LEDA User Manual

Version 2.0

A 17/90

Stefan Näher

Fachbereich Informatik
Universität des Saarlandes
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) A clear separation between (abstract) data types and the data structures used to implement them. This distinction is frequently not made in the combinatorial algorithms literature, but is crucial for a library.

2) Generic data types: Most of the data types in LEDA have type parameters. For example, a dictionary has a key type K and an information type I and a specific dictionary type is obtained by setting, say, K to **int** and I to **real**.

3) LEDA is extendible: Users can include own data types either by implementing data structures from scratch in C++ or by combining already existing LEDA data types.

4) Ease of use: All data types and algorithms are precompiled C++ modules which can be linked with application programs.

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 [S86] or [L89]). The main concepts and some implementation details of LEDA are described in [MN89]. 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.

1. Basics

We start with an example. The following program counts the number of occurrences of each string in a sequence of strings

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

declare2(d_array,string,int)

main()
{
  d_array(string,int) N(0);
  string s;
  while (cin >> s && s != "stop") N[s]++;
  forall_defined(s,N) cout << s << " " << N[s] << "\n";
}
```

The program can be compiled using the LEDA library (cf. section 1.9). When executed it reads a sequence of strings from the standard input until the string "stop" is encountered 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.

The program above uses the data type dictionary array (*d_array*) from the library. This is expressed by the include statement (cf. section 1.8 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 1.1 to 1.6.

1.1 Specifications

In general the specification of a LEDA data type consists of five parts: a definition of the set of objects comprising the (parameterized) abstract data type, a description of how to derive a concrete data type from a parameterized 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, informations about the implementation. The five parts appear under the headers definition, type declaration, 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. A concrete dictionary array type is declared by a type declaration which we discuss next.

- **Type Declaration**

This part gives the syntax for deriving specific data types from parameterized or generic data types, i.e., it shows how to set the formal type parameters of a generic data type to concrete data types. For a generic data type XYZ with k type parameters the statement

declare$k$(XYZ,$t_1, \ldots, t_k$)

introduces a new data type with name "XYZ($t_1, t_2, ..., t_k$)", where $t_1, \ldots, t_k$ are the names of the actual type parameters. (Due to a limitation of the implementation language C++ , the number k of type parameters also appears in the name of the declare-macro.) For example,

declare2($d\_array, string, int$)

introduces a new data type with name "$d\_array(string, int)$". An object of data type $d\_array(string, int)$ is a injective mapping from the set of all strings to the set of variables of type *int*.

Only simple data types are allowed as actual type parameters in declarations of parameterized data types. Simple data types are the C++ built in types *char* and *int*, all C++ pointer types, the LEDA types *bool*, *real*, *string*, *vector*, and *matrix* (cf. section 2), all basic two-dimensional objects from section 6.1 (*point*, *segment*, *line*, *polygon*, *circle*), and all item types (cf. section 1.5). In order to realize generic data types with more complicated subtypes (such as dictionaries, lists, graphs, ...) pointers to these types must be used.

- **Creation**

A variable of a (previously declared) 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

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

introduces a variable y of the data type with name "$\mathrm{XYZ}(t_1,\ldots,t_k)$" 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 most data types, in particular for the simple types, the assignment operator is available for variables of that type. However, assignment is in general not a constant time operation, e.g., if s_1 and s_2 are variables of type *string* then the assignment $s_1 = s_2$ takes time proportional to the length of the value of s_2.

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

- **Operations**

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

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 not defined. 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 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 red black trees. Access operations $A[x]$ take time $O(\log dom(A))$. The space requirement is $O(dom(A))$.

1.2 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 $T_1, \ldots, T_k$, respectively, and returning a result of type T, i.e, $f : T_1 \times \ldots \times T_k \longrightarrow T$.

1.3 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(*real* α, *real* *dist*)

It can either be called with a vector as argument or with two arguments of type *real* 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 simple data types.

1.4 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 *declare2(dictionary, T, ...)* the function

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

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

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

For each of the simple data types *char*, *int*, *real*, *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 *char*, *int*, and *real*, the lexicographic ordering for *string*, and for *point* the lexicographic ordering of the cartesian coordinates. For all other simple 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 type *pair* as the type of pointers to pairs of real numbers, and then we define the lexicographic order on *pair* by declaring an appropriate compare function.

