face* f, *ftype* x)	makes x the information of face f.
edge	M.new_edge(*edge* e1, *edge* e2, *ftype* y)	inserts the edge $e = (source(e_1), source(e_2))$ and its reversal edge e' into M. *Precondition*: e_1 and e_2 are bounding the same face F. The operation splits F into two new faces f, adjacent to edge e and f', adjacent to edge e' with inf(f) = inf (F) and inf(f') = y.
edge	M.split_edge(*edge* e, *vtype* x)	splits edge $e = (v, w)$ and its reversal $r = (w, v)$ into edges (v, u), (u, w), (w, u), and (u, v). Assigns information x to the created node u and returns the edge (u, w).

node	*M*.new_node(*list<edge>& el*, *vtype x*)	
		splits the face bounded by the edges in *el* by inserting a new node *u* and connecting it to all source nodes of edges in *el*. Assigns information *x* to *u* and returns *u*. *Precondition*: all edges in *el* bound the same face.
node	*M*.new_node(*face f*, *vtype x*)	
		splits face *f* into triangles by inserting a new node *u* with information *x* and connecting it to all nodes of *f*. Returns *u*.

4. Implementation

Parameterized planar maps are derived from planar maps. All additional operations for manipulating the node and edge contents take constant time.

7.7 Node Arrays (node_array)

1. Definition

An instance A of the parameterized data type *node_array<E>* is a partial mapping from the node set of a graph G to the set of variables of type E, called the element type of the array. The domain I of A is called the index set of A and $A(v)$ is called the element at position v. A is said to be valid for all nodes in I.

2. Creation

node_array<E> A;

creates an instance A of type *node_array<E>* with empty index set.

node_array<E> A(*graph* G);

creates an instance A of type *node_array<E>* and initializes the index set of A to the current node set of graph G.

node_array<E> A(*graph* G, E x);

creates an instance A of type *node_array<E>*, sets the index set of A to the current node set of graph G and initializes $A(v)$ with x for all nodes v of G.

node_array<E> A(*graph* G, *int* n, E x);

creates an instance A of type *node_array<E>* valid for up to n nodes of graph G and initializes $A(v)$ with x for all nodes v of G. *Precondition*: $n \geq |V|$. A is also valid for the next $n - |V|$ nodes added to G.

3. Operations

E&	A [*node* v]	returns the variable $A(v)$. *Precondition*: A must be valid for v.
void	A.init(*graph* G)	sets the index set I of A to the node set of G, i.e., makes A valid for all nodes of G.
void	A.init(*graph* G, E x)	makes A valid for all nodes of G and sets $A(v) = x$ for all nodes v of G.
void	A.init(*graph* G, *int* n, E x)	makes A valid for at most n nodes of G and sets $A(v) = x$ for all nodes v of G. *Precondition*: $n \geq |V|$. A is also valid for the next $n - |V|$ nodes added to G.

4. Implementation

Node arrays for a graph G are implemented by C++ vectors and an internal numbering of the nodes and edges of G. The access operation takes constant time, *init* takes time $O(n)$, where n is the number of nodes in G. The space requirement is $O(n)$.

Remark: A node array is only valid for a bounded number of nodes of G. This number is either the number of nodes of G at the moment of creation of the array or it is explicitely set by the user. Dynamic node arrays can be realized by node maps (cf. section 7.9).

7.8 Edge Arrays (edge_array)

1. Definition

An instance A of the parameterized data type *edge_array<E>* is a partial mapping from the edge set of a graph G to the set of variables of type E, called the element type of the array. The domain I of A is called the index set of A and $A(e)$ is called the element at position e. A is said to be valid for all edges in I.

2. Creation

edge_array<E> A;

creates an instance A of type *edge_array<E>* with empty index set.

edge_array<E> A(*graph* G);

creates an instance A of type *edge_array<E>* and initializes the index set of A to be the current edge set of graph G.

edge_array<E> A(*graph* G, E x);

creates an instance A of type *edge_array<E>*, sets the index set of A to the current edge set of graph G and initializes $A(v)$ with x for all edges v of G.

edge_array<E> A(*graph* G, *int* n, E x);

creates an instance A of type *edge_array<E>* valid for up to n edges of graph G and initializes $A(e)$ with x for all edges e of G.
Precondition: $n \geq |E|$.
A is also valid for the next $n - |E|$ edges added to G.

3. Operations

E&	A [*edge* e]	returns the variable $A(e)$. *Precondition*: A must be valid for e.
void	A.init(*graph* G)	sets the index set I of A to the edge set of G, i.e., makes A valid for all edges of G.
void	A.init(*graph* G, E x)	makes A valid for all edges of G and sets $A(e) = x$ for all edges e of G.
void	A.init(*graph* G, *int* n, E x)	makes A valid for at most n edges of G and sets $A(e) = x$ for all edges e of G. *Precondition*: $n \geq |E|$. A is also valid for the next $n - |E|$ edges added to G.

4. Implementation

Edge arrays for a graph G are implemented by C++ vectors and an internal numbering of the edges and edges of G. The access operation takes constant time, *init* takes time $O(n)$, where n is the number of edges in G. The space requirement is $O(n)$.

Remark: An edge array is only valid for a bounded number of edges of G. This number is either the number of edges of G at the moment of creation of the array or it is explicitely set by the user. Dynamic edge arrays can be realized by edge maps (cf. section 7.10).

7.9 Node Maps (node_map)

1. Definition

An instance of the data type *node_map<E>* is a map for the nodes of a graph G, i.e., equivalent to *map<node, E>* (cf. 5.8). It can be used as a dynamic variant of the data type *node_array* (cf. 7.7).

2. Creation

node_map<E> M;

introduces a variable M of type *node_map<E>* and initializes it to the map with empty domain.

node_map<E> M(*graph* G);

introduces a variable M of type *node_map<E>* and initializes it with a mapping m from the set of all nodes of G into the set of variables of type E. The variables in the range of m are initialized by a call of the default constructor of type E.

node_map<E> M(*graph* G, E x);

introduces a variable M of type *node_map<E>* and initializes it with a mapping m from the set of all nodes of G into the set of variables of type E. The variables in the range of m are initialized with a copy of x.

3. Operations

void	M.init()	makes M a node map with empty domain.
void	M.init(*graph* G)	makes M to a mapping m from the set of all nodes of G into the set of variables of type E. The variables in the range of m are initialized by a call of the default constructor of type E.
void	M.init(*graph* G, E x)	makes M to a mapping m from the set of all nodes of G into the set of variables of type E. The variables in the range of m are initialized with a copy of x.
E&	M [*node* v]	returns the variable $M(v)$.

4. Implementation

Node maps are implemented by an efficient hashing method based on the internal numbering of the nodes. An access operation takes expected time $O(1)$.

7.10 Edge Maps (edge_map)

1. Definition

An instance of the data type *edge_map<E>* is a map for the edges of a graph G, i.e., equivalent to *map<edge, E>* (cf. 5.8). It can be used as a dynamic variant of the data type *edge_array* (cf. 7.8).

2. Creation

edge_map<E> M;

introduces a variable M of type *edge_map<E>* and initializes it to the map with empty domain.

edge_map<E> M(*graph* G);

introduces a variable M of type *edge_map<E>* and initializes it with a mapping m from the set of all edges of G into the set of variables of type E. The variables in the range of m are initialized by a call of the default constructor of type E.

edge_map<E> M(*graph* G, E x);

introduces a variable M of type *edge_map<E>* and initializes it with a mapping m from the set of all edges of G into the set of variables of type E. The variables in the range of m are initialized with a copy of x.

3. Operations

void	M.init()	makes M an edge map with empty domain.
void	M.init(*graph* G)	makes M to a mapping m from the set of all edges of G into the set of variables of type E. The variables in the range of m are initialized by a call of the default constructor of type E.
void	M.init(*graph* G, E x)	makes M to a mapping m from the set of all edges of G into the set of variables of type E. The variables in the range of m are initialized with a copy of x.
E&	M [*edge* e]	returns the variable $M(e)$.

4. Implementation

Edge maps are implemented by an efficient hashing method based on the internal numbering of the edges. An access operation takes expected time $O(1)$.

7.11 Two Dimensional Node Arrays (node_matrix)

1. Definition

An instance M of the parameterized data type *node_matrix*<E> is a partial mapping from the set of node pairs $V \times V$ of a graph to the set of variables of data type E, called the element type of M. The domain I of M is called the index set of M. M is said to be valid for all node pairs in I. A node matrix can also be viewed as a node array with element type *node_array*<E> (*node_array*<*node_array*<E>>).

2. Creation

node_matrix<E> M;

creates an instance M of type *node_matrix*<E> and initializes the index set of M to the empty set.

node_matrix<E> M(*graph* G);

creates an instance M of type *node_matrix*<E> and initializes the index set to be the set of all node pairs of graph G, i.e., M is made valid for all pairs in $V \times V$ where V is the set of nodes currently contained in G.

node_matrix<E> M(*graph* G, E x);

creates an instance M of type *node_matrix*<E> and initializes the index set of M to be the set of all node pairs of graph G, i.e., M is made valid for all pairs in $V \times V$ where V is the set of nodes currently contained in G. In addition, $M(v, w)$ is initialized with x for all nodes $v, w \in V$.

3. Operations

void	M.init(*graph* G)	sets the index set of M to $V \times V$, where V is the set of all nodes of G.
void	M.init(*graph* G, E x)	sets the index set of M to $V \times V$, where V is the set of all nodes of G and initializes $M(v, w)$ to x for all $v, w \in V$.
node_array<E>&	M [*node* v]	returns the node_array $M(v)$.
E&	M (*node* v, *node* w)	returns the variable $M(v, w)$. *Precondition*: M must be valid for v and w.

4. Implementation

Node matrices for a graph G are implemented by vectors of node arrays and an internal numbering of the nodes of G. The access operation takes constant time, the init operation takes time $O(n^2)$, where n is the number of nodes currently contained in G. The space requirement is $O(n^2)$. Note that a node matrix is only valid for the nodes contained in G at the moment of the matrix declaration or initialization (*init*). Access operations for later added nodes are not allowed.

7.12 Sets of Nodes (node_set)

1. Definition

An instance S of the data type *node_set* is a subset of the nodes of a graph G. S is said to be valid for the nodes of G.

2. Creation

node_set S(*graph* G);

creates an instance S of type *node_set* valid for all nodes currently contained in graph G and initializes it to the empty set.

3. Operations

void	S.insert(*node* x)	adds node x to S.
void	S.del(*node* x)	removes node x from S.
bool	S.member(*node* x)	returns true if x in S, false otherwise.
node	S.choose()	returns a node of S.
int	S.size()	returns the size of S.
bool	S.empty()	returns true iff S is the empty set.
void	S.clear()	makes S the empty set.

4. Implementation

A node set S for a graph G is implemented by a combination of a list L of nodes and a node array of list_items associating with each node its position in L. All operations take constant time, except for clear which takes time $O(|S|)$. The space requirement is $O(n)$, where n is the number of nodes of G.

7.13 Sets of Edges (edge_set)

1. Definition

An instance S of the data type *edge_set* is a subset of the edges of a graph G. S is said to be valid for the edges of G.

2. Creation

edge_set $S(graph\ G)$;

creates an instance S of type *edge_set* valid for all edges currently in graph G and initializes it to the empty set.

3. Operations

void	S.insert(*edge* x)	adds edge x to S.
void	S.del(*edge* x)	removes edge x from S.
bool	S.member(*edge* x)	returns true if x in S, false otherwise.
edge	S.choose()	returns an edge of S.
int	S.size()	returns the size of S.
bool	S.empty()	returns true iff S is the empty set.
void	S.clear()	makes S the empty set.

4. Implementation

An edge set S for a graph G is implemented by a combination of a list L of edges and an edge array of list_items associating with each edge its position in L. All operations take constant time, except for clear which takes time $O(|S|)$. The space requirement is $O(n)$, where n is the number of edges of G.

7.14 Lists of Nodes (node_list)

1. Definition

An instance of the data type *node_list* is a doubly linked list of nodes. It is implemented more efficiently than the general list type *list<node>* (4.7). However, it can only be used with the restriction that every node is contained in at most one *node_list*.

2. Creation

node_list L;

introduces a variable L of type *node_list* and initializes it with the empty list.

3. Operations

void	L.append(*node* v)	appends v to list L.
void	L.push(*node* v)	adds v at the front of L.
void	L.insert(*node* v, *node* w)	inserts v after w into L. *Precondition*: $w \in L$.
node	L.pop()	deletes the first node from L and returns it. *Precondition*: L is not empty.
void	L.del(*node* v)	deletes v from L. *Precondition*: $v \in L$.
bool	L.member(*node* v)	returns true if $v \in L$ and false otherwise.
bool	L (*node* v)	returns true if $v \in L$ and false otherwise.
node	L.first()	returns the first node in L (nil if L is empty).
node	L.last()	returns the last node in L (nil if L is empty).
node	L.succ(*node* v)	returns the successor of v in L. *Precondition*: $v \in L$.
node	L.pred(*node* v)	returns the predecessor of v in L. *Precondition*: $v \in L$.
node	L.cyclic_succ(*node* v)	returns the cyclic successor of v in L. *Precondition*: $v \in L$.
node	L.cyclic_pred(*node* v)	returns the cyclic predecessor of v in L. *Precondition*: $v \in L$.

forall(x, L) { "the elements of L are successively assigned to x" }

7.15 Node Partitions (node_partition)

1. Definition

An instance P of the data type *node_partition* is a partition of the nodes of a graph G.

2. Creation

node_partition $P(graph\ G)$;

creates a *node_partition* P containing for every node v in G a block $\{v\}$.

3. Operations

int	P.same_block(*node* v, *node* w)	returns true if v and w belong to the same block of P, false otherwise.
void	P.union_blocks(*node* v, *node* w)	unites the blocks of P containing nodes v and w.
node	P.find(*node* v)	returns a canonical representative node of the block that contains node v.

4. Implementation

A node partition for a graph G is implemented by a combination of a partition P and a node array of *partition_item* associating with each node in G a partition item in P. Initialization takes linear time, union_blocks takes time $O(1)$ (worst-case), and same_block and find take time $O(\alpha(n))$ (amortized). The space requirement is $O(n)$, where n is the number of nodes of G.

7.16 Node Priority Queues (node_pq)

1. Definition

An instance Q of the parameterized data type *node_pq<P>* is a partial function from the nodes of a graph G to a linearly ordered type P of priorities. The priority of a node is sometimes called the information of the node.

2. Creation

node_pq<P> Q(*graph* G);

creates an instance Q ot type *node_pq<P>* for the nodes of graph G with $dom(Q) = \emptyset$.

3. Operations

void	Q.insert(*node* v, P x)	adds the node v with priority x to Q. *Precondition*: $v \notin dom(Q)$.
P	Q.prio(*node* v)	returns the priority of node v.
bool	Q.member(*node* v)	returns true if v in Q, false otherwise.
void	Q.decrease_p(*node* v, P x)	makes x the new priority of node v. *Precondition*: $x \leq Q$.prio(v).
node	Q.find_min()	returns a node with minimal priority (nil if Q is empty).
void	Q.del(*node* v)	removes the node v from Q.
node	Q.del_min()	removes a node with minimal priority from Q and returns it (nil if Q is empty).
void	Q.clear()	makes Q the empty node priority queue.
int	Q.size()	returns $\|dom(Q)\|$.
int	Q.empty()	returns true if Q is the empty node priority queue, false otherwise.
P	Q.inf(*node* v)	returns the priority of node v.

