LEDA Manual

Version 3.0

Stefan Näher

MPI-I-93-109 **February 1993**

Max-Planck-Institut für Informatik
Im Stadtwald
D-6600 Saarbrücken

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

Table of Contents

Acknowledgement

The author would like to thank Kurt Mehlhorn for many helpful suggestions and valuable comments.

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 allmost any C++ compiler (e.g., cfront2, cfront3, g++-1, g++-2, bcc, ztc).

7. LEDA is available by anonymous ftp from
ftp.cs.uni-sb.de (134.96.252.31) /pub/LEDA
The Distribution contains all sources, installation instructions, a technical report, and the LEDA user manual.

8. LEDA is not in the public domain, but can be used freely for research and teaching. A commercial license is available from the autor.

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 [S91] or [L89]). The main concepts and some implementation details of LEDA are described in [MN89] and [N92]. 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

a) 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). For C++ compilers not supporting templates there is still a non-template variant (LEDA-N-3.0) available.

b) Arbitrary data types (not only pointer and simple types) can be used as actual type parameters (cf. section 1.2).

c) 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).

d) The LEDA memory management system can be customized for user-defined classes (cf. section 7.3)

e) The efficiency of many data types and algorithms has been improved.

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

1. Basics

1.1 A First Example

The following program can be compiled and linked with LEDA's basic library *libL.a* (cf. section 1.10). 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 1.9 for more details). The specification of the data type *d_array* can be found in section 4.4. We use it also as a running example to discuss the principles underlying LEDA in sections 1.2 to 1.10.

Parameterized data types in LEDA are realized by templates, inheritance and dynamic binding (see [N92] for details). For C++ compilers not supporting templates there is still available a non-template version of LEDA using declare macros as described in [N90].

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.

- **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 input function	void **Read**(T&, istream&)
d) an output function	void **Print**(const T&, ostream&)

A compare function "int **compare**(const T&, const T&)" (cf. section 1.6) has to be defined if the data type requires that T is linearly ordered. Section 1.4 contains a complete example.

- **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

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

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., $dictionary\langle string, int\rangle$ D, is creating a new dictionary object with name D rather than introducing a new variable of type $dictionary\langle string, int\rangle$; 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

a) 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 ($E\ x,\ list\_item\ it,\ rel\_pos\ p = after$)
   defines the interface of the insert operation on a list L of elements of type E (cf. section 3.7). Insert takes as arguments an element x of type E, a *list_item it* and an optional relative position argument p. 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 of I as an argument and returns a variable of type E.