```
struct Pair {
  real x;
  real y;
};

typedef Pair* pair;
```

```
int  compare(pair& p, 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;
 }

declare2(dictionary, pair, ... );
```

Sometimes, a user may need additional linear orders on a simple 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(point&,point&)* and then declaring the type *POINT(cmp)* with "declare(*POINT, cmp*)". *POINT* is a parameterized data type (cf. section 1.1) with one parameter which must be the name of a comparing function. All data types *POINT(cmp)* derived from *POINT* are equivalent to the data type *point*, with the only exception that if *POINT(cmp)* is used as an actual parameter in a type declaration, e.g. in "declare2(*dictionary, POINT(cmp),...*)", the resulting type (*dictionary(POINT(cmp),...)*) is based on the linear order defined by *cmp*. For every simple data type t (except of pointer types) there exists such an equivalent parameterized type T which can be used to define additional linear orders on t by declaring types $T(cmp)$ as described for points. The name of T is always the name of t written with capital letters.

In the example, we first declare a function *pol_cmp* and the type *POINT(pol_cmp)*.

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

declare(POINT, pol_cmp)
```

Now, dictionaries based on either ordering can be defined.

```
declare2(dictionary, point, ... )
declare2(dictionary, POINT(pol_cmp), ... )
```

```
main()
{
   dictionary(POINT(pol_cmp), int) D1;   //polar ordering
   dictionary(point, int) D0;     //default ordering
 }
```

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

    declare3(dictionary,point,lexicographic_ordering,...)

and

    declare3(dictionary,point,polar_ordering,...)

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.

1.5 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.

1.6 Input and output

Some parameterized data types (e.g. *list*) provide the operations *print* and *read* for printing their contents to the standard output and for initializing an instance of this type by inserting elements read from the standard input. There are two overloaded functions which can be used for defining input and output functions for user-defined pointer types which are used by *read* and *print* operations (and sometimes for error messages).

void *Read*(T&) { ... } defines an input function for objects of type T.

void *Print*(T&) { ... } defines an output function for objects of type T.

Example: We declare the data type *list*(*pair*) (see section 3.7) and want to read and print lists of pairs. Note that the *Read* and *Print* functions have to be declared before the declaration of the list type.

void Read(*pair*&*p*) { *cin* >> $p \to x$ >> $p \to y$; }

void Print(*pair*&*p*) { *cout* << $p \to x$ << $p \to y$; }

```
declare(list, pair)
main()
{
   list(pair) L;
   L.read("L = ");
   L.print("L = ");
}
```

1.7 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 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_adj_nodes(w, v) { the neighbor nodes of v are successively assigned to w}

Note: Update operations on an object x are not allowed inside the body of an iteration statement for x.

1.8 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 are

<LEDA/basic.h>

This header file contains the declarations for the simple data types *bool*, *real*, *string* (section 2), and the macros and functions described in section 7.

<LEDA/graph.h>

contains the declarations for graphs and related data types and the declarations of all graph algorithms (section 5).

<LEDA/plane.h>

contains the two-dimensional objects *point*, *segment*, *line*, *polygon*, and *circle* and some basic two-dimensional algorithms (section 6.1).

<LEDA/sunview.h>

contains a version of the graphic window data type *gwindow* providing an interface to the SunView window system.

1.9 Libraries

The implementions of all LEDA data types and algorithms are precompiled and contained in three libraries which can be linked with C++ application programs.

a) libL.a

This 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). To compile a program *prog.c* using any of these data types the libL.a library has to be used like this:

CC *prog.c* libL.a

b) libG.a

This 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 u data types or algorithms both the libG.a and libL.a library have to be used:

CC *prog.c* libG.a libL.a

c) libP.a

This 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 all libraries have to be used:

CC *prog.c* libP.a libG.a libL.a -lm (-lsuntool -lsunwindow -lpixrect)

Note that the libraries must be given in this order, the *suntool*, *sunwindow*, and *pixrect* libraries must be added if a SunView graphic window (cf. section 6.7) is used.