4. Implementation

Node priority queues are implemented by fibonacci heaps and node arrays. Operations insert, del_node, del_min take time $O(\log n)$, find_min, decrease_inf, empty take time $O(1)$ and clear takes time $O(m)$, where m is the size of Q. The space requirement is $O(n)$, where n is the number of nodes of G.

7.17 Bounded Node Priority Queues (b_node_pq)

1. Definition

An instance of the data type *b_node_pq<N>* is a priority queue of nodes with integer priorities with the restriction that the size of the minimal interval containing all priorities in the queue is bounded by N, the minimum priority is never decreasing, and every node is contained in at most one queue. When applied to the empty queue the del_min - operation returns a special default minimum node defined in the constructor of the queue.

2. Creation

b_node_pq<N> *PQ*;

introduces a variable *PQ* of type *b_node_pq<N>* and initializes it with the empty queue with default minimum node *nil*.

b_node_pq<N> *PQ*(*node v*);

introduces a variable *PQ* of type *b_node_pq<N>* and initializes it with the empty queue with default minimum node *v*.

3. Operations

node	*PQ*.del_min()	removes the node with minimal priority from *PQ* and returns it (the default minimum node if *PQ* is empty).
void	*PQ*.insert(*node w*, *int p*)	adds node *w* with priority *p* to *PQ*.
void	*PQ*.del(*node w*)	deletes node *w* from *PQ*.

4. Implementation

Bounded node priority queues are implemented by cyclic arrays of doubly linked node lists.

5. Example

Using a *b_node_pq* in Dijktra's shortest paths algorithm.

```
int dijkstra(const GRAPH<int,int>& g, node s, node t)
{ node_array<int> dist(g,MAXINT);
  b_node_pq<100> PQ(t);  // on empty queue del_min returns t
  dist[s] = 0;
  for (node v = s;  v != t; v = PQ.del_min() )
  { int dv = dist[v];
    edge e;
    forall_adj_edges(e,v)
    { node w = g.opposite(v,e);
      int d = dv + g.inf(e);
      if (d < dist[w])
      { if (dist[w] != MAXINT) PQ.del(w);
        dist[w] = d;
        PQ.insert(w,d);
       }
     }
   }
  return dist[t];
}
```

7.18 Graph Generators ()

void complete_graph(*graph& G*, *int n*)

creates a complete graph G with n nodes.

void random_graph(*graph& G*, *int n*, *int m*)

creates a random graph G with n nodes and m edges.

void test_graph(*graph& G*)

creates interactively a user defined graph G.

void complete_bigraph(*graph& G*, *int a*, *int b*, *list<node>& A*, *list<node>& B*)

creates a complete bipartite graph G with a nodes on side A and b nodes on side B. All edges are directed from A to B.

void random_bigraph(*graph& G*, *int a*, *int b*, *int m*, *list<node>& A*, *list<node>& B*)

creates a random bipartite graph G with a nodes on side A, b nodes on side B, and m edges. All edges are directed from A to B.

void test_bigraph(*graph& G*, *list<node>& A*, *list<node>& B*)

creates interactively a user defined bipartite graph G with sides A and B. All edges are directed from A to B.

void random_planar_graph(*graph& G*, *int n*)

creates a random planar graph G with n nodes.

void random_planar_graph(*graph& G*, *node_array<double>& xcoord*, *ycoord*, *int n*)

creates a random planar graph G with n nodes embedded into the unit sqare. The embedding is given by $xcoord[v]$ and $ycoord[v]$ for every node v of G.

void triangulated_planar_graph(*graph& G*, *int n*)

creates a triangulated planar graph G with n nodes.

void triangulated_planar_graph(*graph& G*, *node_array<double>& xcoord*, *ycoord*, *int n*)

creates a triangulated planar graph G with n nodes embedded into the unit sqare. The embedding is given by $xcoord[v]$ and $ycoord[v]$ for every node v of G.

void grid_graph(*graph& G*, *int n*)

creates a grid graph G of size $n \times n$ nodes.

void grid_graph(*graph&*, *node_array<double>& xcoord*, *node_array<double>& ycoord*, *int n*)

creates a grid graph G of size $n \times n$ nodes embedded into the unit sqare. The embedding is given by $xcoord[v]$ and $ycoord[v]$ for every node v of G.

void cmdline_graph(*graph& G*, *int argc*, *char ** argv*)

builds graph G as specified by the command line arguments:

prog	$\longrightarrow$	test_graph()
prog n	$\longrightarrow$	complete_graph(n)
prog n m	$\longrightarrow$	test_graph(n, m)
prog *file*	$\longrightarrow$	G.read_graph(*file*).

7.19 Miscellaneous Graph Functions (graph_misc)

bool	Is_Bidirected(*graph G*, *edge_array<edge>& rev*)	computes for every edge $e = (v, w)$ in G its reversal $rev[e] = (w, v)$ in G (nil if not present). Returns true if every edge has a reversal and false otherwise.
bool	Is_Simple(*graph& G*)	returns true if G is simple, i.e., has no parallel edges, false otherwise.
void	Make_Simple(*graph& G*)	makes G simple by removing one of each pair of parallel edges from G.

7.20 Graph Algorithms (graph_alg)

This section gives a summary of the graph algorithms contained in LEDA. All algorithms are generic, i.e., they accept instances of any user defined parameterized graph type *GRAPH<vtype, etype>* as arguments. The header file <LEDA/graph_alg.h> has to be included.

7.20.1 Basic Algorithms

- **Topological Sorting**

bool TOPSORT(*graph& G, node_array<int>& ord*)

TOPSORT takes as argument a directed graph $G(V, E)$. It sorts G topologically (if G is acyclic) by computing for every node $v \in V$ an integer $ord[v]$ such that $1 \leq ord[v] \leq |V|$ and $ord[v] < ord[w]$ for all edges $(v, w) \in E$. TOPSORT returns true if G is acyclic and false otherwise.

The algorithm ([30]) has running time $O(|V| + |E|)$.

- **Depth First Search**

list<node> DFS(*graph& G, node s, node_array<bool>& reached*)

DFS takes as argument a directed graph $G(V, E)$, a node s of G and a node_array *reached* of boolean values. It performs a depth first search starting at s visiting all reachable nodes v with $reached[v]$ = false. For every visited node v $reached[v]$ is changed to true. DFS returns the list of all reached nodes.

The algorithm ([47]) has running time $O(|V| + |E|)$.

list<edge> DFS_NUM(*graph& G, node_array<int>& dfsnum, node_array<int>& compnum*)

DFS_NUM takes as argument a directed graph $G(V, E)$. It performs a depth first search of G numbering the nodes of G in two different ways. *dfsnum* is a numbering with respect to the calling time and *compnum* a numbering with respect to the completion time of the recursive calls. DFS_NUM returns a depth first search forest of G (list of tree edges).

The algorithm ([47]) has running time $O(|V| + |E|)$.

- **Breadth First Search**

list<node> BFS(*graph& G, node s, node_array<int>& dist*)

BFS takes as argument a directed graph $G(V, E)$ and a node s of G. It performs a breadth first search starting at s computing for every visited node v the distance $dist[v]$ from s to v. BFS returns the list of all reached nodes.

The algorithm ([34]) has running time $O(|V| + |E|)$.

- **Connected Components**

int COMPONENTS(*graph& G*, *node_array<int>& compnum*)

COMPONENTS takes a graph $G(V, E)$ as argument and computes the connected components of the underlying undirected graph, i.e., for every node $v \in V$ an integer *compnum*$[v]$ from $[0 \dots c-1]$ where c is the number of connected components of G and v belongs to the i-th connected component iff *compnum*$[v] = i$. COMPONENTS returns c.

The algorithm ([34]) has running time $O(|V| + |E|)$.

- **Strong Connected Components**

int STRONG_COMPONENTS(*graph& G*, *node_array<int>& compnum*)

STRONG_COMPONENTS takes a directed graph $G(V, E)$ as argument and computes for every node $v \in V$ an integer *compnum*$[v]$ from $[0 \dots c-1]$ where c is the number of strongly connected components of G and v belongs to the i-th strongly connected component iff *compnum*$[v] = i$. STRONG_COMPONENTS returns c.

The algorithm ([34]) has running time $O(|V| + |E|)$.

- **Transitive Closure**

graph TRANSITIVE_CLOSURE(*graph& G*)

TRANSITIVE_CLOSURE takes a directed graph $G(V, E)$ as argument and computes the transitive closure of $G(V, E)$. It returns a directed graph $G'(V', E')$ with $V' = V$ and $(v, w) \in E' \Leftrightarrow$ there is a path form v to w in G.

The algorithm ([23]) has running time $O(|V| \cdot |E|)$.

7.20.2 Network Algorithms

Most of the following network algorithms are overloaded. They work for both integer and real valued edge costs.

- **Single Source Shortest Paths**

void DIJKSTRA(*graph& G*, *node s*, *edge_array<int> cost*, *node_array<int> dist*, *node_array<edge> pred*)

void DIJKSTRA(*graph& G*, *node s*, *edge_array<double> cost*, *node_array<double> dist*, *node_array<edge> pred*)

DIJKSTRA takes as arguments a directed graph $G(V, E)$, a source node s and an edge_array *cost* giving for each edge in G a non-negative cost. It computes for each node v in G the distance *dist*$[v]$ from s (cost of the least cost path from s to v) and the predecessor edge *pred*$[v]$ in the shortest path tree.

The algorithm ([13], [22]) has running time $O(|E| + |V| \log |V|)$.

bool BELLMAN_FORD(*graph& G, node s, edge_array<int> cost, node_array<int> dist, node_array<int> pred*)

bool BELLMAN_FORD(*graph& G, node s, edge_array<double> cost, node_array<double> dist, node_array<edge> pred*)

BELLMAN_FORD takes as arguments a graph $G(V, E)$, a source node s and an edge_array *cost* giving for each edge in G a real (integer) cost. It computes for each node v in G the distance $dist[v]$ from s (cost of the least cost path from s to v) and the predecessor edge $pred[v]$ in the shortest path tree. BELLMAN_FORD returns false if there is a negative cycle in G and true otherwise

The algorithm ([5]) has running time $O(|V| \cdot |E|)$.

- **All Pairs Shortest Paths**

bool ALL_PAIRS_SHORTEST_PATHS(*graph& G, edge_array<int>& cost, node_matrix<int>& dist*)

bool ALL_PAIRS_SHORTEST_PATHS(*graph& G, edge_array<double>& cost, node_matrix<double>& dist*)

ALL_PAIRS_SHORTEST_PATHS takes as arguments a graph $G(V, E)$ and an edge_array *cost* giving for each edge in G a real (integer) valued cost. It computes for each node pair (v, w) of G the distance $dist(v, w)$ from v to w (cost of the least cost path from v to w). ALL_PAIRS_SHORTEST_PATHS returns false if there is a negative cycle in G and true otherwise.

The algorithm ([5], [20]) has running time $O(|V| \cdot |E| + |V|^2 \log |V|)$.

- **Maximum Flow**

int MAX_FLOW(*graph& G, node s, node t, edge_array<int>& cap, edge_array<int>& flow*)

int MAX_FLOW(*graph& G, node s, node t, edge_array<double>& cap, edge_array<double>& flow*)

MAX_FLOW takes as arguments a directed graph $G(V, E)$, a source node s, a sink node t and an edge_array *cap* giving for each edge in G a capacity. It computes for every edge e in G a flow $flow[e]$ such that the total flow from s to t is maximal and $flow[e] \leq cap[e]$ for all edges e. MAX_FLOW returns the total flow from s to t.

The algorithm ([26]) has running time $O(|V|^3)$.

int MIN_COST_MAX_FLOW(*graph& G, node s, node t, edge_array<int>& cap, edge_array<int>& cost, edge_array<int>& flow*)

MIN_COST_MAX_FLOW takes as arguments a directed graph $G(V, E)$, a source node s, a sink node t, an edge_array *cap* giving for each edge in G a capacity, and an edge_array *cost* specifying for each edge an integer cost. It computes for every edge e

in G a flow $flow[e]$ such that the total flow from s to t is maximal, the total cost of the flow is minimal, and $flow[e] \leq cap[e]$ for all edges e. MIN_CONST_MAX_FLOW returns the total flow from s to t.

- **Minimum Cut**

list<node> MIN_CUT(*graph& G*, *edge_array<int>& weight*)

MIN_CUT(G, *weight*) takes as arguments a graph G and an edge_array giving for each edge an integer weight. The algorithm ([44]) computes the cut of minimum weight and returns it as a list of nodes. It has running time $O(|V| \cdot |E| + |V|^2 \log |V|)$.

- **Maximum Cardinality Matching**

list<edge> MAX_CARD_MATCHING(*graph& G*)

MAX_CARD_MATCHING(G) computes a maximum cardinality matching of G, i.e., a maximal set of edges M such that no two edges in M share an end point. It returns M as a list of edges.

The algorithm ([16], [48]) has running time $O(|V| \cdot |E| \cdot \alpha(|E|))$.

- **Maximum Cardinality Bipartite Matching**

list<edge> MAX_CARD_BIPARTITE_MATCHING(*graph& G*, *list<node>& A*, *list<node>& B*)

MAX_CARD_BIPARTITE_MATCHING takes as arguments a directed graph $G(V, E)$ and two lists A and B of nodes. All edges in G must be directed from nodes in A to nodes in B. It returns a maximum cardinality matching of G.

The algorithm ([27]) has running time $O(|E|\sqrt{|V|})$.

- **Maximum Weight Bipartite Matching**

list<edge> MAX_WEIGHT_BIPARTITE_MATCHING(*graph& G*, *list<node>& A*, *list<node>& B*, *edge_array<int>& weight*)

list<edge> MAX_WEIGHT_BIPARTITE_MATCHING(*graph& G*, *list<node>& A*, *list<node>& B*, *edge_array<double>& weight*)

MAX_WEIGHT_BIPARTITE_MATCHING takes as arguments a directed graph G, two lists A and B of nodes and an edge_array giving for each edge an integer (real) weight. All edges in G must be directed from nodes in A to nodes in B. It computes a maximum weight bipartite matching of G, i.e., a set of edges M such that the sum of weights of all edges in M is maximal and no two edges in M share an end point. MAX_WEIGHT_BIPARTITE_MATCHING returns M as a list of edges.

The algorithm ([22]) has running time $O(|V| \cdot |E|)$.

- **Spanning Tree**

list<edge> SPANNING_TREE(*graph& G*)

SPANNING_TREE takes as argument a graph $G(V,E)$. It computes a spanning tree T of of the underlying undirected graph, SPANNING_TREE returns the list of edges of T.
The algorithm ([34]) has running time $O(|V|+|E|)$.

- **Minimum Spanning Tree**

list<edge> MIN_SPANNING_TREE(*graph&G*, *edge_array<int>& cost*)

list<edge> MIN_SPANNING_TREE1(*graph&G*, *edge_array<double>& cost*)

MIN_SPANNING_TREE takes as argument an undirected graph $G(V,E)$ and an edge_array *cost* giving for each edge an integer cost. It computes a minimum spanning tree T of G, i.e., a spanning tree such that the sum of all edge costs is minimal. MIN_SPANNING_TREE returns the list of edges of T.
The algorithm ([31]) has running time $O(|E|\log|V|)$.