b) 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 7.5). For the insert operation on lists this definition reads:
A new item with contents x is inserted after (if $p = after$) or before (if $p = before$) 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 ([AS89]). 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. section 4). 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 ([Pu89]), instead of the default data structure (cf. section 4.4.5).

```
#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, ...). Section 9 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 can be found in [N92].

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 6.1).

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 subtypes. Whenever a type T is used in such a situation, e.g. in *dictionary*<$T, \ldots$> the function

$$int \quad compare(T, T)$$

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) $x\ rel\ y$
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*(T, T) is said to define the linear order *rel* on T if

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

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;

  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 6.1) 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 $dictionary\langle point_1, \ldots\rangle$, 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, cosnt *point*& y)
{ // lexicographic ordering on polar coordinates }

DEFINE_LINEAR_ORDER(*point*,*pol_cmp*,*pol_point*)

Now, dictionaries based on either ordering can be used.

$dictionary\langle pol\_point, int\rangle\ D_1$; // polar ordering
$dictionary\langle point, int\rangle\ D_0$; // 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 Items

Many of the advanced data types in LEDA (e.g. dictionaries), 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 4.3) 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.

1.8 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. 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}

1.9 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 are described in Table 10.1 and 10.2.

1.10 Libraries

The implementions of all LEDA data types and algorithms are precompiled and contained in 5 libraries (libL.a, libG.a, libP.a, libWs.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 (section 2), basic data types (section 3), dictionaries and priority queues (section 4). 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

b) **libG.a** is the LEDA graph library. It contains the implementations of all graph data types and algorithms (section 5). 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

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 (section 6). To compile a program using two-dimensional 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**, **libWs.a** are the LEDA libraries for graphic windows under the X11 (xview) or SunView window systems. Application programs using data type *window* (cf. section 6.7) have to be linked with one of these libraries:

a) For the X11 (xview) window system:
CC *prog.c* -lP -lG -lL -lWx -lxview -lolgx -lX11 -lm

b) For the SunView window system:
CC *prog.c* -lP -lG -lL -lWs -lsuntool -lsunwindow -lpixrect -lm

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

2. Simple Data Types

2.1 Boolean Values (bool)

An instance of the data type *bool* has either the value *true* or *false*. The usual C++ logical operators && (and), || (or), ! (negation) are defined for *bool*.

2.3 Strings (string)

Data type *string* is the LEDA equivalent of *char** in C++ . The differences to the *char**-type are that assignment, compare and concatenation operators are defined and that argument passing by value works properly, i.e., there is passed a copy of the string and not only a copy of a pointer. Furthermore a few useful operations for string manipulations are available.

1. Creation of a string

a) *string* s;

b) *string* $s(char * \ p)$;

b) *string* $s(char \ c)$;

introduces a variable s of type *string*. s is initialized with the empty string (variant a), the string constant p (variant b), or the one-character string "c" (variant c).

2. Operations on a string s

int	s.length()	returns the length of string s
char&	s [*int* i]	returns the character at position i *Precondition*: $0 \le i \le s.\text{length}()-1$
string	s (*int* i, *int* j)	returns the substring of s starting at position i and ending at position j *Precondition*: $0 \le i \le j \le s.\text{length}()-1$
string	s.tail(*int* i)	returns the last i characters of s
string	s.head(*int* i)	returns the first i characters of s
int	s.pos(*string* s_1)	returns the first position of s_1 in s if s_1 is

		a substring of s, -1 otherwise
int	s.pos(*string* s_1, *int* i)	returns the first position of s_1 in s right of position i (-1 if no such position exists)
string	s.insert(*string* s_1, *int* i)	returns $s(0, i-1) + s_1 + s(i, s.length() - 1)$ *Precondition*: $0 \le i \le s$.length()-1
string	s.replace(*string* s_1, *string* s_2, *int* $i = 1$)	returns the string created from s by replacing the i-th occurence of s_1 in s by s_2
string	s.replace_all(*string* s_1, *string* s_2)	returns the string created from s by replacing all occurences of s_1 in s by s_2
string	s.replace(*int* i, *int* j, *string* s_1)	returns the string created from s by replacing $s(i, j)$ by s_1
string	s.replace(*int* i, *string* s_1)	returns the string created from s by replacing $s[i]$ by s_1
string	s.del(*string* s_1, *int* $i = 1$)	returns s.replace(s_1, "", i)
string	s.del_all(*string* s_1)	returns s.replace_all(s_1, "")
string	s.del(*int* i, *int* j)	returns s.replace(i, j, "")
string	s.del(*int* i)	returns s.replace(i, "")
void	s.read(*istream* I, *char* *delim* =' ')	reads characters from input stream I into s until the first occurence 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	$s + s_1$	returns the concatenation of s and s_1
string&	$s += s_1$	appends s_1 to s and returns s
bool	$s == s_1$	true iff s and s_1 are equal
bool	$s != s_1$	true iff s and s_1 are not equal
bool	$s < s_1$	true iff s is lexicographically smaller than s_1
bool	$s > s_1$	true iff s is lexicographically greater than s_1
bool	$s <= s_1$	returns $(s < s_1) \mid\mid (s == s_1)$
bool	$s >= s_1$	returns $(s > s_1) \mid\mid (s == s_1)$

ostream&	O << s	writes string s to the output stream O
istream&	I >> s	read(I,' ')

3. Implementation

Strings are implemented by C++ character vectors. All operations on a string s take time $O(s.length())$.

2.4 Real-Valued Vectors (vector)

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

1. Creation

a) *vector* v(*int* d);

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

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

creates an instance v of type *vector*; v is initialized to the zero vector of dimension d (variant a), the two-dimensional vector (a, b) (variant b) or the three-dimensional vector (a, b, c) (variant c).

2. Operations on a vector v

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*: $0 \le i \le v$.dim()-1.
vector	$v + v_1$	Addition *Precondition*: v.dim() $=$ v_1.dim().
vector	$v - v_1$	Subtraction *Precondition*: v.dim() $=$ v_1.dim().
double	$v * v_1$	Scalar multiplication

		Precondition: v.dim() $= v_1$.dim().
vector	$v * r$	Componentwise multiplication with double r
bool	$v == v_1$	Test for equality
bool	$v\ != v_1$	Test for inequality
ostream&	$O << v$	writes v componentwise to the output stream O
istream&	$I >> v$	reads v componentwise from the input stream I

3. Implementation

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

2.5 Real-Valued Matrices (matrix)

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

1. Creation

matrix $M(int\ n,\ int\ m)$;

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

2. Operations on a matrix M

int	M.dim1()	returns n, the number of rows of M.
int	M.dim2()	returns m, the number of cols of M.
vector	M.row(*int* i)	returns the i-th row of M (an m-vector). *Precondition*: $0 \leq i \leq n-1$.
vector	M.col(*int* i)	returns the i-th column of M (an n-vector). *Precondition*: $0 \leq i \leq m-1$.
matrix	M.trans()	returns M^T ($m \times n$ - matrix).
double	M.det()	returns the determinant of M. *Precondition*: M is quadratic.

matrix	M.inv()	returns the inverse matrix of M. *Precondition*: M.det() $\neq 0$.
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*: $0 \leq i \leq n-1$ and $0 \leq j \leq m-1$.
matrix	$M + M_1$	Addition *Precondition*: M.dim1() $=$ M_1.dim1() and M.dim2() $=$ M_1.dim2().
matrix	$M - M_1$	Subtraktion *Precondition*: M.dim1() $=$ M_1.dim1() and M.dim2() $=$ M_1.dim2().
matrix	$M * M_1$	Multiplication *Precondition*: M.dim2() $=$ M_1.dim1().
matrix	$M * r$	Multiplication with double
vector	$M * v$	Multiplication with vector *Precondition*: M.dim2() $=$ v.dim().
ostream&	$O << M$	writes matrix M to the output stream O
istream&	$I >> M$	reads matrix M from the input stream I

3. 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 $O(nm)$.

3. Basic Data Types

3.1 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, called the index set of A, to the set of variables of data type E, called 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 $[a..b]$.

3. Operations

$E\&$	$A\ [int\ i]$	returns $A(i)$. *Precondition*: $a \le i \le b$
int	A.low()	returns the minimal index a
int	A.high()	returns the maximal index b
$void$	A.sort(int $(*cmp)(E\&, E\&)$)	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 $cmp(out_i, out_j) \le 0$ for $i \le j$ and there is a permutation π of $[a..b]$ such that $out_i = in_{\pi(i)}$ for $a \le i \le b$.
$void$	A.sort(int $(*cmp)(E\&, E\&)$, $int\ l$, $int\ h$)	applies the above defined sorting operations to the sub-array $A[l..h]$.
int	A.binary_search($E\ x$, int $(*cmp)(E\&, E\&)$)	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.
$void$	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 a prompt.
void	A.print(*ostream O*, *char space* = ' ')	Prints the contents of array A 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	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 a 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 $O(|I|)$.

3.2 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 $[b..c]$. $A(i,j)$ is called the element at position (i,j).

2. Creation

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

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

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 $O(|I|)$.

3.3 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.
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.
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

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.

3.4 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.
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

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.

3.5 Bounded Stacks (b_stack)

1. Definition

An instance S of the parmaterized data type *b_stack*<*E*> is a stack (see section 2.3) of bounded size.

2. Creation

b_stack<*E*> $S(n)$;

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 $O(n)$.

3.6 Bounded Queues (b_queue)

1. Definition

An instance Q of the paramerized data type *b_queue*<*E*> is a queue (see section 2.4) of bounded size.

2. Creation

b_queue<*E*> $Q(n)$;

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 $O(n)$.

3.7 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

a) 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

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 $[1 \ldots \vert L \vert]$ (0 if x is not in L).

b) Update Operations

$list\_item$	L.insert($E\ x, list\_item\ it,\ direction\ dir = after$)	inserts a new item $< x >$ after (if $dir = after$) or before (if $dir = before$) item it into L and returns it. *Precondition*: it is an item in L.
$list\_item$	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()$, after$))
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.
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.
$void$	L.assign($list\_item\ it,\ E\ x$)	makes x the contents of item it. *Precondition*: it is an item in L.
$void$	L.conc($list\&\ L1$)	appends list $L1$ to list L and makes $L1$ the empty list
$void$	L.split($list\_item\ it, list\&\ L1,\ L2$)	splits L at item it into lists $L1$ and $L2$

and makes L the empty list. More precisely, if $L = x_1, \ldots, x_{k-1}, it, x_{k+1}, \ldots, x_n$ then $L1 = x_1, \ldots, x_{k-1}$ and $L2 = it, x_{k+1}, \ldots, x_n$
Precondition: it is an item in L.

void	L.apply(*void* $(*f)(E\&)$)	for all items $< x >$ in L function f is called with argument x (passed by reference).
void	L.sort(*int* $(*cmp)(E\&, E\&)$)	sorts the items of L using the ordering defined by the compare function $cmp : E \times E \longrightarrow int$, with $cmp(a,b) \begin{array}{l} < 0, \text{ if } a < b \\ = 0, \text{ if } a = b \\ < 0, \text{ if } a > b \end{array}$ More precisely, if $L = (x_1, \ldots, x_n)$ before the sort then $L = (x_{\pi(1)}, \ldots, x_{\pi(n)})$ for some permutation π of $[1..n]$ and $cmp(L[x_j], L[x_{j+1}]) \leq 0$ for $1 \leq j < n$ after the sort.
void	L.bucket_sort(*int* i, *int* j, *int* $(*f)(E\&)$)	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

c) Input and Output

void	L.read(*istream* I, *char* $delim = ' \backslash n'$)	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* $delim = ' \backslash n'$)	Calls L.read(*cin*, *delim*) to read L from the standard input stream *cin*.
void	L.read(*string* s, *char* $delim = ' \backslash n'$)	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.

d) 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(*direction dir* = *forward*)	moves the iterator to its successor (predecessor) if *dir* = *forward* (*backward*) and to the first (last) item if it is undefined (= nil), returns the iterator.
bool	L.current_element(E& x)	if the iterator is defined (≠ nil) its contents is assigned to x and true is returned else false is returned
bool	L.next_element(E& x)	L.move_iterator(*forward*) + return L.current_element(x)
bool	L.prev_element(E& x)	L.move_iterator(*backward*) + return L.current_element(x)

e) Operators

E&	L [*list_item it*]	returns a reference to the contents of *it*.
list<*E*>&	L = L_1	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 $1 \le i \le n$.

4. Iteration

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" }

5. 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.

3.8 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

4. Iteration

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

5. Implementation

Sets are implemented by randomized search trees ([AS89]). 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.

3.9 Integer Sets (int_set)

1. Definition

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

2. Creation

int_set $S(a, b)$;

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

2. Operations

void	S.insert(*int* x)	adds x to S *Precondition*: $a \leq x \leq b$.
void	S.del(*int* x)	deletes x from S *Precondition*: $a \leq x \leq b$.
bool	S.member(*int* x)	returns true if x in S, false otherwise *Precondition*: $a \leq x \leq b$.
void	S.clear()	makes S the empty set
int_set	$S1 \; = \; S2$	assignment
int_set	$S1 \mid S2$	returns the union of $S1$ and $S2$
int_set	$S1$ & $S2$	returns the intersection of $S1$ and $S2$
int_set	~ S	returns the complement of S

3. 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 $O(b - a + 1)$.

3.10 Partitions (partition)

1. Definition

An instance of the data type *partition* consists of a finite set of items (predefined type *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.

2. 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.

3. Implementation

Partitions are implemented by the union find algorithm with weighted union and path compression (cf. [T83]). Any sequence of n make_block and $m \geq n$ other operations takes time $O(m\alpha(m,n))$.

4. Example

Spanning Tree Algorithms (cf. graph)

3.11 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* w)	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 [T83]). All operations take amortized time $O(\log n)$ where n is the number of maketree operations.

4. Priority Queues and Dictionaries

4.1 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.

2. Creation

a) *priority_queue*<K,I> Q;

b) *_priority_queue*<K,I, *prio_impl*> Q;

creates an instance Q of type *priority_queue*<K,I> and initializes it with the empty priority queue. Variant a) chooses the default data structure (cf. 4.1.4), and variant b) chooses class *prio_impl* as the implementation of the queue (cf. section 9 for a list of possible implementation parameters).

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)
void	Q.del_item(*pq_item* it)	removes the item it from Q. *Precondition*: it is an item in Q.
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.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 $inf(it)$.
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.clear()	makes Q the empty priority queue
bool	Q.empty()	returns true, if Q is empty, false otherwise
int	Q.size()	returns the size of Q.

4. Implementation

Priority queues are implemented by Fibonacci heaps ([FT84]. 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 8.1)

4.2 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 4.1) whose information type is a fixed interval $[a..b]$ of integers.

2. Creation

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

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 on a b_priority_queue Q

The operations are the same as for the data type *priority_queue* with the additional precondition that any information argument must be in the range $[a..b]$.

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.

4.3 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

a) $dictionary<K,I>\ \ D;$

b) $\_dictionary<K,I,dic\_impl>\ D\ ;$

creates an instance D of type $dictionary<K,I>$ and initializes it with the empty dictionary. Variant a) chooses the default data structure (cf. 4.3.4), and variant b) chooses class *dic_impl* as the implementation of the dictionary (cf. section 9 for a list of possible implementation parameters).

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.
bool	D.empty()	returns true if D is empty, false otherwise.
int	D.size()	returns the size of D.

4. Implementation

Dictionaries are implemented by randomized search trees ([AS89]). 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

Using a dictionary to count the number of occurrences of the elements in a sequence of strings, terminated by string "stop".