2. Simple Data Types

Simple data types are the C++ built in types *char*, *int*, the LEDA data types *bool*, *real*, *string*, *vector*, *matrix*, all C++ pointer types and all item types. Simple data types may be used as actual type parameters for generic data types, e.g. *dictionary*(*real*, *string*).

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*.

2.2 Real Numbers (real)

Data type *real* is the LEDA equivalent of the C++ built in type *double*. Variables of the data type *real* behave exactly like variables of type *double* (arithmetic, compare and input/output operators are the same). The only difference between *real* and *double* lies in the fact that *real* is allowed as subtype (type parameter) for generic data types. There is automatic type conversion from *real* to *double*. Thus, all functions taking *double* arguments accept also arguments of type *real* and vice versa. In particular the mathematical functions declared in <math.h> can be used with *real* arguments. The ˜ operator is defined to explicitly convert an instance of the data type *real* to a C++ *double*. This allows the use of the C++ functions *printf* and *form* for formatted ouput of reals.

Example:

```
#include <math.h>

real r = 3.1415;
real s =sin(r);
cout << form("sine of %f = %f\n",~r,~s);
```

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. The ˜ operator converts a string instance to a *char∗*.

1. Creation of a string

a) *string* s;

b) *string* s(*char* ∗ c);

introduces a variable s of type *string* initialized with the empty string. Variant b) takes as argument a string constant c (char∗) and initializes s with 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*: $0 \leq i \leq s$.length()-1
string	s (*int* i, *int* j)	returns the substring of s starting at position i and ending at position j *Precondition*: $0 \leq i \leq j \leq s$.length()-1
string	s.tail(*int* i)	returns $s(i,\ s$.length()-1) *Precondition*: $0 \leq i \leq s$.length()
string	s.head(*int* i)	returns $s(0,\ i)$ *Precondition*: $0 \leq i \leq s$.length()-1
int	s.pos(*string* $s1$)	returns the first position of $s1$ in s if $s1$ is a substring of s, -1 otherwise
string	s.insert(*string* $s1$, *int* i)	returns s.head(i) + $s1$ + s.tail($i+1$) *Precondition*: $0 \leq i \leq s$.length()-1
string	s.replace(*string* $s1$, *string* $s2$)	returns s.head(s.pos($s1$)-1) + $s2$ + s.tail(s.pos($s1$)+$s1$.length()) *Precondition*: $s1$ is a substing of s.
char∗	˜ s	converts s into a C++ string (*char∗*)
string&	s = $s1$	assigns the value of $s1$ to s and returns it

string	$s + s1$	returns the concatenation of s and $s1$
string&	$s += s1$	appends $s1$ to s and returns s
bool	$s == s1$	returns true iff s and $s1$ are equal
bool	$s\ != s1$	returns true iff s and $s1$ are not equal
bool	$s < s1$	returns true iff s is lexicographically smaller than $s1$
bool	$s > s1$	returns true iff s is lexicographically larger than $s1$
bool	$s <= s1$	returns $(s < s1)\ \|\|\ (s == s1)$
bool	$s >= s1$	returns $(s > s1)\ \|\|\ (s == s1)$
ostream&	$O << s$	writes string s to the output stream O
istream&	$I >> s$	reads string s from the input stream I

3. Implementation

Strings are implemented by C++ character vectors. All operations on a string s take time $O(s.length())$, except of $s[]$ and s.length() which take constant time.

2.4 Real-Valued Vectors (vector)

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

1. Creation of a vector

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

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

c) *vector* v(*real* a, *real* b, *real* 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.
real	v.length()	returns the Euclidean length of v
real	v.angle(*vector* w)	returns the angle between v and w.
real&	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().
real	$v * v_1$	Scalar multiplication *Precondition*: v.dim() $=$ v_1.dim().
vector	$v * r$	Componentwise multiplication with real r
vector&	$v = v_1$	Componentwise assignment; returns v *Precondition*: v.dim() $=$ v_1.dim().
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 real variables.