7.20.3 Algorithms for Planar Graphs

- **Planarity Test**

bool PLANAR(*graph& G*, *bool embed* $= false$)

PLANAR takes as input a directed graph $G(V,E)$ and performs a planarity test for G. If the second argument *embed* has value *true* and G is a planar graph it is transformed into a planar map (a combinatorial embedding such that the edges in all adjacency lists are in clockwise ordering). PLANAR returns true if G is planar and false otherwise.
The algorithm ([28]) has running time $O(|V|+|E|)$.
bool PLANAR(*graph&G*, *list<edge>& el*)

PLANAR takes as input a directed graph $G(V,E)$ and performs a planarity test for G. PLANAR returns true if G is planar and false otherwise. If G is not planar a Kuratowsky-Subgraph is computed and returned in *el*.

- **Triangulation**

list<edge> TRIANGULATE_PLANAR_MAP(*graph& G*)

TRIANGULATE_PLANAR_MAP takes a directed graph G representing a planar map. It triangulates the faces of G by inserting additional edges. The list of inserted edges is returned. *Precondition*: G must be connected.
The algorithm ([29]) has running time $O(|V|+|E|)$.

- **Straight Line Embedding**

int STRAIGHT_LINE_EMBEDDING(*graph& G*,
node_array<int>& xcoord, *node_array<int>& ycoord*)

STRAIGHT_LINE_EMBEDDING takes as argument a directed graph G representing a planar map. It computes a straight line embedding of G by assigning non-negative integer coordinates (*xcoord* and *ycoord*) in the range $0..2(n-1)$ to the nodes. STRAIGHT_LINE_EMBEDDING returns the maximal coordinate.
The algorithm ([19]) has running time $O(|V|^2)$.

Chapter 8

Basic Data Types for Two-Dimensional Geometry

LEDA provides a collection of simple data types for two-dimensional geometry, such as points, segments, lines, circles, and polygons. Furthermore, some basic algorithms (section 8.8) are included.

8.1 Points (point)

1. Definition

An instance of the data type *point* is a point in the two-dimensional plane $\mathbb{R}^2$. We use (a, b) to denote a point with first (or x-) coordinate a and second (or y-) coordinate b.

2. Creation

point p;

introduces a variable p of type *point* initialized to the point $(0, 0)$.

point p(*double* x, *double* y);

introduces a variable p of type *point* initialized to the point (x, y).

point p(*vector* v);

introduces a variable p of type *point* initialized to the point defined by vector v.

3. Operations

double	p.xcoord()	returns the first coordinate of p.
double	p.ycoord()	returns the second coordinate of p.

double	*p*.distance(*point q*)	returns the Euclidean distance between *p* and *q*.
double	*p*.distance()	returns the Euclidean distance between *p* and $(0,0)$.
point	*p*.translate(*double a*, *double d*)	returns the point created by translating *p* in direction *a* by distance *d*. The direction is given by its angle with a right oriented horizontal ray.
point	*p*.translate(*vector v*)	returns $p+v$, i.e., *p* translated by vector *v*. *Precondition*: *v*.dim() = 2.
point	*p*.rotate(*point q*, *double a*)	returns the point created by a rotation of *p* about point *q* by angle *a*.
point	*p*.rotate90(*point q*)	returns the point created by a rotation of *p* about point *q* by an angle of 90 degree.
point	*p*.rotate(*double a*)	returns *p*.rotate(*point*(0, 0), *a*).
point	*p*.rotate90()	returns *p*.rotate90(*point*(0, 0)).
int	$p == q$	Test for equality.
int	$p \mathrel{!=} q$	Test for inequality.
point	*p* + *vector v*	Translation by vector *v*.
ostream&	*ostream& O* << *p*	writes *p* to output stream *O*.
istream&	*istream& I* >> *& p*	reads the coordinates of *p* (two *double* numbers) from input stream *I*.

Non-Member Functions

bool	identical(*point p*, *point q*)	Test for identity.
int	orientation(*point a*, *point b*, *point c*)	computes the orientation of points *a*, *b*, *c*, i.e., the sign of the determinant $\begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix}$

bool collinear(*point a*, *point b*, *point c*)

returns true if points a, b, c are collinear, i.e., $orientation(a, b, c) = 0$, and false otherwise.

bool right_turn(*point a*, *point b*, *point c*)

returns true if points a, b, c form a righ turn, i.e., $orientation(a, b, c) > 0$, and false otherwise.

bool left_turn(*point a*, *point b*, *point c*)

returns true if points a, b, c form a left turn, i.e., $orientation(a, b, c) < 0$, and false otherwise.

bool incircle(*point a*, *point b*, *point c*, *point d*)

returns true if point d lies in the interior of the circle through points a, b, and c, and false otherwise.

8.2 Rational Points (rat_point)

1. Definition

An instance of the data type *rat_point* is a point with rational coordinates in the two-dimensional plane. A point (a, b) is represented by homogeneous coordinates (x, y, w) of arbitrary length integers (see 3.1) such that $a = x/w$ and $b = y/w$.

2. Creation

rat_point *p*;

introduces a variable *p* of type *rat_point* initialized to the point $(0, 0)$.

rat_point *p*(*integer a*, *integer b*);

introduces a variable *p* of type *rat_point* initialized to the point (a, b).

rat_point *p*(*integer x*, *integer y*, *integer w*);

introduces a variable *p* of type *rat_point* initialized to the point with homogeneous coordinates(x, y, w).

3. Operations

double	*p*.xcoord()	returns a double precision floating point approximation of the x-coordinate of *p*.
double	*p*.ycoord()	returns a double precision floating point approximation of the y-coordinate of *p*.
integer	*p*.X()	returns the first homogeneous coordinate of *p*.
integer	*p*.Y()	returns the second homogeneous coordinate of *p*.
integer	*p*.W()	returns the third homogeneous coordinate of *p*.
double	*p*.XD()	returns a floating point approximation of the first homogeneous coordinate of *p*.
double	*p*.YD()	returns a floating point approximation of the second homogeneous coordinate of *p*.
double	*p*.WD()	returns a floating point approximation of the third homogeneous coordinate of *p*.
rat_point	*p*.rotate90(*rat_point p*)	returns *p* rotate by 90 degrees about *p*.
rat_point	*p*.rotate90()	returns *p* rotate by 90 degrees about the origin.
rat_point	*p*.translate(*rat_point p*, vector)	returns *p* translated by ...
bool	*identical*	

(rat_point p, rat_point q)

Test for identity ...

bool	*rat_point p* == *rat_point q*	Test for equality.
bool	*rat_point p* != *rat_point q*	Test for inequality.
ostream&	*ostream& O* << *rat_point p*	writes the homogeneous coordinates (x, y, w) of p to output stream O.
istream&	*istream& I* >> *rat_point& p*	reads the homogeneous coordinates (x, y, w) of p from input stream I.
int	orientation(*rat_point a*, *rat_point b*, *rat_point c*)	computes the orientation of points a, b, c, i.e., the sign of the determinant $\begin{vmatrix} a_x & a_y & a_w \\ b_x & b_y & b_w \\ c_x & c_y & c_w \end{vmatrix}$
bool	collinear(*rat_point a*, *rat_point b*, *rat_point c*)	returns true if points a, b, c are collinear, i.e., $orientation(a, b, c) = 0$, and false otherwise.
bool	right_turn(*rat_point a*, *rat_point b*, *rat_point c*)	returns true if points a, b, c form a righ turn, i.e., $orientation(a, b, c) > 0$, and false otherwise.
bool	left_turn(*rat_point a*, *rat_point b*, *rat_point c*)	returns true if points a, b, c form a left turn, i.e., $orientation(a, b, c) < 0$, and false otherwise.

8.3 Segments (segment)

1. Definition

An instance s of the data type *segment* is a directed straight line segment in the two-dimensional plane, i.e., a straight line segment $[p, q]$ connecting two points $p, q \in \mathbb{R}^2$. p is called the start point and q is called the end point of s. The length of s is the Euclidean distance between p and q. The angle between a right oriented horizontal ray and s is called the direction of s. The segment $[(0,0),(0,0)]$ is said to be empty.

2. Creation

segment s(*point* p, *point* q);

introduces a variable s of type *segment*. s is initialized to the segment (p, q)

segment s(*point* p, *vector* v);

introduces a variable s of type *segment*. s is initialized to the segment $(p, p+v)$. *Precondition*: $v.dim() = 2$.

segment s(*double* $x1$, *double* $y1$, *double* $x2$, *double* $y2$);

introduces a variable s of type *segment*. s is initialized to the segment $[(x_1, y_1), (x_2, y_2)]$.

segment s(*point* p, *double* *dir*, *double* *length*);

introduces a variable s of type *segment*. s is initialized to the segment with start point p, direction *dir*, and length *length*.

segment s;

introduces a variable s of type *segment*. s is initialized to the empty segment.

3. Operations

point	s.source()	returns the source point of segment s.
point	s.target()	returns the target point of segment s.
double	s.xcoord1()	returns the x-coordinate of s.start().
double	s.xcoord2()	returns the x-coordinate of s.end().
double	s.ycoord1()	returns the y-coordinate of s.start().
double	s.ycoord2()	returns the y-coordinate of s.end().
double	s.dx()	returns the $xcoord2 - xcoord1$.

double	*s*.dy()	returns the *ycoord*2 − *ycoord*1.
double	*s*.length()	returns the length of *s*.
double	*s*.direction()	returns the direction of *s* as an angle in the intervall $(-\pi, \pi]$.
double	*s*.angle()	returns *s*.direction().
double	*s*.angle(*segment t*)	returns the angle between *s* and *t*, i.e., *t*.direction() - *s*.direction().
bool	*s*.vertical()	returns true iff *s* is vertical.
bool	*s*.horizontal()	returns true iff *s* is horizontal.
double	*s*.slope()	returns the slope of *s*. *Precondition*: *s* is not vertical.
bool	*s*.intersection(*segment t*, *point& p*)	if *s* and *t* are not collinear and intersect the intersection point is assigned to *p* and true is returned, otherwise false is returned.
bool	*s*.intersection_of_lines(*segment t*, *point& p*)	if *s* and *t* are not collinear and the underlying lines intersect the point of intersection is assigned to *p* and true is returned, otherwise false is returned.
segment	*s*.translate(*double a*, *double d*)	returns the segment created by a translation of *s* in direction *a* by distance *d*.
segment	*s*.translate(*vector v*)	returns $s + v$, i.e., the segment created by translating *s* by vector *v*. *Precondition*: *v*.dim() = 2.
segment	*s*.rotate(*point q*, *double a*)	returns the segment created by a rotation of *s* about point *q* by angle *a*.
segment	*s*.rotate(*double a*)	returns *s*.rotate(*s*.start(), *a*).
segment	*s*.rotate90(*point q*)	returns the segment created by a rotation of *s* about point *q* by an angle of 90 degrees.
segment	*s*.rotate90()	returns *s*.rotate90(*s*.start(), *a*).
int	*s* == *t*	Test for equality.
int	*s* != *t*	Test for inequality.

segment	*s* + *vector v*	Translation by vector *v*.
ostream&	*ostream& O* << *s*	writes *s* to output stream *O*.
istream&	*istream& I* >> & *s*	reads the coordinates of *s* (four *double* numbers) from input stream *I*.

Non-Member Functions

bool	identical(*segment s*1, *segment s*2)	Test for identity.
int	orientation(*segment s*, *point p*)	computes orientation(a, b, p), where $a \neq b$ and a and b appear in this order on segment s.
int	cmp_slopes(*segment s*1, *segment s*2)	returns compare(slope(s_1), slope(s_2)).
bool	parallel(*segment s*1, *segment s*2)	returns (cmp_slopes(s_1, s_2) == 0).

8.4 Rational Segments (rat_segment)

1. Definition

An instance s of the data type *rat_segment* is a directed straight line segment with rational coordinates in the two-dimensional plane, i.e., a line segment $[p, q]$ connecting two rational points p and q (cf. 8.2). p is called the start point and q is called the end point of s. The segment $[(0,0),(0,0)]$ is said to be empty.

2. Creation

rat_segment s;

introduces a variable s of type *rat_segment*. s is initialized to the empty segment.

rat_segment s(*rat_point* p, *rat_point* q);

introduces a variable s of type *rat_segment*. s is initialized to the segment (p, q).

rat_segment s(*integer* $x1$, *integer* $y1$, *integer* $x2$, *integer* $y2$);

introduces a variable s of type *rat_segment*. s is initialized to the segment $[(x1, y1), (x2, y2)]$.

3. Operations

rat_point	s.source()	returns the source point of segment s.
rat_point	s.target()	returns the target point of segment s.
double	s.xcoord1()	returns a double precision approximation of the x-coordinate of the start point of segment s.
double	s.xcoord2()	returns a double precision approximation of the x-coordinate of the end point of segment s.
double	s.ycoord1()	returns a double precision approximation of the y-coordinate of the start point of segment s.
double	s.ycoord2()	returns a double precision approximation of the y-coordinate of the end point of segment s.
integer	s.X1()	returns the first homogeneous coordinate of the start point of segment s.
integer	s.X2()	returns the first homogeneous coordinate of the end point of segment s.
integer	s.Y1()	returns the second homogeneous coordinate of the start point of segment s.

integer	s.Y2()	returns the second homogeneous coordinate of the end point of segment s.
integer	s.W1()	returns the third homogeneous coordinate of the start point of segment s.
integer	s.W2()	returns the third homogeneous coordinate of the end point of segment s.
double	s.XD1()	returns a floating point approximation of the first homogeneous coordinate of the start point of segment s.
double	s.XD2()	returns a floating point approximation of the first homogeneous coordinate of the end point of segment s.
double	s.YD1()	returns a floating point approximation of the second homogeneous coordinate of the start point of segment s.
double	s.YD2()	returns a floating point approximation of the second homogeneous coordinate of the end point of segment s.
double	s.WD1()	returns a floating point approximation of the third homogeneous coordinate of the start point of segment s.
double	s.WD2()	returns a floating point approximation of the third homogeneous coordinate of the end point of segment s.
integer	s.dx()	returns the normalized x-difference $X1 \cdot W2 - X2 \cdot W1$ of the segment.
integer	s.dy()	returns the normalized y-difference $Y1 \cdot W2 - Y2 \cdot W1$ of the segment.
double	s.dxd()	returns the optimal floating point approximation of the normalized x-difference of the segment.
double	s.dyd()	returns the optimal floating point approximation of the normalized y-difference of the segment.
bool	s.vertical()	returns true if s is vertical and false otherwise.
bool	s.horizontal()	returns true if s is horizontal and false otherwise.
int	s.cmp_slope(*rat_segment* $s1$)	compares the slopes of s and s_1.
bool	s.intersection(*rat_segment* t, *rat_point&* p)	

if s and t are not collinear and intersect the point of intersection is assigned to p and the result is true, otherwise the result is false.

bool s.intersection_of_lines(*rat_segment t*, *rat_point& p*)

if the lines supporting s and t are not parallel their point of intersection is assigned to p and the result is true, otherwise the result is false.

int s == t Test for equality.

int s != t Test for inequality.

bool *identical*

(rat_segment s1, rat_segment s2)

Test for identity ...

ostream& *ostream& O* << *rat_segment s*

writes the homogeneous coordinates of s (six *integer* numbers) to output stream O.

istream& *istream& I* >> *rat_segment& s*

reads the homogeneous coordinates of s (six *integer* numbers) from input stream I.

int orientation(*rat_segment s*, *rat_point p*)

computes orientation(a, b, p), where $a \neq b$ and a and b appear in this order on segment s.

int cmp_slopes(*rat_segment s*1, *rat_segment s*2)

returns compare(slope(s_1), slope(s_2)).

bool intersection(*rat_segment s*1, *rat_segment s*2)

decides whether $s1$ and $s2$ intersect.