```
#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) << "\n";
 }
```

4.4 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 thelinearly 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.

2. Creation

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

b) *_d_array*$<I,E,impl>$ $A(x)$;

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. Variant a) chooses the default data structure (cf. 4.4.5), and variant b) chooses class *impl* as the implementation of the dictionary (cf. section 9 for a list of possible implementation parameters).

3. Operations

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

4. Iteration

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

5. Implementation

Dictionary arrays are implemented by randomized search trees ([AS89]). Access operations $A[x]$ take time $O(\log dom(A))$. The space requirement is $O(dom(A))$.

6. Example

Program 1: Using a dictionary array to count the number of occurences 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] << "\n";
}
```

Program 2: Using a d_array to realize an english/german dictionary.

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

main()
{
   d_array<string,string> trans;
   trans["hello"]  = "hallo";
   trans["world"]  = "Welt";
   trans["book"]   = "Buch";
   trans["key"]    = "Schluessel";
   string s;
   forall_defined(s,trans) cout << s << " " << trans[s] << "\n";
}
```

4.5 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 the data type I, called the index type of A, to the set of variables of data type E, called the element type of A. I must be an integer, pointer, or item type.

2. Creation

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

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

4. Iteration

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

5. Implementation

Hashing arrays are implemented by dynamic perfect hashing ([DKMMRT88]). Access operations $A[x]$ take time $O(1)$. Hashing arrays are more efficient than dictionary arrays.

4.6 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 of 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

a) *sortseq*$<K,I>$ S;

b) *_sortseq*$<K,I,seq\_impl>$ S ;

creates an instance S of type *sortseq*$<K,I>$ and initializes it to the empty sorted sequence. Variant a) chooses the default data structure (cf. 4.6.4), and variant b) chooses class *seq_impl* as the implementation of the sorted sequence (cf. section 9 for a list of possible implementation parameters).

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) $< k$ or key(it) is minimal with key(it) $> k$
seq_item	S.locate($K\ k$)	returns the item $< k', i >$ in S such that k' is minimal with $k' >= k$ (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 $k' <= k$ (nil if no such item exists).
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.max()	returns the item with maximal key (nil if S is empty).
seq_item	S.min()	returns the item with minimal key (nil if S is empty).
void	S.del_item(*seq_item* it)	removes the item it from S. *Precondition*: it is an item in S.
void	S.del($K\ k$)	removes the item with key k from S (null operation if no such item exists).
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* $< K,I >$ & S_1, *sortseq* $< K,I >$ & S_2)	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 $S1 = x_1, \ldots, x_{k-1}, it$ and $S2 = x_{k+1}, \ldots, x_n$ *Precondition*: *it* is an item in S.
sortseq $< K,I >$	S.conc(*sortseq* $< K,I >$ & S_1)	appends S_1 to S, makes S_1 empty and returns S. *Precondition*: S.key(S.max()) $\leq$ 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_item 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

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