1. Creation of a matrix

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).
real	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() $\neq 0$.
real&	M (*int i*, *int j*)	returns $M_{i,j}$. *Precondition*: $0 \leq i \leq n-1$ and $0 \leq j \leq m-1$.
matrix&	$M = \quad M_1$	Componentwise assignment; returns M. *Precondition*: M.dim1() $=$ M_1.dim1() and M.dim2() $=$ M_1.dim2().

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 real
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 real numbers. All operations take time $O(nm)$, except of det, inv, and solve which take time $O(n!)$, and dim1, dim2, row, and col, which take constant time. The space requirement is $O(nm)$.

3. Basic Data Types

3.1 One Dimensional Arrays (array)

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

1. Declaration of an array type

declare$(array, E)$

introduces a new data type with name $array(E)$ consisting of all arrays with element type E.

2. Creation of an array

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

creates an instance A of type $array(E)$ with index set $[a..b]$.

3. Operations on an array A

E&	A [*int i*]	returns $A(i)$. *Precondition*: $a \leq i \leq b$
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 $cmp(out_i, out_j) \leq 0$ for $i \leq j$ and there is a permutation π of $[a..b]$ such that $out_i = in_{\pi(i)}$ for $a \leq i \leq b$.
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*.

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)

An instance A of the data type *array2* is a mapping from a set of pairs $I = [a..b] \times [c..d]$, called the index set of A, to a set of variables of a 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).

1. Declaration of a two dimensional array type

declare(*array*2, E)

introduces a new data type with name *array*2(E) consisting of all two-dimensional arrays with element type E.

2. Creation of a two-dimensional array

*array*2(E) $A(a,b,c,d)$;

creates an instance A of type *array*2(E) with index set $[a..b] \times [c..d]$.

3. Operations on a two-dimensional array A

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)

An instance S of the data type *stack* is a sequence of elements of a 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.

1. Declaration of a stack type

declare($stack, E$)

introduces a new data type with name $stack(E)$ consisting all all stacks with element type E.

2. Creation of a stack

$stack(E)$ S;

creates an instance S of type $stack(E)$. S is initialized with the empty stack.

3. Operations on a stack S

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.
E	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)

An instance Q of the data type *queue* is a sequence of elements of a 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.

1. Declaration of a queue type

declare$(queue, E)$

introduces a new data type with name $queue(E)$ consisting all all queues with element type E.

2. Creation of a queue

$queue(E)\ \ Q;$

creates an instance Q of type $queue(E)$. Q is initialized with the empty queue.

3. Operations on a queue Q

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.
E	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)

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

1. Declaration of a bounded stack type

declare(*b_stack*, E)

introduces a new data type with name *b_stack*(E) consisting all bounded stacks with element type E.

2. Creation of a bounded stack

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 on a b_stack S

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.
E	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)

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

1. Declaration of a bounded queue type

declare(b_queue, E)

introduces a new data type with name $b\_queue(E)$ consisting all bounded queue with element type E.

2. Creation of a bounded queue

$b\_queue(E)$ $Q(n)$;

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 on a b_queue Q

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.
E	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)

An instance L of the data type *list* is a sequence of items (*list_item*). Each item in L contains an element of a 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.

1. Declaration of a list type

declare(*list*, E)

introduces a new data type with name *list*(E) consisting of all lists with element type E.

2. Creation of list

list(E) L;

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

3. Operations on a list L

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 $[1 \ldots \lvert L \rvert]$ (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 $cmp : E \times E \longrightarrow int$,

$$\text{with } cmp(a,b) \begin{array}{l} < 0, \text{ if } a < b \\ = 0, \text{ if } a = b \\ < 0, \text{ if } a > b \end{array}$$

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 Operations

void L.read(*string s*, *char delim*)

Prints string s on the standard output and then reads a sequence of objects of type E terminated by the delimiter *delim* from the standard input 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(*string s*)

calls L.read(s, '\n').

void L.read(*char delim*)

calls L.read("", *delim*).

void L.read()

calls L.read("", '\n').

void L.print(*string s*, *char space*)

Prints the contents of list L to the standard

		output using the overload *Print* function to print each element. The elements are separated by the space character *space*. String *s* is used as a header.
void	*L*.print(*string s*)	calls *L*.print(*s*, ' ').
void	*L*.print(*char space*)	calls *L*.print("", *space*).
void	*L*.print()	calls *L*.print("", ' ').