8.5 Straight Lines (line)

1. Definition

An instance l of the data type *line* is a directed straight line in the two-dimensional plane. The angle between a right oriented horizontal line and l is called the direction of l.

2. Creation

line l(*point* p, *point* q);

introduces a variable l of type *line*. l is initialized to the line passing through points p and q directed form p to q.

line l(*segment* s);

introduces a variable l of type *line*. l is initialized to the line supporting segment s.

line l(*point* p, *vector* v);

introduces a variable l of type *line*. l is initialized to the line of all poinnts $p + \lambda v$. *Precondition*: $v.dim() = 2$ and $v.length() > 0$.

line l(*point* p, *double* a);

introduces a variable l of type *line*. l is initialized to the line passing through point p with direction a.

line l;

introduces a variable l of type *line*. l is initialized to the line passing through the origin with direction 0.

3. Operations

double	l.direction()	returns the direction of l.
double	l.angle(*line* g)	returns the angle between l and g, i.e., g.direction() $-$ l.direction().
double	l.angle()	returns l.direction().
bool	l.vertical()	returns true iff l is vertical.
bool	l.horizontal()	returns true iff l is horizontal.
double	l.slope()	returns the slope of l. *Precondition*: l is not vertical.
double	l.y_proj(*double* x)	returns p.ycoord(), where $p \in l$ with p.xcoord() $= x$. *Precondition*: l is not vertical.

double	l.x_proj(*double* y)	returns p.xcoord(), where $p \in l$ with p.ycoord() $= y$. *Precondition*: l is not horizontal.
double	l.y_abs()	returns the y-abscissa of l (l.y_proj(0)). *Precondition*: l is not vertical.
bool	l.intersection(*line* g, *point&* *inter*)	if l and g are not collinear and intersect the intersection point is assigned to *inter* and true is returned, otherwise false is returned.
bool	l.intersection(*segment* s, *point&* *inter*)	if l and s are not collinear and intersect the intersection point is assigned to *inter* and true is returned, otherwise false is returned.
line	l.translate(*double* a, *double* d)	returns the line created by a translation of l in direction a by distance d.
line	l.translate(*vector* v)	returns $l+v$, i.e., the line created by translating l by vector v. *Precondition*: v.dim() = 2.
line	l.rotate(*point* q, *double* a)	returns the line created by a rotation of l about point q by angle a.
line	l.rotate(*double* a)	returns l.rotate(*point*(0,0), a).
segment	l.perpendicular(*point* p)	returns the normal of p with respect to l.
bool	l == g	Test for equality.
bool	l != g	Test for inequality.
int	orientation(*line* l, *point* p)	computes orientation(a, b, p), where $a \neq b$ and a and b appear in this order on line l.
int	cmp_slopes(*line* $l1$, *line* $l2$)	returns compare(slope(l_1), slope(l_2)).

8.6 Polygons (polygon)

1. Definition

An instance P of the data type *polygon* is a simple polygon in the two-dimensional plane defined by the sequence of its vertices in clockwise order. The number of vertices is called the size of P. A polygon with empty vertex sequence is called empty.

2. Creation

polygon P(*list<point> pl*);

introduces a variable P of type *polygon*. P is initialized to the polygon with vertex sequence *pl*.
Precondition: The vertices in *pl* are given in clockwise order and define a simple polygon.

polygon P;

introduces a variable P of type *polygon*. P is initialized to the empty polygon.

3. Operations

list<point>	P.vertices()	returns the vertex sequence of P.
list<segment>	P.segments()	returns the sequence of bounding segments of P in clockwise order.
list<point>	P.intersection(*segment s*)	returns $P \cap s$ as a list of points.
list<point>	P.intersection(*line l*)	returns $P \cap l$ as a list of points.
list<polygon>	P.intersection(*polygon Q*)	returns $P \cap Q$ as a list of polygons.
bool	P.inside(*point p*)	returns true if p lies inside of P, false otherwise.
bool	P.outside(*point p*)	returns $!P$.inside(p).
polygon	P.translate(*double a*, *double d*)	returns the polygon created by a translation of P in direction a by distance d.
polygon	P.translate(*vector v*)	returns $P + v$, i.e., the polygon created by translating P by vector v. *Precondition*: $v.dim() = 2$.
polygon	P.rotate(*point q*, *double a*)	returns the polygon created by a rotation of P about point q by angle a.
int	P.size()	returns the size of P.
bool	P.empty()	returns true if P is empty, false otherwise.

8.7 Circles (circle)

1. Definition

An instance C of the data type *circle* is a circle in the two-dimensional plane, i.e., the set of points having a certain distance r from a given point p. r is called the radius and p is called the center of C. The circle with center $(0,0)$ and radius 0 is called the empty circle.

2. Creation

circle C(*point* c, *double* r);

introduces a variable C of type *circle*. C is initialized to the circle with center c and radius r.

circle C(*double* x, *double* y, *double* r);

introduces a variable C of type *circle*. C is initialized to the circle with center (x, y) and radius r.

circle C(*point* a, *point* b, *point* c);

introduces a variable C of type *circle*. C is initialized to the circle through points a, b, and c. *Precondition*: a, b, and c are not collinear.

circle C;

introduces a variable C of type *circle*. C is initialized to the empty circle.

3. Operations

point	C.center()	returns the center of C.
double	C.radius()	returns the radius of C.
list<point>	C.intersection(*circle* D)	returns $C \cap D$ as a list of points.
list<point>	C.intersection(*line* l)	returns $C \cap l$ as a list of points.
list<point>	C.intersection(*segment* s)	returns $C \cap s$ as a list of points.
segment	C.left_tangent(*point* p)	returns the line segment starting in p tangent to C and left of segment $[p, C.center()]$.
segment	C.right_tangent(*point* p)	returns the line segment starting in p tangent to C and right of segment $[p, C.center()]$.
double	C.distance(*point* p)	returns the distance between C and p (negative if p inside C).
double	C.distance(*line* l)	returns the distance between C and l (negative if l intersects C).

double	C.distance(*circle* D)	returns the distance between C and D (negative if D intersects C).
bool	C.inside(*point* p)	returns true if p lies inside of C, false otherwise.
bool	C.outside(*point* p)	returns $!C$.inside(p).
circle	C.translate(*double* a, *double* d)	returns the circle created by a translation of C in direction a by distance d.
circle	C.translate(*vector* v)	returns $C + v$, i.e., the circle created by translating C by vector v. *Precondition*: v.dim() = 2.
circle	C.rotate(*point* q, *double* a)	returns the circle created by a rotation of C about point q by angle a.
circle	C.rotate(*double* a)	returns the circle created by a rotation of C about the origin by angle a.
bool	$C == D$	Test for equality.
bool	$C\ != D$	Test for inequality.

8.8 Plane Algorithms (plane_alg)

- **Triangulations**

- **Line segment intersection**

void SWEEP_SEGMENTS(*list<segment> L*, *GRAPH<point, int>& G*);

SWEEP_SEGMENTS takes a list of segments L as input and computes the planar graph G induced by the set of straight line segments in L. The nodes of G are all endpoints and all proper intersection points of segments in L. The edges of G are the maximal relatively open subsegments of segments in L that contain no node of G. All edges are directed from left to right or upwards. The algorithm ([6]) runs in time $O((n + s)\log n)$ where n is the number of segments and s is the number of vertices of the graph G.

void SEGMENT_INTERSECTION(*list<segment> L*, *list<point>& P*);

SEGMENT_INTERSECTION takes a list of segments L as input and computes the list of intersection points between all segments in L.
The algorithm ([6]) has running time $O((n + k)\log n)$, where n is the number of segments and k is the number of intersections.

- **Convex Hulls**

list<point> CONVEX_HULL(*list<point> L*);

CONVEX_HULL takes as argument a list of points and returns the polygon representing the convex hull of L. It is based on a randomized incremental algorithm.
Running time: $O(n \log n)$ (with high probability), where n is the number of points.

list<point> CONVEX_HULL(*list<point>* *L*); *list<rat_point>* CONVEX_HULL(*list<rat_point> L*);

CONVEX_HULL takes as argument a list of (rational) points and returns the polygon representing the convex hull of L.
Running time: $O(n \log n)$ (with high probability), where n is the number of points.

- **Voronoi Diagrams**

void VORONOI(*list<point>& sites*, *double R*, *GRAPH<point, point>& G*)

VORONOI takes as input a list of points *sites* and a real number R. It computes a directed graph G representing the planar subdivision defined by the Voronoi-diagram of *sites* where all "infinite" edges have length R. For each node v G.inf(v) is the corresponding Voronoi vertex (*point*) and for each edge e G.inf(e) is the site (*point*) whose Voronoi region is bounded by e.
The algorithm ([12]) has running time $O(n \log n)$ (with high probability), where n is the number of sites.

Chapter 9

Advanced Data Types for Two-Dimensional Geometry

9.1 Two-Dimensional Dictionaries (d2_dictionary)

1. Definition

An instance D of the parameterized data type *d2_dictionary*$<K1, K2, I>$ is a collection of items (*dic2_item*). Every item in D contains a key from the linearly ordered data type $K1$, a key from the linearly ordered data type $K2$, and an information from data type I. $K1$ and $K2$ are called the key types of D, and I is called the information type of D. The number of items in D is called the size of D. A two-dimensional dictionary of size zero is said to be empty. We use $<k_1, k_2, i>$ to denote the item with first key k_1, second key k_2, and information i. For each pair $(k_1, k_2) \in K1 \times K2$ there is at most one item $<k_1, k_2, i> \in D$. Additionally to the normal dictionary operations, the data type *d2_dictionary* supports rectangular range queries on $K1 \times K2$.

2. Creation

d2_dictionary$<K1, K2, I>$ D;

creates an instance D of type *d2_dictionary*$<K1, K2, I>$ and initializes D to the empty dictionary.

3. Operations

$K1$	D.key1(*dic2_item it*)	returns the first key of item *it*. *Precondition*: *it* is an item in D.
$K2$	D.key2(*dic2_item it*)	returns the second key of item *it*. *Precondition*: *it* is an item in D.
I	D.inf(*dic2_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in D.
dic2_item	D.min_key1()	returns the item with minimal first key.

dic2_item	D.min_key2()	returns the item with minimal second key.
dic2_item	D.max_key1()	returns the item with maximal first key.
dic2_item	D.max_key2()	returns the item with maximal second key.
dic2_item	D.insert($K1\ x,\ K2\ y,\ I\ i$)	associates the information i with the keys x and y. If there is an item $<x,y,j>$ in D then j is replaced by i, else a new item $<x,y,i>$ is added to D. In both cases the item is returned.
dic2_item	D.lookup($K1\ x,\ K2\ y$)	returns the item with keys x and y (nil if no such item exists in D).
list<dic2_item>	D.range_search($K1\ x0,\ K1\ x1,\ K2\ y0,\ K2\ y1$)	returns the list of all items $<k_1,k_2,i>$ in D with $x_0 \le k_1 \le x_1$ and $y_0 \le k_2 \le y_1$.
list<dic2_item>	D.all_items()	returns the list of all items of D.
void	D.del($K1\ x,\ K2\ y$)	deletes the item with keys x and y from D.
void	D.del_item(*dic2_item it*)	removes item *it* from D. *Precondition*: *it* is an item in D.
void	D.change_inf(*dic2_item it*, $I\ i$)	makes i the information of item *it*. *Precondition*: *it* is an item in D.
void	D.clear()	makes D the empty d2_dictionary.
bool	D.empty()	returns true if D is empty, false otherwise.
int	D.size()	returns the size of D.

4. Implementation

Two-dimensional dictionaries are implemented by dynamic two-dimensional range trees [50, 33] based on BB[α] trees. Operations insert, lookup, del_item, del take time $O(\log^2 n)$, range_search takes time $O(k + \log^2 n)$, where k is the size of the returned list, key, inf, empty, size, change_inf take time $O(1)$, and clear takes time $O(n \log n)$. Here n is the current size of the dictionary. The space requirement is $O(n \log n)$.

9.2 Sets of Two-Dimensional Points (point_set)

1. Definition

An instance S of the parameterized data type *point_set<I>* is a collection of items (*ps_item*). Every item in S contains a two-dimensional point as key (data type *point*), and an information from data type I, called the information type of S. The number of items in S is called the size of S. A point set of size zero is said to be empty. We use $<p,i>$ to denote the item with point p, and information i. For each point p there is at most one item $<p,i> \in S$. Beside the normal dictionary operations, the data type *point_set* provides operations for rectangular range queries and nearest neighbor queries.

2. Creation

point_set<I> S;

creates an instance S of type *point_set<I>* and initializes S to the empty set.

3. Operations

point	S.key(*ps_item it*)	returns the point of item *it*. *Precondition*: *it* is an item in S.
I	S.inf(*ps_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in S.
ps_item	S.insert(*point p*, *I i*)	associates the information i with point p. If there is an item $<p,j>$ in S then j is replaced by i, else a new item $<p,i>$ is added to S. In both cases the item is returned.
ps_item	S.lookup(*point p*)	returns the item with point p (nil if no such item exists in S).
ps_item	S.nearest_neighbor(*point q*)	returns the item $<p,i> \in S$ such that the distance between p and q is minimal.
list<ps_item>	S.range_search(*double x*0, *double x*1, *double y*0, *double y*1)	returns all items $<p,i> \in S$ with $x_0 \leq p.\text{xcoord}() \leq x_1$ and $y_0 \leq p.\text{ycoord}() \leq y_1$.
list<ps_item>	S.convex_hull()	returns the list of items containing all points of the convex hull of S in clockwise order.
void	S.del(*point p*)	deletes the item with point p from S.

void	S.del_item(*ps_item it*)	removes item *it* from S. *Precondition*: *it* is an item in S.
void	S.change_inf(*ps_item it*, $I\ i$)	makes i the information of item *it*. *Precondition*: *it* is an item in S.
list<ps_item>	S.all_items()	returns the list of all items in S.
list<point>	S.all_points()	returns the list of all points in S.
void	S.clear()	makes S the empty point_set.
bool	S.empty()	returns true iff S is empty.
int	S.size()	returns the size of S.

4. Implementation

Point sets are implemented by a combination of two-dimensional range trees [50, 33] and Voronoi diagrams. Operations insert, lookup, del_item, del take time $O(\log^2 n)$, key, inf, empty, size, change_inf take time $O(1)$, and clear takes time $O(n \log n)$. A range_search operation takes time $O(k + \log^2 n)$, where k is the size of the returned list. A nearest_neighbor query takes time $O(n^2)$, if it follows any update operation (insert or delete) and $O(\log n)$ otherwise. Here n is the current size of the point set. The space requirement is $O(n^2)$.

9.3 Sets of Intervals (interval_set)

1. Definition

An instance S of the parameterized data type *interval_set<I>* is a collection of items (*is_item*). Every item in S contains a closed interval of the *double* numbers as key and an information from data type I, called the information type of S. The number of items in S is called the size of S. An interval set of size zero is said to be empty. We use $<x, y, i>$ to denote the item with interval $[x, y]$ and information i; x (y) is called the left (right) boundary of the item. For each interval $[x, y]$ there is at most one item $<x, y, i> \in S$.