```
#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 stop = S.locate(s2);
         for (seq_item it = S.locate(s1); it != stop; it = S.succ(it))
             cout << S.key(it) << "\n";
       }
 }
```

4.7 Persistent Dictionaries (p_dictionary)

1. Definition

The difference between dictionaries (cf. section 4.3) 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) $\neq k$.

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 $< k, i > \in D$.

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).
I	D.access(K k)	returns the information associated with k *Precondition*: there is an item with key k in D.
p_dictionary(K, I)	D.del(K k)	returns $\{ x \in \mathrm{D} \mid key(x) \neq k \}$.
p_dictionary(K, I)	D.del_item(*p_dic_item* it)	returns $\{ x \in \mathrm{D} \mid x \neq it \}$.
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)	Let $k = key(it)$, returns D.del_item$(it) \cup \{< k, i >\}$. *Precondition*: $it \in D$.
p_dictionary(K, I)	D.clear()	returns an empty persistent dictionary.
bool	D.empty()	returns true if D is empty, false otherwise.
int	D.size()	returns the size of D.

4. Implementation

Persistent Dictionaries are implemented by leaf oriented persistent red black trees (cf. [DSST89]). Operations insert, lookup, del_item, del take time $O(\log n)$, key, inf, empty, size, change_inf and clear take time $O(1)$. The space requirement is $O(1)$ for each update operation.

5. Graphs and Related Data Types

5.1 Directed graphs (graph)

1. Definition

An instance G of the data type *graph* consists of a list of nodes V and a list of edges E (*node* and *edge* are predefined data types). Every edge $e \in E$ is a pair of nodes $(v, w) \in V \times V$, v is called the source of e and w is called the target of e. With every node v the list of its adjacent edges $adj\_list(v) = \{\ e \in E\ |source(e) = v\ \}$, called the adjacency list of v, is associated.

2. Creation

graph G;

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

3. Operations

a) Access operations

int	G.indeg(*node* v)	returns the indegree of node v
int	G.outdeg(*node* v)	returns the outdegree of node 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
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<node>	G.all_nodes()	returns the list V of all nodes of G
node	G.first_node()	returns the first node in V
node	G.last_node()	returns the last node in V
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)
list<edge>	G.all_edges()	returns the list E of all edges of G
edge	G.first_edge()	returns the first edge in E

edge	G.last_edge()	returns the last edge in E
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)
list<*edge*>	G.adj_edges(*node v*)	returns the list of all edges adjacent to v
list<*node*>	G.adj_nodes(*node v*)	returns the list of all nodes adjacent to v
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 $source(e)$ (nil if it does not exist)
edge	G.adj_pred(*edge e*)	returns the predecessor of edge e in the adjacency list of $source(e)$ (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 $source(e)$
edge	G.cyclic_adj_pred(*edge e*)	returns the cyclic predecessor of edge e in the adjacency list of $source(e)$
node	G.choose_node()	returns a node of G (nil if G is empty)
edge	G.choose_edge()	returns an edge of G (nil if G is empty)

b) Update operations

node	G.new_node()	adds a new node to G and returns it
void	G.del_node(*node v*)	deletes v and all edges adjacent to v from G. *Precondition*: $indeg(v) = 0$.
edge	G.new_edge(*node v, w*)	adds a new edge (v, w) to G by appending it to the adjacency list of v and returns it.
edge	G.new_edge(*edge e, node w, rel_pos dir = after*)	adds a new edge $e' = (source(e), w)$ to G by inserting it after (dir=after) or before (dir =before) edge e into the adjacency list of $source(e)$, returns e'.
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 $e' = (w, v)$ into G by appending it to the adjacency list of w, returns e'
void	G.rev()	all edges in G are reversed
void	G.sort_nodes(*int*(**cmp*)(*node&*, *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 8.1)
void	G.sort_nodes(*node_array*<*T*> A)	the nodes of G are sorted according to the entries of node_array A (cf. section 5.7) *Precondition*: T must be linearly ordered
void	G.sort_edges(*int*(**cmp*)(*edge&*, *edge&*))	the edges of G 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 8.1)
void	G.sort_edges(*edge_array*<*T*> A)	the edges of G are sorted according to the entries of edge_array A (cf. section 5.7) *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. The list of all inserted edges is returned.
void	G.make_undirected()	every edge (v, w) in G is inserted into the adjacency list of w.
void	G.make_directed()	every edge (v, w) in G is removed from the adjacency list of w.
void	G.clear()	makes G the empty graph

c) Iterators

With the adjacency list of every node v is associated a list iterator called the adjacency iterator of v (cf. list). There are operations to initialize, move, and read these iterators. They are used to implement iteration statements (forall_adj_edges, forall_adj_nodes).

void	G.init_adj_iterator(*node* v)	assigns nil to the adjacency iterator of node v
bool	G.current_adj_edge(*edge*& e, *node* v)	if the adjacency iterator of v is defined ($\neq$ nil) its contents is assigned to e and true is returned else false is returned.
bool	G.next_adj_edge(*edge*& e, *node* v)	moves the adjacency iterator of v forward (to the first item of *adj_list*(v) if it is nil) and returns G.current_adj_edge(e, v)
bool	G.current_adj_node(*node*& w, *node* v)	if G.current_adj_edge(e, v) = true then assign *target*(e) to w and return true, else return false
bool	G.next_adj_node(*node*& w, *node* v)	if G.next_adj_edge(e, v) = true then assign *target*(e) to w and return true, else return false
void	G.reset()	assign nil to all adjacency iterators in G

d) Miscellaneous operations

void	G.write(*ostream* O = *cout*)	writes a compressed representation of G to the output stream O.
void	G.write(*string* s)	writes a compressed representation of G to the file with name s.
void	G.read(*istream* I = *cin*)	reads a compressed representation of G from the input stream I.
void	G.read(*string* s)	reads a compressed representation of G from the file with name s.
void	G.print_node(*node* v, *ostream* O = *cout*)	writes a readable representation of node v to the output stream O

void	G.print_edge(*edge* e, *ostream* $O = cout$)	writes a readable representation of edge e to the output stream O
void	G.print(*ostream* $O = cout$)	writes a readable representation of G to the output stream O

4. 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_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" }

5. Implementation

Graphs are implemented by doubly linked adjacency lists. Most operations take constant time, except of 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 $O(n + m)$.

5.2 Undirected graphs (ugraph)

1. Definition

An instance G of the data type *ugraph* consists of a set of nodes V and a set of undirected edges E. Every edge $e \in E$ is a set of two nodes $\{v, w\}$, v and w are called the endpoints of e. With every node v is associated the list of its adjacent edges $adj\_list(v) = \{\ e \in E\ | v \in e\ \}$.

2. Creation

ugraph G;

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

3. Operations

Most operations are the same as for directed graphs. The following operations are either additional or have different effects.

node	G.opposite(*node* v, *edge* e)	returns w if $e = \{v, w\}$, nil otherwise
int	G.degree(*node* v)	returns the degree of node v.
edge	G.new_edge(*node* v, *node* w)	inserts the undirected edge $\{v, w\}$ into G by appending it to the adjacency lists of both v and w and returns it
edge	G.new_edge(*node* v, *node* w, *edge* $e1$, *edge* $e2$, $dir1 = after$, $dir2 = after$)	inserts the undirected edge $\{v, w\}$ after (if $dir1 = after$) or before (if $dir1 = before$) the edge $e1$ into the adjacency list of v and after (if $dir2 = after$) or before (if $dir2 = before$) the edge $e2$ into the adjacency list of w and returns it
edge	G.adj_succ(*edge* e, *node* v)	returns the successor of edge e in the adjacency list of v.
edge	G.adj_pred(*edge* e, *node* v)	returns the predecessor of edge e in the adjacency list of v.
edge	G.cyclic_adj_succ(*edge* e, *node* v)	returns the cyclic successor of edge e in the adjacency list of v.
edge	G.cyclic_adj_pred(*edge* e, *node* v)	returns the cyclic predecessor of edge e in the adjacency list of v.

4. Implementation

Undirected graphs are implemented like directed graphs by adjacency lists. The adjacency list of a node v contains all edges $\{v, w\}$ of the graph. Most operations take constant time, except of 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 $O(n + m)$.

5.3 Planar Maps (planar_map)

1. Definition

An instance M of the data type *planar_map* is the combinatorial embedding of a planar graph.

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 an undirected 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

Most operations are the same as for directed graphs. The following operations are either additional or have different effects.

face	M.adj_face(*edge* e)	returns the face of M to the right of e.
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.
list<*edge*>	M.adj_edges(*face* f)	returns the list of all edges of M bounding face f in clockwise order.
list<*node*>	M.adj_nodes(*face* f)	returns the list of all nodes of M adjacent to face f in clockwise order.
edge	M.reverse(*edge* e)	returns the reversal of edge e in M.
edge	M.first_face_edge()	returns the first edge of face f in M.
edge	M.succ_face_edge(*edge* e)	returns the successor edge of e in face f 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* e_1, *edge* e_2)	inserts the edge $e = (source(e_1), source(e_2))$ and its reversal edge into M. *Precondition*: e_1 and e_2 are bounding the same face F. The operation splits F into two new faces.
edge	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(*face* f)	splits face f into triangles by inserting a new node u and connecting it to all nodes of f. Returns u.
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.
list<edge>	M.triangulate()	triangulates all faces of M by inserting new edges. The list of inserted edges is is returned.
int	M.straight_line_embedding(*node_array*(*int*) *xcoord*, *node_array*(*int*) *ycoord*)	computes a straight line embedding for M with integer coordinates $xcoord[v]$, $ycoord[v]$) in the range $0 \ldots 2(n-1)$ for every node v of M, and returns the maximal used coordinate.

4. 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" }

5. Implementation

Planar maps are implemented by parameterized directed graphs. All operations take constant time, except of, 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 take time $O(n)$ where n is the current size (number of edges) of the planar map.

5.4 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

$GRAPH{<}vtype, etype{>}\quad G;$

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

3. Operations

In addition to the operations of the data type *graph* (see section 2):

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* y)	makes y the information of edge e
node	G.new_node(*vtype* x)	adds a new node $< x >$ to G and returns it
edge	G.new_edge(*node* v, w, *etype* x)	
		adds a new edge $e =< v, w, x >$ to G by

		appending it to the adjacency list of v and returns e.
edge	G.new_edge(*edge e, node w, etype x, dir = after*)	adds a new edge $e' = < source(e), w, x >$ to G by inserting it after (*dir*=after) or before (*dir* =before) edge e into the adjacency list of $source(e)$ and returns e'.
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.

4. Operators

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

5. Implementation

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

5.5 Parameterized undirected graphs (UGRAPH)

1. Definition

A parameterized undirected graph G is an undirected 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 the undirected edge $\{v, w\}$ with information y and $< x >$ to denote a node with information x.

2. Creation

$UGRAPH<vtype, etype>\ \ G;$

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

3. Operations

In addition to the operations of the data type *ugraph* (see section 5.3):

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
edge	G.new_edge(*node* v, *node* w, *etype* x)	inserts the undirected edge $< \{v, w\}, x >$ into G by appending it to the adjacency lists of both v and w and returns it
edge	G.new_edge(*node* v, *node* w, *edge* $e1$, *edge* $e2$, *etype* x, *rel_pos* $dir1 =$) $after$, *rel_pos* $dir2 = after$)	inserts the undirected edge $< \{v, w\}, x >$ after (if $dir1 = after$) or before (if $dir1 = before$) the edge $e1$ into the adjacency list of v and after (if $dir2 = after$) or before (if $dir2 = before$) the edge $e2$ into the adjacency list of w and returns it.

4. Implementation

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

5.6 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 operaations to access or update the node and face entries.

2. Creation

$PLANAR\_MAP{<}vtype, ftype{>}\quad M(GRAPH(vtype, ftype)\ G);$

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

In addition to the operations of the data type *planar_map*:

vtype	M.inf(*node* v)	returns the information of node v
ftype	M.inf(*face* f)	returns 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* y)	makes y the information of face f
edge	M.new_edge(*edge* e_1, *edge* e_2, *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 $\inf(f') = y$.

4. Implementation

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

5.7 Node and edge arrays (node_array, edge_array)

1. Definition

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

2. Creation

a) *node/edge_array*<E> A;

b) *node/edge_array*<E> A(*graph* G);

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

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

creates an instance A of type *node_array*(E) or *edge_array*(E). Variant a) initializes the index set of A to the empty set, Variants b) and c) initialize the index set of A to be the entire node (edge) set of graph G, i.e., A is made valid for all nodes (edges) currently contained in G. Variant c) in addition initializes $A(i)$ with x for all nodes (edges) i of G. Variant d) makes A a *node/edge_array*(E) valid for up to n nodes/edges of G, *Precondition*: $n \geq |V|$ $(|E|)$, this is useful if you want to use the array for later inserted nodes/edges.

3. Operations

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

4. Implementation

Node (edge) 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 (edges) currently in G. The space requirement is $O(n)$.

Remark: A node (edge) array is only valid for a bounded number of the nodes (edges) contained in G. This number is either the total number of nodes of G at the moment of the array creation (variants a) ...c)) or it is explicitely set by the user (variant d)). Access operations for additional later added nodes (edges) are not allowed. Fully dynamic node and edge arrays can be realized by using hashing arrays, e.g., $h\_array(node, \ldots)$ (cf. section 4.5).

5.8 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

a) *node_matrix*<*E*> M;

b) *node_matrix*<*E*> $M(G)$;

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

creates an instance M of type *node_matrix*<*E*>. Variant a) initializes the index set of M to the empty set, Variants b) and c) initialize the index set of A 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. Variant c) in addition initializes $M(v, w)$ 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 $v, w \in V$.
E&	M (*node* v, *node* w)	returns the variable $M(v, w)$. *Precondition*: M must be valid for v and w.
node_array(E)&	$M[v]$	returns the node_array $M(v)$.

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.

5.9 Sets of nodes and edges (node_set, edge_set)

1. Definition

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

2. Creation

node_set $S(G)$;
edge_set $S(G)$;

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

3. Operations on a node/edge set S

void	S.insert(x)	adds node (edge) x to S
void	S.del(x)	removes node (edge) x from S
bool	S.member(x)	returns true if x in S, false otherwise
node/edge	S.choose()	return a node (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

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

5.10 Node partitions (node_partition)

1. Definition

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

2. Creation

node_partition $P(G)$;

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

3. Operations on a node_partition P

bool	P.same_block(*node* v, *node* w)	returns true if v and w belong to the same block of P.
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.

5.11 Node priority queues (node_pq)

1. Definition

An instance Q of the parameterized data type *node_pq*<*I*> is a partial function from the nodes of a graph G to the linearly ordered type I.

2. Creation

node_pq<*I*> $Q(G)$;

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

3. Operations

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

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.

5.12 Graph Algorithms

This sections 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.

5.12.1 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 $1 \le ord[v] \le |V|$ and $ord[v] < ord[w]$ for all edges $(v, w) \in E$. TOPSORT returns true if G is acyclic and false otherwise.

The algorithm ([Ka62]) 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 ([T72]) 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 ([T72]) 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 ([M84]) has running time $O(|V| + |E|)$.

• **Connected Components**

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

COMPONENTS takes an undirected graph $G(V,E)$ as argument and computes for every node $v \in V$ an integer $compnum[v]$ from $[0 \ldots c-1]$ 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 ([M84]) 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 $[0 \ldots c-1]$ 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 ([M84]) 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 ([GK79]) has running time $O(|V| \cdot |E|)$.

5.12.2 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 ([Di59,FT87]) 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 ([Be58]) has running time $O(|V| \cdot |E|)$.

- **All Pairs Shortest Paths**

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

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

ALL_PAIRS_SHORTES_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).

The algorithm ([Be58,Fl62]) 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 ([GT88]) has running time $O(|V|^3)$.

- **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 ([E65,T83]) 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 ([HK75]) 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 ([FT87]) has running time $O(|V| \cdot |E|)$.

- **Spanning Tree**

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

SPANNING_TREE takes as argument an undirected graph $G(V, E)$. It computes a spanning tree T of G, SPANNING_TREE returns the list of edges of T.

The algorithm ([M84]) has running time $O(|V| + |E|)$.

- **Minimum Spanning Tree**

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

list<edge> MIN_SPANNING_TREE(*ugraph&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 ([Kr56]) has running time $O(|E| \log |V|)$.

5.12.3 Algorithms for Planar Graphs

• **Planarity Test**

bool PLANAR(*graph*&*G*)

PLANAR takes as input a directed graph $G(V, E)$ and performs a planarity test for G. If 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 ([HT74]) has running time $O(|V| + |E|)$.

• **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.

The algorithm ([HU89]) 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 ([Fa48]) has running time $O(|V|^2)$.

5.13 Miscellaneous

5.13.1 Some useful functions

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 test_bigraph(*graph*& G, *nodelist*& A, *nodelist*& B)

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

bool compute_correspondence(*graph*& G, *edge_array*(*edge*)& *reversal*)

computes for every edge $e = (v, w)$ in G its reversal $reversal[e] = (w, v)$ in G (nil if not present). Returns true if every edge has a reversal and false otherwise.

void eliminate_parallel_edges(*graph*& G)

removes all parallel edges from 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*)

6. Data Types For Two-Dimensional Geometry

6.1 Basic two-dimensional objects

LEDA provides a collection of simple data types for two-dimensional geometry, such as points, segments, lines, circles, and polygons. All these types can be used as type parameters in parameterized data types. Their declarations are contained in the header file <**LEDA/plane.h**>. Furthermore, some basic algorithms (section 6.1.6) are included.

6.1.1 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

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

b) *point* p;

introduces a variable p of type *point* initialized to the point (x, y). Variant b) initializes p to the point $(0, 0)$.

3. Operations

double	p.xcoord()	returns the first coordinate of point p
double	p.ycoord()	returns the second coordinate of point 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(*vector* v)	returns $p + v$, i.e., p translated by vector v. *Precondition*: v.dim() = 2.
point	p.translate(*double* α, *double* d)	returns the point created by translating p in direction α by distance d. The

		direction is given by its angle with a right oriented horizontal ray.
point	p.rotate(*point* q, *double* α)	returns the point created by a rotation of p about point q by angle α.
point	p.rotate(*double* α)	returns p.rotate(*point*$(0,0)$, α).

4. Operators

bool	*point* == *point*	test for equality
bool	*point* != *point*	test for inequality
point	*point* + *vector*	translation by vector

Input and output operators:

ostream&	*ostream* << *point*	writes a point to an output stream
istream&	*istream* >> *point*	reads the coordinates of a point (two doubles) from an input stream

6.1.2 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

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

b) *segment* s(*double* x_1, *double* y_1, *double* x_2, *double* y_2);

c) *segment* s(*point* p, *double* α, *double* d);

d) *segment* s;

introduces a variable s of type *segment*. s is initialized to the segment from p to q (variant a), to the segment from (x_1,y_1) to (x_2,y_2) (variant v), to the segment with start point p, direction α, and length d (variant c), or to the empty segment (variant d).

3. Operations

point	s.start()	returns the start point of segment s.
point	s.end()	returns the end point of segment s.
double	s.xcoord1()	returns the x-coordinate of s.start().
double	s.ycoord1()	returns the y-coordinate of s.start().
double	s.xcoord2()	returns the x-coordinate of s.end().
double	s.ycoord2()	returns the y-coordinate of s.end().
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(*segment* t)	returns the angle between s and t, i.e., t.direction() - s.direction().
double	s.angle()	returns s.direction().
bool	s.horizontal()	returns true iff s is horizontal.
bool	s.vertical()	returns true iff s is vertical.
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.
segment	s.rotate(*point* q, *double* α)	returns the segment created by a rotation of s about point q by angle α.
segment	s.rotate(*double* α)	returns s.rotate(s.start(),α).
segment	s.translate(*vector* v)	returns $s + v$, i.e., the segment created by translating s by vector v. *Precondition*: v has dimension 2.
segment	s.translate(*double alpha*, *double* d)	returns the segment created by a translation of s in direction α by distance d.

3. Operators

bool	*segment* == *segment*	test for equality
bool	*segment* != *segment*	test for inequality
segment	*segment* + *vector*	translation by vector

Input and output operators:

ostream&	*ostream* << *segment*	writes a segment to an output stream.
istream&	*istream* >> *segment*	reads the coordinates of a segment (four doubles) from an input stream.

6.1.3 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

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

b) *line* l(*segment* s);

c) *line* l(*point* p, *double* α);

d) *line* l;

introduces a variable l of type *line*. l is initialized to the line passing through points p and q directed form p to q (variant a), to the line supporting segment s (variant b), to the line passing through point p with direction α (variant c), or a line through $(0,0)$ with direction 0 (variant d).

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.horizontal()	returns true iff l is horizontal.
bool	l.vertical()	returns true iff l is vertical.
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&* p)	if l and g are not collinear and intersect the intersection point is assigned to p and true is returned, otherwise false is returned.
bool	l.intersection(*segment* s, *point&* p)	if l and s are not collinear and intersect the intersection point is assigned to p and true is returned, otherwise false is returned.
line	l.translate(*vector* v)	returns $l + v$, i.e., the line created by translating l by vector v. *Precondition*: v has dimension 2.
line	l.translate(*double* α, *double* d)	returns the line created by a translation of l in direction α by distance d.
line	l.rotate(*point* q, *double* α)	returns the line created by a rotation of l about point q by angle α.
line	l.rotate(*double* α)	returns l.rotate(*point*$(0,0)$, α).
segment	l.perpendicular(*point* p)	returns the nromal of p with respect to l.

4. Operators

bool	*line* == *line*	test for equality
bool	*line* != *line*	test for inequality

6.1.4 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

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

b) *polygon* P;

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. Variant b) creates the empty polygon and assigns it to P.

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(*line* l)	returns $P \cap l$ as a list of points.
list<point>	P.intersection(*segment* s)	returns $P \cap s$ as a list of points.
list<polygon>	P.intersection(*polygon* Q)	returns $P \cap Q$ as a list of points.
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(*vector* v)	returns $P + v$, i.e., the polygon created by translating P by vector v. *Precondition*: v has dimension 2.
polygon	P.translate(*double* α, *double* d)	returns the polygon created by a translation of P in direction α by distance d
polygon	P.rotate(*point* q, *double* α)	returns the polygon created by a rotation of P about point q by angle α.

double	P.size()	returns the size of P.
bool	P.empty()	returns true if P is empty, false otherwise.

4. Operators

bool	*polygon* == *polygon*	test for equality
bool	*polygon* != *polygon*	test for inequality

6.1.5 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

a) *circle* C(*point* p, *double* r);

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

c) *circle* C;

introduces a variable C of type *circle*. C is initialized to the circle with center p and radius r (variant a), to the circle with center (x, y) and radius r (variant b), or to the empty circle (variant c).

3. Operations

double	C.radius()	returns the radius of C.
point	C.center()	returns the center of C.
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.
list<point>	C.intersection(*circle* D)	returns $C \cap D$ 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(*vector* v)	returns $C + v$, i.e., the circle created by translating C by vector v. *Precondition*: v.dim $= 2$.
circle	C.translate(*double* α, *double* d)	returns the circle created by a translation of C in direction α by distance d.
circle	C.rotate(*point* q, *double* α)	returns the circle created by a rotation of C about point q by angle α.

4. Operators

bool	*circle* == *circle*	test for equality
bool	*circle* != *circle*	test for inequality

6.1.6 Algorithms

- **Line segment intersection**

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 ([BO79]) has running time $O((n+k)\log n)$, where n is the number of segments and k is the number of intersections.

- **Convex hull of point set**

polygon 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.

- **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 ([De92]) has running time $O(n \log n)$ (with high probability), where n is the number of sites.

6.2 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