2. Creation

interval_set<I> S;

creates an instance S of type *interval_set<I>* and initializes S to the empty set.

3. Operations

double	S.left(*is_item it*)	returns the left boundary of item *it*. *Precondition*: *it* is an item in S.
double	S.right(*is_item it*)	returns the right boundary of item *it*. *Precondition*: *it* is an item in S.
I	S.inf(*is_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in S.
is_item	S.insert(*double x*, *double y*, *I i*)	associates the information i with interval $[x, y]$. If there is an item $<x, y, j>$ in Sthen j is replaced by i, else a new item $<x, y, i>$ is added to S. In both cases the item is returned.
is_item	S.lookup(*double x*, *double y*)	returns the item with interval $[x, y]$ (nil if no such item exists in S).
list<is_item>	S.intersection(*double a*, *double b*)	returns all items $<x, y, i> \in S$ with $[x, y] \cap [a, b] \neq \emptyset$.
void	S.del(*double x*, *double y*)	deletes the item with interval $[x, y]$ from S.
void	S.del_item(*is_item it*)	removes item *it* from S. *Precondition*: *it* is an item in S.
void	S.change_inf(*is_item it*, *I i*)	makes i the information of item *it*. *Precondition*: *it* is an item in S.
void	S.clear()	makes S the empty interval_set.
bool	S.empty()	returns true iff S is empty.
int	S.size()	returns the size of S.

4. Implementation

Interval sets are implemented by two-dimensional range trees [50, 33]. Operations insert, lookup, del_item and del take time $O(\log^2 n)$, intersection takes time $O(k+\log^2 n)$, where k is the size of the returned list. Operations left, right, inf, empty, and size take time $O(1)$, and clear $O(n \log n)$. Here n is always the current size of the interval set. The space requirement is $O(n \log n)$.

9.4 Sets of Parallel Segments (segment_set)

1. Definition

An instance S of the parameterized data type *segment_set<I>* is a collection of items (*seg_item*). Every item in S contains as key a line segment with a fixed direction α (see data type segment) and an information from data type I, called the information type of S. α is called the orientation of S. We use $<s,i>$ to denote the item with segment s and information i. For each segment s there is at most one item $<s,i> \in S$.

2. Creation

segment_set<I> S(*double a*);

creates an empty instance S of type *segment_set<I>* with orientation a.

segment_set<I> S;

creates an empty instance S of type *segment_set<I>* with orientation zero, i.e., horizontal segments.

3. Operations

segment	S.key(*seg_item it*)	returns the segment of item *it*. *Precondition*: *it* is an item in S.
I	S.inf(*seg_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in S.
seg_item	S.insert(*segment s*, *I i*)	associates the information i with segment s. If there is an item $<s,j>$ in S then j is replaced by i, else a new item $<s,i>$ is added to S. In both cases the item is returned.
seg_item	S.lookup(*segment s*)	returns the item with segment s (nil if no such item exists in S).
list<seg_item>	S.intersection(*segment q*)	returns all items $<s,i> \in S$ with $s \cap q \neq \emptyset$. *Precondition*: q is orthogonal to the segments in S.
list<seg_item>	S.intersection(*line l*)	returns all items $<s,i> \in S$ with $s \cap l \neq \emptyset$. *Precondition*: l is orthogonal to the segments in S.
void	S.del(*segment s*)	deletes the item with segment s from S.
void	S.del_item(*seg_item it*)	removes item *it* from S. *Precondition*: *it* is an item in S.
void	S.change_inf(*seg_item it*, *I i*)	

		makes i the information of item *it*. *Precondition*: *it* is an item in S.
void	S.clear()	makes S the empty segment_set.
bool	S.empty()	returns true iff S is empty.
int	S.size()	returns the size of S.

4. Implementation

Segment sets are implemented by dynamic segment trees based on BB[α] trees ([50, 33]) trees. Operations key, inf, change_inf, empty, and size take time $O(1)$, insert, lookup, del, and del_item take time $O(\log^2 n)$ and an intersection operation takes time $O(k + \log^2 n)$, where k is the size of the returned list. Here n is the current size of the set. The space requirement is $O(n \log n)$.

9.5 Planar Subdivisions (subdivision)

1. Definition

An instance S of the parameterized data type *subdivision<I>* is a subdivision of the two-dimensional plane, i.e., an embedded planar graph with straight line edges (see also sections 7.5 and 7.6). With each node v of S is associated a point, called the position of v and with each face of S is associated an information from data type I, called the information type of S.

2. Creation

subdivision<I> S(*GRAPH<point, I>* G);

creates an instance S of type *subdivision<I>* and initializes it to the subdivision represented by the parameterized directed graph G. The node entries of G (of type point) define the positions of the corresponding nodes of S. Every face f of S is assigned the information of one of its bounding edges in G.
Precondition: G represents a planar subdivision, i.e., a straight line embedded planar map.

3. Operations

point	S.position(*node* v)	returns the position of node v.
I	S.inf(*face* f)	returns the information of face f.
face	S.locate_point(*point* p)	returns the face containing point p.

4. Implementation

Planar subdivisions are implemented by parameterized planar maps and an additional data structure for point location based on persistent search trees [15]. Operations position and inf take constant time, a locate_point operation takes time $O(\log^2 n)$. Here n is the number of nodes. The space requirement and the initialization time is $O(n^2)$.

Chapter 10

Graphics

10.1 Graphic Windows (window)

1. Definition

The data type *window* provides an interface for the input and output of basic two-dimensional geometric objects (cf. section 5.1) using the X11 window system. Application programs using data type *window* have to be linked with *libWx.a* and the X11 library (cf. section 1.6):

CC *prog.c* -lP -lG -lL -lWx -lX11 -lm

An instance W of the data type *window* is an iso-oriented rectangular window in the two-dimensional plane. The default representation of W on the screen is a square of maximal possible edge length positioned in the upper right corner (cf. creation, variant c)). The coordinates and scaling of W used for drawing operations are defined by three double parameters: x_0, the x-coordinate of the left side, x_1, the x-coordinate of the right side, and y_0, the y-coordinate of the bottom side. The y-coordinate of the top side of W is determined by the current size and shape of the window on the screen, which can be changed interactively. A graphic window supports operations for drawing points, lines, segments, arrows, circles, polygons, graphs, ... and for graphical input of all these objects using the mouse input device. Most of the drawing operations have an optional color argument. Possible colors are *black* (default), *white*, *blue*, *green*, *red*, *violet*, and *orange*. On monochrome displays all colors different from *white* are turned to *black*. There are 6 parameters used by the drawing operations:

1. The *line width* parameter (default value 1 pixel) defines the width of all kinds of lines (segments, arrows, edges, circles, polygons).

2. The *line style* parameter defines the style of lines. Possible line styles are *solid* (default), *dashed*, and *dotted*.

3. The *node width* parameter (default value 10 pixels) defines the diameter of nodes created by the draw_node and draw_filled_node operations.

4. The *text mode* parameter defines how text is inserted into the window. Possible values are *transparent* (default) and *opaque*.

5. The *drawing mode* parameter defines the logical operation that is used for setting pixels in all drawing operations. Possible values are *src_mode* (default) and *xor_mode*. In *src_mode* pixels are set to the respective color value, in *xor_mode* the value is bitwise added to the current pixel value.

6. The *redraw function* parameter is used to redraw the entire window whenever a redrawing is necessary, e.g., if the window shape on the screen has been changed. Its type is pointer to a void-function taking no arguments, i.e., void (*F)();

2. Creation

window $W(float\ xpix,\ float\ ypix,\ float\ xpos,\ float\ ypos);$

creates a window W of physical size $xpix \times ypix$ pixels with its upper left corner at position $(xpos, ypos)$ on the screen.

window $W(float\ xpix,\ float\ ypix);$

creates a window W of physical size $xpix \times ypix$ pixels positioned into the upper right corner of the screen.

window $W;$

creates a maximal squared window W positioned into the upper right corner of the screen.

All three variants initialize the coordinates of W to $x_0 = 0$, $x_1 = 100$ and $y_0 = 0$. The *init* operation (see below) can later be used to change the window coordinates and scaling.

3. Operations

3.1 Initialization

void W.init(*double* $x0$, *double* $x1$, *double* $y0$)

sets the coordinates of W to x_0, x_1, and y_0.

void W.set_grid_mode(*int* d)

adds a rectangular grid with integer coordinates and grid distance d to W, if $d > 0$. Removes grid from W, if $d \leq 0$.

void W.init(*double* $x0$, *double* $x1$, *double* $y0$, *int* d)

like init(x_0, x_1, y_0) followed by set_grid_mode(d).

void W.clear(*color* c = *BG_color*)

clears W.

3.2 Setting parameters

int	*W*.set_line_width(*int pix*)	sets the line width parameter to *pix* pixels and returns its previous value.
line_style	*W*.set_line_style(*line_style s*)	sets the line style parameter to *s* and returns its previous value.
int	*W*.set_node_width(*int pix*)	sets the node width parameter to *pix* pixels and returns its previous value.
text_mode	*W*.set_text_mode(*text_mode m*)	sets the text mode parameter to *m* and returns its previous value.
drawing_mode	*W*.set_mode(*drawing_mode m*)	sets the drawing mode parameter to *m* and returns its previous value.
void	*W*.set_frame_label(*string s*)	makes *s* the window frame label.
void	*W*.reset_frame_label()	restores the standard LEDA frame label.
void	*W*.set_redraw(*void* (*$*F$*)())	sets the redraw function parameter to *F*.
void	*W*.set_flush(*bool b*)	flushes $X11$ output stream after each draw action iff $b = true$.
bool	*W*.load_text_font(*string fn*)	loads $X11$ font *fn* and uses it as text font. Returns true on success and false if the font is not available.
bool	*W*.load_bold_font(*string fn*)	loads $X11$ font *fn* and uses it as bold font. Returns true on success and false if the font is not available.
bool	*W*.load_message_font(*string fn*)	load $X11$ font *fn* and use it as message font. Returns true on success and false if the font is not available.

3.3 Reading parameters and window coordinates

int	*W*.get_line_width()	returns the current line width.
line_style	*W*.get_line_style()	returns the current line style.
int	*W*.get_node_width()	returns the current node width.
text_mode	*W*.get_text_mode()	returns the current text mode.
drawing_mode	*W*.get_mode()	returns the current drawing mode.
double	*W*.xmin()	returns x_0, the minimal x-coordinate of *W*.
double	*W*.ymin()	returns y_0, the minimal y-coordinate of *W*.
double	*W*.xmax()	returns x_1, the maximal x-coordinate of *W*.
double	*W*.ymax()	returns y_1, the maximal y-coordinate of *W*.
double	*W*.scale()	returns the number of pixels of a unit length line segment.

3.4 Drawing points

void *W*.draw_point(*double x*, *double y*, *color c = FG_color*)

draws the point (x, y) as a cross of a vertical and a horizontal segment intersecting at (x, y).

void *W*.draw_point(*point p*, *color c = FG_color*)

draws point (*p*.xcoord(),*p*.ycoord()).

void *W*.draw_pix(*double x*, *double y*, *color c = FG_color*)

sets the color of the pixel at position (x, y) to *c*.

void *W*.draw_pix(*point p*, *color c = FG_color*)

sets the color of the pixel at position *p* to *c*.

3.5 Drawing line segments

void *W*.draw_segment(*double x*1, *double y*1, *double x*2, *double y*2, *color c = FG_color*)

draws a line segment from (x_1, y_1) to (x_2, y_2).

void *W*.draw_segment(*point p*, *point q*, *color c = FG_color*)

draws a line segment from point *p* to point *q*.

void *W*.draw_segment(*segment s*, *color c = FG_color*) draws line segment s.

3.6 Drawing lines

void W.draw_line(*double* $x1$, *double* $y1$, *double* $x2$, *double* $y2$, *color* $c = FG\_color$)

draws a straight line passing through points (x_1, y_1) and (x_2, y_2).

void W.draw_line(*point* p, *point* q, *color* $c = FG\_color$)

draws a straight line passing through points p and q.

void W.draw_line(*segment* s, *color* $c = FG\_color$)

draws the line supporting s.

void W.draw_line(*line* l, *color* $c = FG\_color$)

draws line l.

void W.draw_hline(*double* y, *color* $c = FG\_color$)

draws a horizontal line with y-coordinate y.

void W.draw_vline(*double* x, *color* $c = FG\_color$)

draws a vertical line with x-coordinate x.

void W.draw_arc(*point* p, *point* q, *double* r , *color* $c = FG\_color$)

draws a circular arc with radius r from p to q with the center lying to the right of the directed segment $p \longrightarrow q$.

3.7 Drawing arrows

void W.draw_arrow(*double* $x1$, *double* $y1$, *double* $x2$, *double* $y2$, *color* $c = FG\_color$)

draws an arrow pointing from (x_1, y_1) to (x_2, y_2).

void W.draw_arrow(*point* p, *point* q, *color* $c = FG\_color$)

draws an arrow pointing from point p to point q.

void W.draw_arrow(*segment* s, *color* $= FG\_color$)

draws an arrow pointing from s.start() to s.end().

void W.draw_arc_arrow(*point* p , *point* q, *double* r, *color* $c = FG\_color$)

draws a circular arc arrow with radius r pointing from p to q with the center lying to the right of the directed segment $p \longrightarrow q$.

3.8 Drawing circles

void W.draw_circle(*double* x, *double* y, *double* r, *color* $c = FG\_color$)

draws the circle with center (x, y) and radius r.

void W.draw_circle(*point* p, *double* r, *color* $c = FG\_color$)

draws the circle with center p and radius r.

void W.draw_circle(*circle* C, *color* $c = FG\_color$)

draws circle C.

void W.draw_ellipse(*double* x, *double* y, *double* $r1$, *double* $r2$, *color* $c = FG\_color$)

draws the ellipse with center (x, y) and radii $r1$ and $r2$.

void W.draw_ellipse(*point* p, *double* $r1$, *double* $r2$, *color* $c = FG\_color$)

draws the ellipse with center p and radii $r1$ and $r2$.

3.9 Drawing discs

void W.draw_disc(*double* x, *double* y, *double* r, *color* $c = FG\_color$)

draws a filled circle with center (x, y) and radius r.

void W.draw_disc(*point* p, *double* r, *color* $c = FG\_color$)

draws a filled circle with center p and radius r.

void W.draw_disc(*circle* C, *color* $c = FG\_color$)

draws filled circle C.

void W.draw_filled_ellipse(*double* x, *double* y, *double* $r1$, *double* $r2$, *color* $c = FG\_color$)

draws a filled ellipse with center (x, y) and radii $r1$ and $r2$.

void W.draw_filled_ellipse(*point* p, *double* $r1$, *double* $r2$, *color* $c = FG\_color$)

draws a filled ellipse with center p and radii $r1$ and $r2$.

3.10 Drawing polygons

void W.draw_polygon(*list<point>* lp, *color* $c = FG\_color$)

draws the polygon with vertex sequence lp.

void W.draw_polygon(*polygon* P, *color* $c = FG\_color$)

draws polygon P.

void W.draw_filled_polygon(*list<point>* lp, *color* $c = FG\_color$)

draws the filled polygon with vertex sequence lp.

void W.draw_filled_polygon(*polygon* P, *color* $c = FG\_color$)

draws filled polygon P.