$d2\_dictionary{<}K1, K2, I{>}\quad D;$

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.max_key1()	returns the item with maximal first key.
$dic2\_item$	D.max_key2()	returns the item with maximal second key.
$dic2\_item$	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.insert($K1\ k_1,\ K2\ k_2,\ I\ i$)	associates the information i with the keys k_1 and k_2. If there is an item $< k_1, k_2, j >$ in D then j is replaced by i, else a new item $< k_1, k_2, i >$ is added to D. In both

		cases the item is returned.
dic2_item	D.lookup($K1\ k_1,\ K2\ k_2$)	returns the item with keys k_1 and k_2 (nil if no such item exists in D).
list<dic2_item>	D.range_search($K1\ a,\ K1\ b,\ K2\ c,\ K2\ d$)	returns the list of all items $< k_1, k_2, i > \in D$ with $a \le k_1 \le b$ and $c \le k_2 \le d$.
list<dic2_item>	D.all_items()	returns the list of all items of D.
void	D.del($K1\ k_1,\ K2\ k_2$)	deletes the item with keys k_1 and k_2 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 [Wi85, Lu78] 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 $O(n \log n)$.

6.3 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_itemit*)	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 [Wi85, Lu78] 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 $O(logn)$ otherwise. Here n is the current size of the point set. The space requirement is $O(n^2)$.

6.4 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 real 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 $[x, y] \subset \mathbb{R}$ 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 S then 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 [Wi85, Lu78]. 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 $O(n \log n)$.

6.5 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

a) *segment_set<I>* $S(double\ \alpha)$;

b) *segment_set<I>* S;

creates an empty instance S of type *segment_set<I>* with orientation α. Variant b) creates a segment set of orientation zero, i.e., for 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.
ps_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_itemit*)	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 ([Wi85, Lu78]) 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 $O(n \log n)$.

6.6 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 5.3 and 5.6). 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.

2. Operations

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

3. Implementation

Planar subdivisions are implemented by parameterized planar maps and an additional data structure for point location based on persistent search trees ([DSST89]). 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 requiremnt and the initialization time is $O(n^2)$.

6.7 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 or SunView window system. There are two object code libraries libWx.a, and libWs.a containing implementations for both the X11 (xview toolkit) and the SunView environments. Application programs using data type *window* have to be linked with one of these libraries (cf. section 1.6):

a) For the X11 (xview) window system:
CC *prog.c* -lP -lG -lL -lWx -lxview -lolgx -lX11 -lm

b) For the SunView window system:
CC *prog.c* -lP -lG -lL -lWs -lsuntool -lsunwindow -lpixrect -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 850×850 pixel square 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

a) *window* W(*int xpix*, *int ypix*, *int xpos*, *int ypos*);

b) *window* W(*int xpix*, *int ypix*);

c) *window* W;

Variant a) creates a window W of physical size $xpix \times ypix$ pixels with its upper left corner at position $(xpos, ypos)$ on the screen, variant b) places W into the upper right corner of the screen, and variant c) creates a 850×850 pixel window positioned into the upper right corner.

All three variants initialize the coordinates of W to $x0 = 0$, $x1 = 100$ and $y0 = 0$. 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* x_0, *double* x_1, *double* y_0)	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* x_0, *double* x_1, *double* y_0, *int* d)	like init(x_0, x_1, y_0) followed by set_grid(d)
void	W.clear()	W is erased.

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(*linestyle 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_redraw(*void* $(*F)()$)	Sets the redraw function parameter to F.

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* $c = black$)

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, $c = black$)

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

3.5 Drawing line segments

void W.draw_segment(*double* x_1, *double* y_1, *double* x_2, *double* y_2, *color* $c = black$)

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

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

draws a line segment from point p to point q.

void W.draw_segment(*segment* s, *color* $c = black$)

draws line segment s.

3.6 Drawing lines

void W.draw_line(*double* x_1, *double* y_1, *double* x_2, *double* y_2, *color* $c = black$)

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* $c = black$)

draws a straight line passing through points p and q.

void W.draw_line(*line* l, *color* $c = black$)

draws line l.

void W.draw_hline(*double* y, *color* $c = black$)

draws a horizontal line with y-coordinate y.

void W.draw_vline(*double* x, *color* $c = black$)

draws a vertical line with x-coordinate x.

3.7 Drawing arrows

void W.draw_arrow(*double* x_1, *double* y_1, *double* x_2, *double* y_2, *color* $c = black$)

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

void W.draw_arrow(*point* p, *point* q, *color* $c = black$)

draws an arrow pointing from point p to point q.

void W.draw_arrow(*segment* s, *color* $c = black$)

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

3.8 Drawing circles

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

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

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

draws the circle with center p and radius r.

void W.draw_circle(*circle* C, *color* $c = black$)

draws circle C.

3.9 Drawing discs

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

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

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

draws a filled circle with center p and radius r.

void W.draw_disc(*circle* C, *color* $c = black$)

draws filled circle C.

3.10 Drawing polygons

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

draws the polygon with vertex sequence lp.

void W.draw_polygon(*polygon* P, *color* $c = black$)

draws polygon P.

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

draws the filled polygon with vertex sequence lp.

void W.draw_filled_polygon(*polygon* P, *color* $c = black$)

draws filled polygon P.

3.11 Drawing functions

void W.plot_xy(*double* x_0, *double* x_1, (*double*)($*F$)(*double*), *color* $c = black$)
draws function F in range $[x_0, x_1]$, i.e., all points (x, y) with $y = F(x)$ and $x_0 \le x \le x_1$

void W.plot_yx(*double* y_0, *double* y_1, (*double*)($*F$)(*double*), *color* $c = black$)
draws function F in range $[y_0, y_1]$, i.e., all points (x, y) with $x = F(y)$ and $y_0 \le y \le y_1$

3.12 Drawing text

void W.draw_text(*double* x, *double* y, *string* s, *color* $c = black$)
writes string s starting at position (x, y).

void W.draw_text(*point* p, *string* s, *color* $c = black$)
writes string s starting at position p.

void W.draw_ctext(*double* x, *double* y, *string* s, *color* $c = black$)
writes string s centered at position (x, y).

void W.draw_ctext(*point* p, *string* s, *color* $c = black$)
writes string s centered at position p.

3.13 Drawing nodes

void W.draw_node(*double* x_0, *double* y_0, *color* $c = black$)
draws a node at position (x_0, y_0).

void W.draw_node(*point* p, *color* $c = black$)
draws a node at position p.

void W.draw_filled_node(*double* x_0, *double* y_0, *color* $c = black$)
draws a filled node at position (x_0, y_0).

void W.draw_filled_node(*point* p, *color* $c = black$)
draws a filled node at position p.

void W.draw_text_node(*double* x, *double* y, *string* s, *color* $c = black$)
draws a node with label s at position (x_0, y_0).

void W.draw_text_node(*point* p, *string* s, *color* $c = black$)
draws a node with label s at position p.

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

draws a node with integer label i at position (x_0, y_0).

void W.draw_int_node(*point p, int i, color c = black*)
draws a node with integer label i at position p.

3.14 Drawing edges

void W.draw_edge(*double* x_1, *double* y_1, *double* x_2, *double* y_2, *color c = black*)
draws an edge from (x_1, y_1) to (x_2, y_2).

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

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

void W.draw_edge_arrow(*double* x_1, *double* y_1, *double* x_2, *double* y_2, *color c = black*)
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 = black*)
draws a directed edge from p to q.

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

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 to (x, y) and the pressed button is returned.

int W.read_mouse_seg(*double* x_0, *double* y_0, *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_rect(*double* x_0, *double* y_0, *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_circle(*double* x_0, *double* y_0, *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.
bool	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* $*$ S)	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* $*$ S)	like read_panel with vertical button layout
int	W.read_int(*string* p)	displays a panel with prompt p for integer input, returns the input
double	W.read_real(*string* p)	displays a panel with prompt p for real input returns the input
string	W.read_string(*string* p)	displays a panel with prompt p for string input, returns the input
void	W.message(*string* s)	displays message s (each call adds a new line).
void	W.del_message()	deletes the text written by all previous message operations.

3.16 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** $(W >> p)$ $\{ \ldots \}$ ".

3.16.1 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)

3.16.2 Input

window&	$W >> p$	reads a point p: clicking the left button assigns the current cursor position to p.
window&	$W >> s$	reads a segment s: use the left button to input the start and end point of s.
window&	$W >> l$	reads a line l: use the left button to input two different points on l
window&	$W >> C$	reads a circle C: use the left button to input the center of C and a point on C
window&	$W >> P$	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.

6.8 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. With each item (except of text items) is associated a variable of a certain type (int, bool, string, double, color) whose value can be manipulated through the item and a string label.

2. Creation

panel $P(string\ h)$;

creates an empty panel P with header h.

3. Operations

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($string\ s,\ doubel\&\ x$)	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.int_item($string\ s,\ int\&\ x$)	adds an integer item with label s and variable x to P.
void	P.int_item($string\ s,\ int\&\ x,\ int\ min,\ int\ max$)	adds an integer slider item with label s, variable x, and range $min,\ldots,max$ to P.
void	P.int_item($string\ s,\ int\&\ x,\ int\ low,\ int\ high,\ int\ step$)	adds an integer choice item with label s, variable x, range $low,\ldots,\ high$, and step size $step$ 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($string\ s,\ string\&\ x, list\langle string\rangle\ L$)	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*s_1, *string* $s_2, \ldots, s_k$)	adds an integer item with label s, variable x, and choices $s_1, \ldots, s_k$ to P $(k \leq 5)$
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
int	P.open()	P is displayed on the screen until a button of P is selected. Returns the number of the button.

7. Miscellaneous

This section describes some additional useful data types, functions and macros of LEDA. They can be used in any program that includes the <LEDA/basic.h> header file.

7.1 Streams

The stream data types described in this section are all derived from the C++ stream types *istream* and *ostream*. Some of these types may be obsolete in combination with the latest versions of the standard C++ I/O library.

7.1.1 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 the file with name s.

3. Operations

All operations and operators (>>) defined for C++ istreams can be applied to file input streams as well.

7.1.2 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(*string s*);

creates an instance O of type file_ostream connected to the file with name s.

3. Operations

All operations and operators (<<) defined for C++ ostreams can be applied to file output streams as well.

7.1.3 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.

7.1.4 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.clear()	clears the contents of O
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.

7.1.5 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.

7.1.6 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.

7.2 Some useful functions and macros

int	read_int(string s = "")	prints s and reads an integer
char	read_char(string s = "")	prints s and reads a character
double	read_real(string s = "")	prints s and reads a real number
string	read_string(string s = "")	prints s and reads a line of input
bool	Yes(string s = "")	returns (read_char(s) == 'y')
void	init_random()	initializes the random number generator.
double	random()	returns a real valued random number in $[0, 1]$
int	random(int a, int b)	returns a random integer in $[a..b]$
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 T up to this moment and assings the current time to T.
void	print_statistics()	prints a summary of the currently used memory

newline	cout << "\n"
forever	for(;;)
loop(a,b,c)	for $(a = b; a <= c; a++)$
in_range(a,b,c)	$(b <= a$ && $a <= c)$
Max(a,b)	$((a > b)\ ?\ a\ :\ b)$
Min(a,b)	$((a > b)\ ?\ b\ :\ a)$

7.3 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 creates new and delete operators for type T allocating and deallocating memory using LEDA's internal memory manager. 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;
```

7.4 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.

8. Programs

8.1 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;
        L.append(v);
        S.push(v);
       }

   while ( !S.empty() )
      { v = S.pop();
        forall_adj_nodes(w,v)
           if ( !reached[w] )
           { reached[w] = true;
             L.append(w);
             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);
         }
     }
 }
```

8.2 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&lt;segment,point&gt; X_structure;
typedef p_dictionary&lt;segment,int&gt;     Y_structure;

sortseq<double,Y_structure>  HISTORY;

void SWEEP(list&lt;segment&gt;& 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);
}
```

9. Implementations

9.1 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).

9.1.1 Dictionaries

ab_tree	a-b tree	dictionary, d_array, **sortseq**	[BC72]
avl_tree	AVL tree	dictionary, d_array	[AVL62]
bb_tree	BB[α] tree	dictionary, d_array, sortseq	[BM80]
ch_hashing	hashing with chaining	dictionary, d_array	[M84]
dp_hashing	dyn. perf. hashing	**h_array**	[DKMMRT88,W92]
pers_tree	persistent tree	**p_dictionary**	[DSST89]
rb_tree	red-black tree	dictionary, d_array, sortseq	[GS78]
rs_tree	rand. search tree	**dictionary**, **d_array**, sortseq	[AS89]
skiplist	skip lists	dictionary, d_array, sortseq	[Pu90]

9.1.2 Priority Queues

f_heap	Fibonnacci heap	**priority_queue**	[FT87]
p_heap	pairing heap	priority_queue	[SV87]
k_heap	k-nary heap	priority_queue	[M84]
m_heap	monotonic heap	priority_queue	[M84]
eb_tree	Emde-Boas tree	priority_queue	[EKZ77,W92]

9.1.3 Geometry

range_tree	range tree	**d2_dictionary**, **point_set**	[Wi85,Lu78]
seg_tree	segment tree	**seg_set**	[B79,Ed82]
ps_tree	priority search tree	—	[MC81]
iv_tree	interval tree	**interval_set**	[MC80,Ed82]
delaunay_tree	delaunay tree	**point_set**	[De92]

9.2 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.

9.2.1 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 4.3 and 4.4).

```
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;
};
```

9.2.2 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 4.1).

```
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;
};
```

9.2.3 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 4.6).

```
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;
};
```

10. Tables

10.1 Data Types

Name	Item	Header	Library	Page
array	——	array.h	libL.a	21
array2	——	array.h	libL.a	23
b_priority_queue	b_pq_item	b_prio.h	libL.a	41
b_queue	——	b_queue.h	libL.a	27
b_stack	——	b_stack.h	libL.a	26
bool	——	basic.h	libL.a	15
circle	——	plane.h	libP.a	85
cmd_istream	——	stream.h	libL.a	110
cmd_ostream	——	stream.h	libL.a	110
d2_dictionary	d2_dic_item	d2_dictionary.h	libP.a	88
d_array	——	d_array.h	libL.a	44
dictionary	dic_item	dictionary.h	libL.a	42
edge_array	——	graph.h	libG.a	65
edge_set	——	edge_set.h	libG.a	68
file_istream	——	stream.h	libL.a	109
file_ostream	——	stream.h	libL.a	109
graph	node/edge	graph.h	libG.a	53
GRAPH	node/edge	graph.h	libG.a	61
h_array	——	h_array.h	libL.a	46
int_set	——	int_set.h	libL.a	34
interval_set	is_item	interval_set.h	libP.a	92
line	——	plane.h	libP.a	82
list	list_item	list.h	libL.a	28
matrix	——	matrix.h	libL.a	19
node_array	——	graph.h	libG.a	65
node_matrix	——	graph.h	libG.a	67
node_partition	——	node_partition.h	libG.a	69
node_pq	——	node_pq.h	libG.a	70
node_set	——	node_set.h	libG.a	68
panel	——	window.h	libP.a/libWx.a	106
partition	partition_item	partition.h	libL.a	35
planar_map	node/edge/face	planar_map.h	libG.a	59
point	——	plane.h	libP.a	79
point_set	ps_item	point_set.h	libP.a	90
polygon	——	plane.h	libP.a	84

priority_queue	pq_item	prio.h	libP.a	39
p_dictionary	p_dic_item	p_dictionary.h	libL.a	50
PLANAR_MAP	node/edge/face	planar_map.h	libG.a	64
queue	——	queue.h	libL.a	25
segment	——	plane.h	libP.a	80
segment_set	seg_item	segment_set.h	libP.a	94
set	——	set.h	libL.a	33
sortseq	seq_item	sortseq.h	libL.a	47
stack	——	stack.h	libL.a	24
string	——	basic.h	libL.a	15
string_istream	——	stream.h	libL.a	111
string_ostream	——	stream.h	libL.a	111
subdivision	node/face	subdivision.h	libP.a	96
tree_collection	d_vertex	tree_collection.h	libL.a	36
ugraph	node/edge	ugraph.h	libG.a	57
UGRAPH	node/edge	ugraph.h	libG.a	63
vector	——	vector.h	libL.a	17
window	——	window.h	libP.a/libWx.a	97