3.11 Drawing functions

void W.plot_xy(*double* $x0$, *double* $x1$, *draw_func_ptr* F, *color* $c = FG\_color$)

draws function F in range $[x_0, x_1]$, i.e., all points (x, y) with $y = F(x)$ and $x_0 \leq x \leq x_1$.

void W.plot_yx(*double* $y0$, *double* $y1$, *draw_func_ptr* F, *color* $c = FG\_color$)

draws function F in range $[y_0, y_1]$, i.e., all points (x, y) with $x = F(y)$ and $y_0 \leq y \leq y_1$.

3.12 Drawing text

void W.draw_text(*double* x, *double* y, *string* s, *color* $c = FG\_color$)

writes string s starting at position (x, y).

void W.draw_text(*point* p, *string* s, *color* $c = FG\_color$)

writes string s starting at position p.

void W.draw_ctext(*double* x, *double* y, *string* s, *color* $c = FG\_color$)

writes string s centered at position (x, y).

void W.draw_ctext(*point* p, *string* s, *color* $c = FG\_color$)

writes string s centered at position p.

3.13 Drawing nodes

void W.draw_node(*double* $x0$, *double* $y0$, *color* $c = FG\_color$)

draws a node at position (x_0, y_0).

void W.draw_node(*point* p, *color* $c = FG\_color$)

draws a node at position p.

void W.draw_filled_node(*double* $x0$, *double* $y0$, *color* $c = FG\_color$)

draws a filled node at position (x_0, y_0).

void W.draw_filled_node(*point* p, *color* $c = FG\_color$)

draws a filled node at position p.

void W.draw_text_node(*double* x, *double* y, *string* s, *color* $c = BG\_color$)

draws a node with label s at position (x, y).

void W.draw_text_node(*point* p, *string* s, *color* $c = BG\_color$)

draws a node with label s at position p.

void W.draw_int_node(*double* x, *double* y, *int* i, *color* $c = BG\_color$)

draws a node with integer label i at position (x, y).

void W.draw_int_node(*point* p, *int* i, *color* $c = BG\_color$)

draws a node with integer label i at position p.

3.14 Drawing edges

void W.draw_edge(*double* $x1$, *double* $y1$, *double* $x2$, *double* $y2$, *color* c = *FG_color*)
draws an edge from (x_1, y_1) to (x_2, y_2).

void W.draw_edge(*point* p, *point* q, *color* c = *FG_color*)
draws an edge from p to q.

void W.draw_edge(*segment* s, *color* c = *FG_color*)
draws an edge from s.start() to s.end().

void W.draw_edge_arrow(*double* $x1$, *double* $y1$, *double* $x2$, *double* $y2$, *color* c = *FG_color*)
draws a directed edge from (x_1, y_1) to (x_2, y_2).

void W.draw_edge_arrow(*point* p, *point* q, *color* c = *FG_color*)
draws a directed edge from p to q.

void W.draw_edge_arrow(*segment* s, *color* c = *FG_color*)
draws a directed edge from s.start() to s.end().

void W.draw_arc_edge(*point* p, *point* q, *double* r, *color* c = *FG_color*)
draws a circular edge arc with radius r from p to q with the center lying to the right of the directed segment $p \longrightarrow q$.

void W.draw_arc_edge_arrow(*point* p, *point* q, *double* r, *color* c = *FG_color*)
draws a circular directed edge arc with radius r from p to q with the center lying to the right of the directed segment $p \longrightarrow q$.

3.15 Mouse Input

int W.read_mouse()
displays the mouse cursor until a button is pressed. Returns integer 1 for the left, 2 for the middle, and 3 for the right button (-1,-2,-3, if the shift key is pressed simultaneously).

int W.read_mouse(*double&* x, *double&* y)
displays the mouse cursor on the screen until a button is pressed. When a button is pressed the current position of the cursor is assigned to (x, y) and the pressed button is returned.

int W.read_mouse(*point&* p)
displays the mouse cursor on the screen until a button is pressed. When a button is pressed the current position of the cursor is assigned to p and the pressed button is returned.

int *W*.read_mouse_seg(*double* $x0$, *double* $y0$, *double&* x, *double&* y)

displays a line segment from (x_0, y_0) to the current cursor position until a mouse button is pressed. When a button is pressed the current position is assigned to (x, y) and the pressed button is returned.

int *W*.read_mouse_seg(*point* p, *point&* q)

displays a line segment from p to the current cursor position until a mouse button is pressed. When a button is pressed the current position is assigned to q and the pressed button is returned.

int *W*.read_mouse_rect(*double* $x0$, *double* $y0$, *double&* x, *double&* y)

displays a rectangle with diagonal from (x_0, y_0) to the current cursor position until a mouse button is pressed. When a button is pressed the current position is assigned to (x, y) and the pressed button is returned.

int *W*.read_mouse_rect(*point* p, *point&* q)

displays a rectangle with diagonal from p to the current cursor position until a mouse button is pressed. When a button is pressed the current position is assigned to q and the pressed button is returned.

int *W*.read_mouse_circle(*double* $x0$, *double* $y0$, *double&* x, *double&* y)

displays a circle with center (x_0, y_0) passing through the current cursor position until a mouse button is pressed. When a button is pressed the current position is assigned to (x, y) and the pressed button is returned.

int *W*.read_mouse_circle(*point* p, *point&* q)

displays a circle with center p passing through the current cursor position until a mouse button is pressed. When a button is pressed the current position is assigned to q and the pressed button is returned.

int *W*.get_button() non-blocking read operation, i.e., if a button was pressed its number is returned, otherwise 0 is returned.

int *W*.get_button(*double&* x, *double&* y)

if a button was pressed the corresponding position is assigned to (x, y) and the button number is returned, otherwise 0 is returned.

int *W*.get_button(*point&* p) if a button was pressed the corresponding position is assigned to p and the button number is returned, otherwise 0 is returned.

3.16 Events

unsigned	*W*.button_press_time()	returns $X11$ time-stamp of last button press event.
unsigned	*W*.button_release_time()	returns $X11$ time-stamp of last button release event.

3.17 Panel Input

int	*W*.confirm(*string s*)	displays string *s* and asks for confirmation. Returns true iff the answer was "yes".
void	*W*.acknowledge(*string s*)	displays string *s* and asks for acknowledgement.
int	*W*.read_panel(*string h*, *int n*, *string**)	displays a panel with header *h* and an array $S[1..n]$ of *n* string buttons, returns the index of the selected button.
int	*W*.read_vpanel(*string h*, *int n*, *string**)	like read_panel with vertical button layout.
string	*W*.read_string(*string p*)	displays a panel with prompt *p* for string input, returns the input.
double	*W*.read_real(*string p*)	displays a panel with prompt *p* for real input returns the input.
int	*W*.read_int(*string p*)	displays a panel with prompt *p* for integer input, returns the input.
void	*W*.message(*string s*)	displays message *s* (each call adds a new line).
void	*W*.del_messages()	deletes the text written by all previous message operations.

3.18 Input and output operators

For input and output of basic geometric objects in the plane such as points, lines, line segments, circles, and polygons the << and >> operators can be used. Similar to C++ input streams windows have an internal state indicating whether there is more input to read or not. Its initial value is true and it is turned to false if an input sequence is terminated by clicking the right mouse button (similar to ending stream input by the eof character). In conditional statements objects of type *window* are automatically converted to boolean by returning this internal state. Thus, they can be used in conditional statements in the same way as C++ input streams. For example, to read a sequence of points terminated by a right button click, use " **while** (W >> p) { ... } ".

3.18.1 Output

window&	*W* << *point p*	like *W*.draw_point(*p*).

window&	W	<<	*segment s*	like W.draw_segment(s).
window&	W	<<	*line l*	like W.draw_line(l).
window&	W	<<	*circle C*	like W.draw_circle(C).
window&	W	<<	*polygon P*	like W.draw_polygon(P).

3.18.2 Input

window&	W	>>	*point& p*	reads a point p: clicking the left button assigns the current cursor position to p.
window&	W	>>	*segment& s*	reads a segment s: use the left button to input the start and end point of s.
window&	W	>>	*line& l*	reads a line l: use the left button to input two different points on l.
window&	W	>>	*circle& C*	reads a circle C: use the left button to input the center of C and a point on C.
window&	W	>>	*polygon& P*	reads a polygon P: use the left button to input the sequence of vertices of P, end the sequence by clicking the middle button.

As long as an input operation has not been completed the last read point can be erased by simultaneously pressing the shift key and the left mouse button.

3.19 Non-Member Functions

int	read_mouse(*window* ∗ &*w*, *double*& *x*, *double*& *y*)	waits for mouse input, assigns a pointer to the corresonding window to w and the position in $*w$ to (x, y) and returns the number of the pressed button.
void	put_back_event()	puts last read event back to the input stream of events.

10.2 Panels (panel)

1. Definition

Panels are windows used for displaying text messages and updating the values of variables. A panel P consists of a set of panel items and a set of buttons. A variable of a certain type (int, bool, string, double, color) is associated with each item (except for text items). It can be manipulated through the item and a string label.

2. Creation

panel P;

creates an empty panel P.

panel $P(string\ s)$;

creates an empty panel P with header s.

panel $P(string\ s,\ int\ w,\ int\ h)$;

creates an empty panel P of width w and height h with header s.

3. Operations

void	P.set_bg_color(*color bg_col*)	sets the panel background color to *bg_col*.
void	P.buttons_per_line(*int n*)	defines the maximal number n of buttons per line.
void	P.label(*string s*)	sets the panel label to s.
void	P.text_item(*string s*)	adds a text_item s to P.
void	P.bool_item(*string s, bool& x*)	adds a boolean item with label s and variable x to P.
void	P.real_item(*string s, double& x*)	adds a real item with label s and variable x to P.
void	P.color_item(*string s, color& x*)	adds a color item with label s and variable x to P.
void	P.lstyle_item(*string s, line_style& x*)	adds a line style item with label s and variable x to P.
void	P.int_item(*string s, int& x*)	adds an integer item with label s and variable x to P.

void	*P*.int_item(*string s*, *int& x*, *int l*, *int h*, *int step*)	adds an integer choice item with label *s*, variable *x*, range *l*,..., *h*, and step size *step* to *P*.
void	*P*.int_item(*string s*, *int& x*, *int l*, *int h*)	adds an integer slider item with label *s*, variable *x*, and range *l*,...,*h* to *P*.
void	*P*.string_item(*string s*, *string& x*)	adds a string item with label *s* and variable *x* to *P*.
void	*P*.string_item(*string s*, *string& x*, *list<string>& L*)	adds a string item with label *s*, variable *x*, and menu *L* to *P*.
void	*P*.choice_item(*string s*, *int& x*, *list<string>& L*)	adds an integer item with label *s*, variable *x*, and choices from *L* to *P*.
void	*P*.choice_item(*string s*, *int& x*, *string s1*, ..., *string sk*)	adds an integer item with label *s*, variable *x*, and choices s_1, ..., s_k to *P* ($k \leq 4$).
int	*P*.button(*string s*)	adds a button with label *s* to *P* and returns its number.
void	*P*.new_button_line()	starts a new line of buttons.
void	*P*.display()	displays *P* centered on screen.
void	*P*.display(*int x*, *int y*)	displays *P* with left upper corner at (x, y).
void	*P*.display(*window& W*)	displays *P* centered over window *W*.
void	*P*.display(*window& W*, *int x*, *int y*)	displays *P* with left upper corner at position (x, y) of window *W*.
int	*P*.read()	waits for a button selection in *P*. Returns the number of the selected button.
int	*P*.open()	*P*.display() + *P*.read().
int	*P*.open(*int x*, *int y*)	*P*.display(x, y) + *P*.read().
int	*P*.open(*window& W*)	*P*.display(W) + *P*.read().
int	*P*.open(*window& W*, *int x*, *int y*)	*P*.display(W, x, y) + *P*.read().

Chapter 11

Miscellaneous

This section describes some additional useful data types, functions and macros of LEDA. The stream data types described in this section are all derived from the C++ stream types *istream* and *ostream*. They can be used in any program that includes the <LEDA/stream.h> header file. Some of these types may be obsolete in combination with the latest versions of the standard C++ I/O library.

11.1 File Input Streams (file_istream)

1. Definition

An instance I of the data type *file_istream* is an C++ istream connected to a file F, i.e., all input operations or operators applied to I read from F.

2. Creation

file_istream $I(string\ s)$;

creates an instance I of type file_istream connected to file s.

3. Operations

All operations and operators (>>) defined for C++ istreams can be applied to file input streams as well.

11.2 File Output Streams (file_ostream)

1. Definition

An instance O of the data type *file_ostream* is an C++ ostream connected to a file F, i.e., all output operations or operators applied to O write to F.

2. Creation

file_ostream $O(char * \ s)$;

creates an instance O of type file_ostream connected to file s.

3. Operations

All operations and operators (<<) defined for C++ ostreams can be applied to file output streams as well.

11.3 String Input Streams (string_istream)

1. Definition

An instance I of the data type *string_istream* is an C++ istream connected to a string s, i.e., all input operations or operators applied to I read from s.

2. Creation

string_istream $I(string \ s)$;

creates an instance I of type string_istream connected to the string s.

3. Operations

All operations and operators (>>) defined for C++ istreams can be applied to string input streams as well.

11.4 String Output Streams (string_ostream)

1. Definition

An instance O of the data type *string_ostream* is an C++ ostream connected to an internal string buffer, i.e., all output operations or operators applied to O write into this internal buffer. The current value of the buffer is called the contents of O.

2. Creation

string_ostream O;

creates an instance O of type string_ostream.

3. Operations

string	O.str()	returns the current contents of O.

All operations and operators (<<) defined for C++ ostreams can be applied to string output streams as well.

11.5 Command Input Streams (cmd_istream)

1. Definition

An instance *I* of the data type *cmd_istream* is an C++ istream connected to the output of a shell command *cmd*, i.e., all input operations or operators applied to *I* read from the standard output of command *cmd*.

2. Creation

cmd_istream *I*(*string cmd*);

creates an instance *I* of type cmd_istream connected to the output of command *cmd*.

3. Operations

All operations and operators (>>) defined for C++ istreams can be applied to command input streams as well.

11.6 Command Output Streams (cmd_ostream)

1. Definition

An instance *O* of the data type *cmd_ostream* is an C++ ostream connected to the input of a shell command *cmd*, i.e., all output operations or operators applied to *O* write into the standard input of command *cmd*.

2. Creation

cmd_ostream *O*(*string cmd*);

creates an instance *O* of type cmd_ostream connected to the input of command *cmd*.

3. Operations

All operations and operators (<<) defined for C++ ostreams can be applied to command output streams as well.

11.7 Some Useful Functions (misc)

The following functions and macros are defined in <LEDA/basic.h>.

int	read_int(*string s*)	prints *s* and reads an integer from *cin*.
double	read_real(*string s*)	prints *s* and reads a real number from *cin*.
string	read_string(*string s*)	prints *s* and reads a line from *cin*.
char	read_char(*string s*)	prints *s* and reads a character from *cin*.
int	Yes(*string s*)	returns (read_char(*s*) == 'y').
float	used_time()	returns the currently used cpu time in seconds.
float	used_time(*float& T*)	returns the cpu time used by the program from time *T* up to this moment and assigns the current time to *T*.
void	wait(*float sec*)	suspends execution for *sec* seconds.
void	print_statistics()	prints a summary of the currently used memory.
T	Max(*T a*, *T b*)	returns the maximum of *a* and *b*.
T	Min(*T a*, *T b*)	returns the minimum of *a* and *b*.