10.2 Algorithms

Name	Header	Library	Page
ALL_PAIRS_SHORTEST_PATHS	graph_alg.h	libG.a	73
BELLMAN_FORD	graph_alg.h	libG.a	73
BFS	graph_alg.h	libG.a	72
COMPONENTS	graph_alg.h	libG.a	72
CONVEX_HULL	plane_alg.h	libP.a	87
DFS	graph_alg.h	libG.a	71
DFS_NUM	graph_alg.h	libG.a	71
DIJKSTRA	graph_alg.h	libG.a	73
MAX_CARD_MATCHING	graph_alg.h	libG.a	74
MAX_CARD_BIPARTITE_MATCHING	graph_alg.h	libG.a	74
MAX_FLOW	graph_alg.h	libG.a	74
MAX_WEIGHT_BIPARTITE_MATCHING	graph_alg.h	libG.a	75
MIN_SPANNING_TREE	graph_alg.h	libG.a	75
PLANAR	graph_alg.h	libG.a	76
SEGMENT_INTERSECTION	plane_alg.h	libP.a	87
SPANNING_TREE	graph_alg.h	libG.a	75
STRAIGHT_LINE_EMBEDDING	graph_alg.h	libG.a	76
STRONG_COMPONENTS	graph_alg.h	libG.a	72
TOPSORT	graph_alg.h	libG.a	71
TRANSITIVE_CLOSURE	graph_alg.h	libG.a	72
TRIANGULATE_PLANAR_MAP	graph_alg.h	libG.a	76
VORONOI	plane_alg.h	libP.a	87

11. References

[AS89] C. Aragon, R. Seidel: "Randomized Search Trees", Proc. 30th IEEE Symposium on Foundations of Computer Science, 540-545, 1989

[AHU83] A.V. Aho, J.E. Hopcroft, J.D. Ullman: "Data Structures and Algorithms", Addison-Wesley Publishing Company, 1983

[AVL62] G.M. Adelson-Veslkii, Y.M. Landis: "An Algorithm for the Organization of Information", Doklady Akademi Nauk, Vol. 146, 263-266, 1962

[B79] J.L. Bentley: "Decomposable Searching Problems", Information Processing Letters, Vol. 8, 244-252, 1979

[Be58] R.E. Bellman: "On a Routing Problem", Quart. Appl. Math. 16, 87-90, 1958

[BC72] R. Bayer, E. McCreight: "Organizatino and Maintenance of Large Ordered Indizes", Acta Informatica, Vol. 1, 173-189, 1972

[BM80] N. Blum, K. Mehlhorn: "On the Average Number of Rebalancing Operations in Weight-Balanced Trees", Theoretical Computer Science 11, 303-320, 1980

[BO79] J.L. Bentley, Th. Ottmann: "Algorithms for Reporting and Counting Geometric Intersections", IEEE Trans. on Computers C 28, 643-647, 1979

[CLR90] T.H. Cormen, C.E. Leiserson, R.L. Rivest: "Introduction to Algorithms", MIT Press/McGraw-Hill Book Company, 1990

[CT76] D. Cheriton, R.E. Tarjan: "Finding Minimum Spanning Trees", SIAM Journal of Computing, Vol. 5, 724-742, 1976

[De92] O. Devillers: "Robust and Efficient Implementation of the Delaunay Tree", Technical Report, INRIA, 1992

[Di59] E.W. Dijkstra: “A Note on Two Problems in Connection With Graphs”, Num. Math., Vol. 1, 269-271, 1959

[DKMMRT88] 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

[DSST89] 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

[E65] J. Edmonds: “Paths, Trees, and Flowers”, Canad. J. Math., Vol. 17, 449-467, 1965

[Ed82] H. Edelsbrunner: “Intersection Problems in Computational Geometry”, Ph.D. thesis, TU Graz, 1982

[EKZ77] P.v. Emde Boas, R. Kaas, E. Zijlstra: “Design and Implementation of an Efficient Pririty Queue”, Math. Systems Theory, Vol. 10, 99-127, 1977

[Fa48] I. Fary: “On Straight Line Representing of Planar Graphs”, Acta. Sci. Math. Vol. 11, 229-233, 1948

[Fl62] F.W. Floyd: “Algorithm 97: Shortest Paths”, Communcication of the ACM, Vol. 5, p. 345, 1962

[FT87] 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

[GK79] A. Goralcikova, V. Konbek: “A Reduct and Closure Algorithm for Graphs”, Mathematical Foundations of Computer Science, LNCS 74, 301-307, 1979

[GOP90] K.E. Gorlen, S.M. Orlow, P.S. Plexico: “Data Abstraction and Object-Oriented Programming in C++ ”, John Wiley & Sons, 1990

[**GS78**] 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

[**GT88**] Goldberg, R.E.Tarjan: "A New Approach to the Maximum Flow Problem", Journal of the ACM, Vol. 35, 921-940, 1988

[**HK75**] 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

[**HT74**] J.E. Hopcroft, R.E. Tarjan: "Efficient Planarity Testing", Journal of the ACM, Vol. 21, 549-568, 1974

[**HU89**] T. Hagerup, C. Uhrig: "Triangulating a Planar Map Without Introducing multiple Arcs", unpublished, 1989

[**Ka62**] A.B. Kahn: "Topological Sorting of Large Networks", Communications of the ACM, Vol. 5, 558-562, 1962

[**Kr56**] J.B. Kruskal: "On the Shortest Spanning Subtree of a Graph and the Travelling Salesman Problem", Proc. American Math. Society 7, 48-50, 1956

[**Li89**] S.B. Lippman: "C++ Primer", Addison-Wesley, Publishing Company, 1989

[**Lu78**] G.S. Luecker: "A Data Structure for Orthogonal Range Queries", Proc. 19th IEEE Symposium on Foundations of Computer Science, 28-34, 1978

[**M84**] K. Mehlhorn: "Data Structures and Algorithms", Vol. 1–3, Springer Publishing Company, 1984

[**MC80**] D.M. McCreight: "Efficient Algorithms for Enumerating Intersecting Intervals", Xerox Parc Report, CSL-80-09, 1980

[**MC81**] D.M. McCreight: "Priority Search Trees", Xerox Parc Report, CSL-81-05, 1981

[**MN89**] K. Mehlhorn, S. Näher: " LEDA, a Library of Efficient Data Types and Algorithms", TR A 04/89, FB10, Universität des Saarlandes, Saarbrücken, 1989

[**N90**] S.Näher: "LEDA2.0 User Manual", technischer Bericht A 17/90, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1990

[**N92**] S. Näher: "Parameterized Data Types in LEDA", in preparation

[**Pu90**] W. Pugh: "Skip Lists: A Probabilistic Alternative to Balanced Trees", Communications of the ACM, Vol. 33, No. 6, 668-676, 1990

[**S91**] B. Stroustrup: " The C++ Programming Language, Second Edition", Addison-Wesley Publishing Company, 1991

[**SV87**] J.T. Stasko, J.S. Vitter: "Pairing Heaps: Experiments and Analysis", Communications of the ACM, Vol. 30, 234-249, 1987

[**T72**] R.E. Tarjan: "Depth First Search an Linear Graph Algorithms", SIAM Journal of Computing, Vol. 1, 146-160, 1972

[**T83**] R.E. Tarjan: "Data Structures and Network Algorithms", CBMS-NSF Regional Conference Series in Applied Mathematics, Vol. 44, 1983

[**We92**] M. Wenzel: "Wörterbücher für ein beschränktes Universum", Diplomarbeit, Fachbereich Informatik, Universität des Saarlandes, 1992

[**Wi85**] D.E. Willard: "New Data Structures for Orthogonal Queries", SIAM Journal of Computing, 232-253, 1985