11.8 Memory Management

LEDA offers an efficient memory management system that is used internally for all node, edge and item types. This system can easily be customized for user defined classes by the "LEDA_MEMORY" macro. You simply have to add the macro call "LEDA_MEMORY(*T*)" to the declaration of a class *T*. This redefines new and delete operators for type *T*, such that they allocate and deallocate memory using LEDA's internal memory manager.

Attention: There is a restriction on the size of the type *T*, however. Macro LEDA_MEMORY may only be applied to types *T* with **sizeof(T) < 256**. Note that this condition is (for efficiency reasons) not checked.

We continue the example from section 1.5:

```
struct pair {
  double x;
  double y;

  pair(){ x = y = 0; }
  pair(const pair& p) { x = p.x; y = p.y; }

  friend ostream& operator<<(ostream&,const pair&)      { ...}
  friend istream& operator>>(istream&,pair&)            { ...}
  friend int      compare(const pair& p, const pair& q) { ...}

  LEDA_MEMORY(pair)

};

dictionary<pair,int> D;
```

11.9 Error Handling

LEDA tests the preconditions of many (not all!) operations. Preconditions are never tested, if the test takes more than constant time. If the test of a precondition fails an error handling routine is called. It takes an integer error number i and a $char*$ error message string s as arguments. It writes s to the diagnostic output (cerr) and terminates the program abnormally if $i \neq 0$. Users can provide their own error handling function *handler* by calling set_error_handler(*handler*). After this function call *handler* is used instead of the default error handler. *handler* must be a function of type *void handler*(*int*, *char∗*). The parameters are replaced by the error number and the error message respectively.

Chapter 12

Programs

12.1 Graph and network algorithms

In this section we list the C++ sources for some of the graph algorithms in the library (cf. section 5.12).

Depth First Search

```
#include <LEDA/graph.h>
#include <LEDA/stack.h>
list<node> DFS(graph& G, node v, node_array<bool>& reached)
{
   list<node> L;
   stack<node> S;
   node w;

   if ( !reached[v] )
     { reached[v] = true;
       S.push(v);
      }

   while ( !S.empty() )
         { v = S.pop();
           L.append(v);
           forall_adj_nodes(w, v)
             if ( !reached[w] )
               { reached[w] = true;
                 S.push(w);
               }

         }

   return L;
}
```

Breadth First Search

```
#include <LEDA/graph.h>
#include <LEDA/queue.h>
void BFS(graph& G, node v, node_array<int>& dist)
{
   queue<node> Q;
   node w;

   forall_nodes(w, G) dist[w] = -1;

   dist[v] = 0;
   Q.append(v);

   while ( !Q.empty() )
         { v = Q.pop();
           forall_adj_nodes(w, v)
              if (dist[w] < 0)
                { Q.append(w);
                  dist[w] = dist[v] + 1;
                }
         }
}
```

Connected Components

```
#include <LEDA/graph.h>
int COMPONENTS(ugraph& G, node_array < int > & compnum)
{
   node v, w;
   list<node> S;
   int count = 0;

   node_array(bool) reached(G, false);

   forall_nodes (v, G)
      if ( !reached[v] )
        { S = DFS(G, v, reached);
          forall (w, S) compnum[w] = count;
          count + +;
        }

   return count;
}
```

Depth First Search Numbering

```
#include <LEDA/graph.h>
int dfs_count1, dfs_count2;
void d_f_s(node v,node_array<bool>& S, node_array<int>& dfsnum,
    node_array<int>& compnum, list<edge> T )
{ // recursive DFS

  node w;
  edge e;

  S[v] = true;
  dfsnum[v] = ++dfs_count1;

  forall_adj_edges (e, v)
    { w = G.target(e);
     if ( !S[w] )
      { T.append(e);
        d_f_s(w, S, dfsnum, compnum, T);
      }
    }

  compnum[v] = ++dfs_count2;
}


list<edge>
DFS_NUM(graph& G, node_array<int>& dfsnum, node_array<int>& compnum
)
{
  list<edge> T;
  node_array<bool> reached(G, false);
  node v;
  dfs_count1 = dfs_count2 = 0;
  forall_nodes (v, G)
    if ( !reached[v] ) d_f_s(v, reached, dfsnum, compnum, T);
  return T;
}
```

Topological Sorting

```
#include <LEDA/graph.h>
bool TOPSORT(graph& G, node_array<int>&ord)
{
   node_array<int> INDEG(G);
   list<node> ZEROINDEG;

   int count = 0;
   node v, w;
   edge e;

   forall_nodes(v, G)
      if ((INDEG[v]=G.indeg(v))==0) ZEROINDEG.append(v);

   while (!ZEROINDEG.empty())
      { v = ZEROINDEG.pop();
       ord[v] = ++count;

       forall_adj_nodes(w, v)
          if (--INDEG[w]==0) ZEROINDEG.append(w);
      }

   return (count==G.number_of_nodes());
}
```

```
//TOPSORT1 sorts node and edge lists according to the topological ordering:

bool TOPSORT1(graph& G)
{
   node_array<int> node_ord(G);
   edge_array<int> edge_ord(G);

   if (TOPSORT(G,node_ord))
     { edge e;
       forall_edges(e, G) edge_ord[e]=node_ord[target(e)];
       G.sort_nodes(node_ord);
       G.sort_edges(edge_ord);
       return true;
     }
   return false;
}
```

Strongly Connected Components

```
#include <LEDA/graph.h>
#include <LEDA/array.h>

int STRONG_COMPONENTS(graph& G, node_array<int>& compnum)
{
  node v, w;
  int n = G.number_of_nodes();
  int count = 0;
  int i;

  array<node> V(1, n);
  list<node> S;
  node_array<int> dfs_num(G), compl_num(G);
  node_array<bool> reached(G, false);

  DFS_NUM(G, dfs_num, compl_num);

  forall_nodes (v, G) V[compl_num[v]] = v;

  G.rev();

  for (i = n; i > 0; i - -)
    if ( !reached[V[i]] )
     { S = DFS(G, V[i], reached);
       Forall(w, S) compnum[w] = count;
       count + +;
     }

  return count;

}
```

Dijkstra's Algorithm

```
#include <LEDA/graph.h>
#include <LEDA/node_pq.h>

void DIJKSTRA( graph& G, node s, edge_array<int>& cost,
    node_array<int>& dist, node_array<edge>& pred )
{
  node_pq<int> PQ(G);

  int c;
  node u, v;
  edge e;

  forall_nodes(v, G)
    { pred[v] = 0;
      dist[v] = infinity;
      PQ.insert(v, dist[v]);
    }

  dist[s] = 0;
  PQ.decrease_inf(s, 0);

  while ( ! PQ.empty())
    { u = PQ.del_min();

      forall_adj_edges(e, u)
        { v = G.target(e);
          c = dist[u] + cost[e];
          if ( c < dist[v])
            { dist[v] = c;
              pred[v] = e;
              PQ.decrease_inf(v, c);
            }
        } /* forall_adj_edges *

    } /* while */

}
```

Bellman/Ford Algorithm

```
#include <LEDA/graph.h>
#include <LEDA/queue.h>

bool BELLMAN_FORD(graph& G, node s, edge_array<int>& cost,
     node_array<int>& dist, node_array<edge>& pred)
{
  node_array<bool> in_Q(G, false);
  node_array<int> count(G, 0);

  int n = G.number_of_nodes();
  queue<node> Q(n);

  node u, v;
  edge e;
  int c;

  forall_nodes (v, G)
     { pred[v] = 0;
       dist[v] = infinity;
     }
  dist[s] = 0;
  Q.append(s);
  in_Q[s] = true;

  while (!Q.empty())
     { u = Q.pop();
       in_Q[u] = false;

       if (++count[u] > n) return false;   //negative cycle

       forall_adj_edges (e, u)
          { v = G.target(e);
            c = dist[u] + cost[e];

            if (c < dist[v])
              { dist[v] = c;
                pred[v] = e;
                if (!in_Q[v])
                  { Q.append(v);
                    in_Q[v] = true;
                  }
              }
          } /* forall_adj_edges */
     } /* while */
  return true;
}
```

All Pairs Shortest Paths

```
#include <LEDA/graph.h>

void all_pairs_shortest_paths(graph& G, edge_array<double>& cost,
     node_matrix<double>& DIST)
{
   // computes for every node pair (v, w) DIST(v, w) = cost of the least cost
   // path from v to w, the single source shortest paths algorithms BELLMAN_FORD
   // and DIJKSTRA are used as subroutines

   edge e;
   node v;
   double C = 0;

   forall_edges(e, G) C+ = fabs(cost[e]);
   node s = G.new_node();                     // add s to G
   forall_nodes(v, G) G.new_edge(s, v);        // add edges (s, v) to G

   node_array<double> dist1(G);
   node_array<edge> pred(G);
   edge_array<double> cost1(G);
   forall_edges(e, G) cost1[e] = (G.source(e) == s) ? C : cost[e];

   BELLMAN_FORD(G, s, cost1, dist1, pred);

   G.del_node(s);                             // delete s from G
   edge_array(double) cost2(G);
   forall_edges(e, G) cost2[e] = dist1[G.source(e)] + cost[e] − dist1[G.target(e)];

   forall_nodes(v, G) DIJKSTRA(G, v, cost2, DIST[v], pred);

   forall_nodes(v, G)
      forall_nodes(w, G) DIST(v, w) = DIST(v, w) − dist1[v] + dist1[w];
}
```

Minimum Spanning Tree

```
#include <LEDA/graph.h>
#include <LEDA/node_partition.h>

void MIN_SPANNING_TREE(graph& G, edge_array<double>& cost, list<edge>& EL)
{
   node v, w;
   edge e;
   node_partition Q(G);

   G.sort_edges(cost);

   EL.clear();
   forall_edges(e, G)
     { v = G.source(e);
       w = G.target(e);
       if (!(Q.same_block(v, w))
         { Q.union_blocks(v, w);
           EL.append(e);
         }
     }
}
```

12.2 Geometry

Using a persistent dictionary (cf. section 4.7) for planar point location (sweep line algorithm).

```
#include <LEDA/plane.h>
#include <LEDA/prio.h>
#include <LEDA/sortseq.h>
#include <LEDA/p_dictionary.h>


double X_POS; // current position of sweep line

int compare(segment s1,segment s2)
{
  line l1(s1);
  line l2(s2);

  double y1 = l1.y_proj(X_POS);
  double y2 = l2.y_proj(X_POS);

  return compare(y1, y2);
}
typedef priority_queue<segment,point> X_structure;
typedef p_dictionary<segment,int> Y_structure;

sortseq<double,Y_structure> HISTORY;

void SWEEP(list<segment>& L)
{
  // Precondition: L is a list of non-intersecting
  // from left to right directed line segments

  X_structure X;
  Y_structure Y;
  segment s;

  forall (s,L)                      // initialize the X_structure
    { X.insert(s, s.start());
      X.insert(s,s.end());
    }

  HISTORY.insert(-MAXDOUBLE,Y); // insert empty Y_structure at -infinity

  while ( ! X.empty() )
    { point p;
      segment s;
      X.del_min(s,p);               // next event: endpoint p of segment s
      X_POS = p.xcoord();

      if (s.start()== p)
        Y = Y.insert(s,0);          // p is left end of s

      else
        Y = Y.del(s);               // p is right end of s
        HISTORY.insert(X_POS, Y);    // insert Y into history sequence
    }
    HISTORY.insert(MAXDOUBLE,Y); // insert empty Y_structure at +infinity
}
segment LOCATE(point p)
{
    X_POS = p.xcoord();
    Y_structure Y = HISTORY.inf(HISTORY.pred(X_POS));
    p_dic_item pit = Y.succ(segment(p, 0, 1));
    if (pit! = nil)
      return Y.key(pit);
    else
      return segment(0);
}
```

Chapter 13

Implementations

13.1 List of data structures

This section lists the data structures for dictionaries, dictionary arrays, priority queues, and geometric data types currently contained in LEDA. For each of the data structures its name and type, the list of LEDA data types it can implement, and a literature reference are given. Before using a data structures *xyz* the corresponding header file <LEDA/impl/*xyz*.h> has to be included (cf. section 1.2 for an example).

13.1.1 Dictionaries

ab_tree	a-b tree	dictionary, d_array, **sortseq**	[7]
avl_tree	AVL tree	dictionary, d_array	[3]
bb_tree	BB[α] tree	dictionary, d_array, sortseq	[8]
ch_hashing	hashing with chaining	dictionary, d_array	[34]
dp_hashing	dyn. perf. hashing	**h_array**	[14], [49]
pers_tree	persistent tree	**p_dictionary**	[15]
rb_tree	red-black tree	dictionary, d_array, sortseq	[25]
rs_tree	rand. search tree	**dictionary**, **d_array**, sortseq	[1]
skiplist	skip lists	dictionary, d_array, sortseq	[43]

13.1.2 Priority Queues

f_heap	Fibonnacci heap	**priority_queue**	[22]
p_heap	pairing heap	priority_queue	[46]
k_heap	k-nary heap	priority_queue	[34]
m_heap	monotonic heap	priority_queue	[34]
eb_tree	Emde-Boas tree	priority_queue	[18], [49]

13.1.3 Geometry

range_tree	range tree	**d2_dictionary**, **point_set**	[50], [33]
seg_tree	segment tree	**seg_set**	[4], [17]
ps_tree	priority search tree	—	[36]
iv_tree	interval tree	**interval_set**	[35], [17]
delaunay_tree	delaunay tree	**point_set**	[12]

13.2 User Implementations

In addition to the data structures listed in the previous section user-defined data structures can also be used as actual implementation parameters provided they fulfill certain requirements.

13.2.1 Dictionaries

Any class *dic_impl* that provides the following operations can be used as actual implementation parameter for the *_dictionary<K, I, dic_impl>* and the *_d_array<I, E, dic_impl>* data types (cf. sections 5.1 and 5.5).

```
typedef ... dic_impl_item;

class dic_impl {

   virtual int cmp(GenPtr, GenPtr) const = 0;
   virtual int int_type()          const = 0;
   virtual void clear_key(GenPtr&)  const = 0;
   virtual void clear_inf(GenPtr&)  const = 0;
   virtual void copy_key(GenPtr&)   const = 0;
   virtual void copy_inf(GenPtr&)   const = 0;

public:

   dic_impl();
   dic_impl(const dic_impl&);
   virtual  dic_impl();

   dic_impl& operator=(const dic_impl&);

   GenPtr key(dic_impl_item) const;
   GenPtr inf(dic_impl_item)  const;

   dic_impl_item insert(GenPtr,GenPtr);
   dic_impl_item lookup(GenPtr) const;
   dic_impl_item first_item()     const;
   dic_impl_item next_item(dic_impl_item) const;

   dic_impl_item item(void* p) const { return dic_impl_item(p); }

   void change_inf(dic_impl_item,GenPtr);
   void del_item(dic_impl_item);
   void del(GenPtr);
   void clear();

   int size() const;
};
```

13.2.2 Priority Queues

Any class *prio_impl* that provides the following operations can be used as actual implementation parameter for the *_priority_queue<K, I, prio_impl>* data type (cf. section 6.1).

```
typedef ... prio_impl_item;

class prio_impl

   virtual int cmp(GenPtr, GenPtr) const = 0;
   virtual int int_type()          const = 0;
   virtual void clear_key(GenPtr&)  const = 0;
   virtual void clear_inf(GenPtr&)  const = 0;
   virtual void copy_key(GenPtr&)  const = 0;
   virtual void copy_inf(GenPtr&)  const = 0;

public:

   prio_impl();
   prio_impl(int);
   prio_impl(int,int);
   prio_impl(const prio_impl&);
   virtual  prio_impl();

   prio_impl& operator=(const prio_impl&);

   prio_impl_item insert(GenPtr,GenPtr);
   prio_impl_item find_min()  const;
   prio_impl_item first_item() const;
   prio_impl_item next_item(prio_impl_item) const;

   prio_impl_item item(void* p) const { return prio_impl_item(p); }

   GenPtr key(prio_impl_item) const;
   GenPtr inf(prio_impl_item)  const;

   void del_min();
   void del_item(prio_impl_item);
   void decrease_key(prio_impl_item,GenPtr);
   void change_inf(prio_impl_item,GenPtr);
   void clear();

   int size() const;
};
```

13.2.3 Sorted Sequences

Any class *seq_impl* that provides the following operations can be used as actual implementation parameter for the *_sortseq<K, I, seq_impl>* data type (cf. section 5.3).

```
typedef ... seq_impl_item;

class seq_impl {

  virtual int cmp(GenPtr, GenPtr) const = 0;
  virtual int int_type()           const = 0;
  virtual void clear_key(GenPtr&)  const = 0;
  virtual void clear_inf(GenPtr&)  const = 0;
  virtual void copy_key(GenPtr&)   const = 0;
  virtual void copy_inf(GenPtr&)   const = 0;

public:

  seq_impl();
  seq_impl(const seq_impl&);
  virtual  seq_impl();

  seq_impl& operator=(const seq_impl&);
  seq_impl& conc(seq_impl&);

  seq_impl_item insert(GenPtr,GenPtr);
  seq_impl_item insert_at_item(seq_impl_item,GenPtr,GenPtr);
  seq_impl_item lookup(GenPtr)      const;
  seq_impl_item locate(GenPtr)      const;
  seq_impl_item locate_pred(GenPtr) const;
  seq_impl_item succ(seq_impl_item) const;
  seq_impl_item pred(seq_impl_item) const;
  seq_impl_item item(void* p) const { return seq_impl_item(p); }

  GenPtr key(seq_impl_item) const;
  GenPtr inf(seq_impl_item)  const;

  void del(GenPtr);
  void del_item(seq_impl_item);
  void change_inf(seq_impl_item,GenPtr);
  void split_at_item(seq_impl_item,seq_impl&,seq_impl&);
  void reverse_items(seq_impl_item,seq_impl_item);
  void clear();

  int size() const;
};
```

Chapter 14

Tables

14.1 Data Types

Name	Item	Header	Library	Page
array	——	array.h	libL.a	39
array2	——	array2.h	libL.a	41
b_node_pq	——	b_node_pq.h	libG.a	101
b_priority_queue	b_pq_item	b_prio.h	libL.a	76
b_queue	——	b_queue.h	libL.a	45
b_stack	——	b_stack.h	libL.a	44
circle	——	circle.h	libP.a	125
cmd_istream	——	stream.h	libL.a	155
cmd_ostream	——	stream.h	libL.a	155
d2_dictionary	d2_dic_item	d2_dictionary.h	libP.a	129
d_array	——	d_array.h	libL.a	64
dictionary	dic_item	dictionary.h	libL.a	57
edge_array	——	edge_array.h	libG.a	92
edge_map	——	edge_map.h	libG.a	94
edge_set	——	edge_set.h	libG.a	97
file_istream	——	stream.h	libL.a	153
file_ostream	——	stream.h	libL.a	153
floatf	——	floatf.h	libL.a	33
graph	node/edge	graph.h	libG.a	77
GRAPH	node/edge	graph.h	libG.a	83
h_array	——	h_array.h	libL.a	67
integer	——	integer.h	libL.a	29
int_set	——	int_set.h	libL.a	54
interval_set	is_item	interval_set.h	libP.a	133
line	——	line.h	libP.a	122
list	list_item	list.h	libL.a	46
map	——	map.h	libL.a	68
matrix	——	matrix.h	libL.a	26
node_array	——	node_array.h	libG.a	91
node_list	——	node_list.h	libG.a	98
node_map	——	node_map.h	libG.a	93

node_matrix	——	graph.h	libG.a	95
node_partition	——	node_partition.h	libG.a	99
node_pq	——	node_pq.h	libG.a	100
node_set	——	node_set.h	libG.a	96
panel	——	panel.h	libP.a/libWx.a	150
partition	partition_item	partition.h	libL.a	55
planar_map	node/edge/face	planar_map.h	libG.a	87
point	——	point.h	libP.a	111
point_set	ps_item	point_set.h	libP.a	131
polygon	——	polygon.h	libP.a	124
p_queue	pq_item	p_queue.h	libL.a	71
p_dictionary	p_dic_item	p_dictionary.h	libL.a	69
PLANAR_MAP	node/edge/face	planar_map.h	libG.a	89
queue	——	queue.h	libL.a	43
rational	——	rational.h	libL.a	31
rat_point	——	rat_point.h	libP.a	114
rat_segment	——	rat_segment.h	libP.a	119
real	——	real.h	libL.a	35
segment	——	segment.h	libP.a	116
segment_set	seg_item	segment_set.h	libP.a	135
set	——	set.h	libL.a	53
sortseq	seq_item	sortseq.h	libL.a	60
stack	——	stack.h	libL.a	42
string	——	string.h	libL.a	19
string_istream	——	stream.h	libL.a	154
string_ostream	——	stream.h	libL.a	154
subdivision	node/face	subdivision.h	libP.a	137
tree_collection	d_vertex	tree_collection.h	libL.a	56
ugraph	node/edge	ugraph.h	libG.a	86
UGRAPH	node/edge	ugraph.h	libG.a	86
vector	——	vector.h	libL.a	24
window	——	window.h	libP.a/libWx.a	139

14.2 Algorithms

Name	Header	Library	Page
ALL_PAIRS_SHORTEST_PATHS	graph_alg.h	libG.a	107
BELLMAN_FORD	graph_alg.h	libG.a	107
BFS	graph_alg.h	libG.a	106
COMPONENTS	graph_alg.h	libG.a	106
CONVEX_HULL	plane_alg.h	libP.a	127
DFS	graph_alg.h	libG.a	106
DFS_NUM	graph_alg.h	libG.a	106
DIJKSTRA	graph_alg.h	libG.a	107
MAX_CARD_MATCHING	graph_alg.h	libG.a	107
MAX_CARD_BIPARTITE_MATCHING	graph_alg.h	libG.a	107
MAX_FLOW	graph_alg.h	libG.a	107
MAX_WEIGHT_BIPARTITE_MATCHING	graph_alg.h	libG.a	107
MIN_CUT	graph_alg.h	libG.a	107
MIN_COST_MAX_FLOW	graph_alg.h	libG.a	107
MIN_SPANNING_TREE	graph_alg.h	libG.a	107
PLANAR	graph_alg.h	libG.a	110
SEGMENT_INTERSECTION	plane_alg.h	libP.a	127
SPANNING_TREE	graph_alg.h	libG.a	107
STRAIGHT_LINE_EMBEDDING	graph_alg.h	libG.a	110
STRONG_COMPONENTS	graph_alg.h	libG.a	106
SWEEP_SEGMENTS	plane_alg.h	libP.a	127
TOPSORT	graph_alg.h	libG.a	106
TRANSITIVE_CLOSURE	graph_alg.h	libG.a	106
TRIANGULATE_PLANAR_MAP	graph_alg.h	libG.a	110
VORONOI	plane_alg.h	libP.a	127

Bibliography

[1] C. Aragon, R. Seidel: "Randomized Search Trees", Proc. 30th IEEE Symposium on Foundations of Computer Science, 540-545, 1989

[2] A.V. Aho, J.E. Hopcroft, J.D. Ullman: "Data Structures and Algorithms", Addison-Wesley Publishing Company, 1983

[3] G.M. Adelson-Veslkii, Y.M. Landis: "An Algorithm for the Organization of Information", Doklady Akademi Nauk, Vol. 146, 263-266, 1962

[4] J.L. Bentley: "Decomposable Searching Problems", Information Processing Letters, Vol. 8, 244-252, 1979

[5] R.E. Bellman: "On a Routing Problem", Quart. Appl. Math. 16, 87-90, 1958

[6] J.L. Bentley, Th. Ottmann: "Algorithms for Reporting and Counting Geometric Intersections", IEEE Trans. on Computers C 28, 643-647, 1979

[7] R. Bayer, E. McCreight: "Organizatino and Maintenance of Large Ordered Indizes", Acta Informatica, Vol. 1, 173-189, 1972

[8] N. Blum, K. Mehlhorn: "On the Average Number of Rebalancing Operations in Weight-Balanced Trees", Theoretical Computer Science 11, 303-320, 1980

[9] Ch. Burnikel, K. Mehlhorn, and St. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and experimental results. In *LNCS*, volume 855, pages 227–239. Springer-Verlag Berlin/New York, 1994. Proceedings of ESA'94.

[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest: "Introduction to Algorithms", MIT Press/McGraw-Hill Book Company, 1990

[11] D. Cheriton, R.E. Tarjan: "Finding Minimum Spanning Trees", SIAM Journal of Computing, Vol. 5, 724-742, 1976

[12] O. Devillers: "Robust and Efficient Implementation of the Delaunay Tree", Technical Report, INRIA, 1992

[13] E.W. Dijkstra: "A Note on Two Problems in Connection With Graphs", Num. Math., Vol. 1, 269-271, 1959

[14] M. Dietzfelbinger, A. Karlin, K.Mehlhorn, F. Meyer auf der Heide, H. Rohnert, R. Tarjan: "Upper and Lower Bounds for the Dictionary Problem", Proc. of the 29th Annual IEEE Symposium on Foundations of Computer Science, 1988

[15] J.R. Driscoll, N.Sarnak, D. Sleator, R.E. Tarjan: "Making Data Structures Persistent", Proc. of the 18th Annual ACM Symposium on Theory of Computing, 109-121, 1986

[16] J. Edmonds: "Paths, Trees, and Flowers", Canad. J. Math., Vol. 17, 449-467, 1965

[17] H. Edelsbrunner: "Intersection Problems in Computational Geometry", Ph.D. thesis, TU Graz, 1982

[18] P.v. Emde Boas, R. Kaas, E. Zijlstra: "Design and Implementation of an Efficient Pririty Queue", Math. Systems Theory, Vol. 10, 99-127, 1977

[19] I. Fary: "On Straight Line Representing of Planar Graphs", Acta. Sci. Math. Vol. 11, 229-233, 1948

[20] F.W. Floyd: "Algorithm 97: Shortest Paths", Communcication of the ACM, Vol. 5, p. 345, 1962

[21] S. Fortune and C. van Wyk. Efficient exact arithmetic for computational geometry. *Proc. of the 9th Symp. on Computational Geometry*, pages 163–171, 1993.

[22] M.L. Fredman, and R.E. Tarjan: "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms", Journal of the ACM, Vol. 34, 596-615, 1987

[23] A. Goralcikova, V. Konbek: "A Reduct and Closure Algorithm for Graphs", Mathematical Foundations of Computer Science, LNCS 74, 301-307, 1979

[24] K.E. Gorlen, S.M. Orlow, P.S. Plexico: "Data Abstraction and Object-Oriented Programming in C++ ", John Wiley & Sons, 1990

[25] L.J. Guibas, R. Sedgewick: " A Dichromatic Framework for Balanced Trees", Proceedings of the 19th IEEE Symposium on Foundations of Computer Science, 8-21, 1978

[26] Goldberg, R.E.Tarjan: "A New Approach to the Maximum Flow Problem", Journal of the ACM, Vol. 35, 921-940, 1988

[27] J.E. Hopcroft, R.M. Karp: "An $O(n^{2.5})$ Algorithm for Matching in Bipartite Graphs", SIAM Journal of Computing, Vol. 4, 225-231, 1975

[28] J.E. Hopcroft, R.E. Tarjan: "Efficient Planarity Testing", Journal of the ACM, Vol. 21, 549-568, 1974

[29] T. Hagerup, C. Uhrig: "Triangulating a Planar Map Without Introducing multiple Arcs", unpublished, 1989

[30] A.B. Kahn: "Topological Sorting of Large Networks", Communications of the ACM, Vol. 5, 558-562, 1962

[31] J.B. Kruskal: "On the Shortest Spanning Subtree of a Graph and the Travelling Salesman Problem", Proc. American Math. Society 7, 48-50, 1956

[32] S.B. Lippman: "C++ Primer", Addison-Wesley, Publishing Company, 1989

[33] G.S. Luecker: "A Data Structure for Orthogonal Range Queries", Proc. 19th IEEE Symposium on Foundations of Computer Science, 28-34, 1978

[34] K. Mehlhorn: "Data Structures and Algorithms", Vol. 1–3, Springer Publishing Company, 1984

[35] D.M. McCreight: "Efficient Algorithms for Enumerating Intersecting Intervals", Xerox Parc Report, CSL-80-09, 1980

[36] D.M. McCreight: "Priority Search Trees", Xerox Parc Report, CSL-81-05, 1981

[37] M. Mignotte. *Mathematics for Computer Algebra.* Springer Verlag, 1992.

[38] K. Mehlhorn, S. Näher: " LEDA, a Library of Efficient Data Types and Algorithms", TR A 04/89, FB10, Universität des Saarlandes, Saarbrücken, 1989

[39] K. Mehlhorn, S. Näher: " LEDA, a Platform for Combinatorial and Geometric Computing", Communications of the ACM, Vol. 38, No. 1, 96-102, 1995

[40] K. Mehlhorn and S. Näher. Implementation of a sweep line algorithm for the straight line segment intersection problem. Technical Report MPI-I-94-160, Max-Planck-Institut für Informatik, Saarbrücken, 1994.

[41] K. Mehlhorn and St. Näher. The implementation of geometric algorithms. In *13th World Computer Congress IFIP94*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994.

[42] S. Näher: "LEDA2.0 User Manual", technischer Bericht A 17/90, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1990

[43] W. Pugh: "Skip Lists: A Probabilistic Alternative to Balanced Trees", Communications of the ACM, Vol. 33, No. 6, 668-676, 1990

[44] M. Stoer and F. Wagner: "A Simple Min Cut Algorithm", Algorithms – ESA '94, LNCS 855, 141- 147, 1994

[45] B. Stroustrup: "The C++ Programming Language, Second Edition", Addison-Wesley Publishing Company, 1991

[46] J.T. Stasko, J.S. Vitter: "Pairing Heaps: Experiments and Analysis", Communications of the ACM, Vol. 30, 234-249, 1987

[47] R.E. Tarjan: "Depth First Search an Linear Graph Algorithms", SIAM Journal of Computing, Vol. 1, 146-160, 1972

[48] R.E. Tarjan: "Data Structures and Network Algorithms", CBMS-NSF Regional Conference Series in Applied Mathematics, Vol. 44, 1983

[49] M. Wenzel: "Wörterbücher für ein beschränktes Universum", Diplomarbeit, Fachbereich Informatik, Universität des Saarlandes, 1992

[50] D.E. Willard: "New Data Structures for Orthogonal Queries", SIAM Journal of Computing, 232-253, 1985