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 ($\neq$ 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 *L*.contents(*it*)
list(E)&	*L*1 = *L*2	assignment:

The assignment operator makes list *L*1 a copy of list *L*2. More precisely if *L*2 is the sequence of items $x_1, x_2, \ldots x_n$ then *L*1 is made a sequence of item $y_1, y_2, \ldots y_n$ with $L1[y_i] = L2[x_i]$ for $1 \leq i \leq n$.

5. Iteration

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

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

6. 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)

An instance S of the data type *set* is collection of elements of a linearly ordered type U, 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.

1. Declaration of a set type

declare(*set*, U)

introduces a new data type with name *set*(U) consisting of all sets with element type U. *Precondition*: U is linearly ordered.

2. Creation of a set

set(U) S;

creates an instance S of type *set*(U) and initializes it to the empty set.

3. Operations on a set S

void	S.insert(U x)	adds x to S
void	S.del(U x)	deletes x from S
bool	S.member(U x)	returns true if x in S, false otherwise
U	S.choose()	returns an element of S (error if S is empty) *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 red black trees. 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)

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

1. Creation of an int_set

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 on a int_set S

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)

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.

1. Creation of a partition

partition P;

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

2. Operations on a partition P

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 $O(m\alpha(m,n))$, where α is a functional inverse of Ackerman's function.

4. Example

Spanning Tree Algorithms (cf. graph)

3.11 Dynamic collections of trees (tree_collection)

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

1. Declaration of a dynamic tree collection type

declare(*tree_collection*, I)

introduces a new data type with name *tree_collection*(I) consisting of all dynamic tree collections with information type I.

2. Creation of a tree_collection

tree_collection(I) D;

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

3. Operations on a tree_collection D

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, *real&* 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, *real* x)	Adds real 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)

An instance Q of the data type *priority_queue* is a collection of items (type *pq_item*). Every item contains a key from a type K and an information from a 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. on I.

1. Declaration of a priority queue type

declare2(*priority_queue*, K, I)

introduces a new data type with name *priority_queue*(K, I) consisting of all priority queues with key type K and information type I. *Precondition*: I is linearly ordered.

2. Creation of a priority queue

priority_queue(K, I) Q;

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

3. Operations on a priority_queue Q

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 with minimal information from Q and returns its key. *Precondition*: Q is not empty.

pq_item	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)

An instance Q of the data type *b_priority_queue* is a priority_queue (cf. section 4.1) whose items contain informations from a fixed interval $[a..b]$ of integers.

1. Declaration of a bounded priority queue type

declare(*b_priority_queue*, K)

introduces a new data type with name *b_priority_queue*(K) consisting of all bounded priority queues with key type K.

2. Creation of a bounded priority queue

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)

An instance D of the data type *dictionary* is a collection of items (*dic_item*). Every item in D contains a key from a linearly ordered data type K, called the key type of D, and an information from a 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 item $< k, i > \in D$.

1. Declaration of a dictionary type

declare2(*dictionary*, K, I)

introduces a new data type with name *dictionary*(K, I) consisting of all dictionaries with key type K and information type I. *Precondition*: K is linearly ordered.

2. Creation of a dictionary

dictionary(K, I) D;

creates an instance D of type *dictionary*(K, I) and initializes D to the empty dictionary.

3. Operations on a dictionary D

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. Iteration

forall_dic_items(i, D) { "the items of D are successively assigned to i " }

5. Implementation

Dictionaries are implemented by leaf oriented red black trees. 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 $O(n)$.

6. 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>

declare2(dictionary, string, int)
main()
{
  dictionary(string, int) D;
  string s;
  dic_item it;
  while ( (cin >> s) && (s != "stop") )
        { it = D.lookup(s);
          if (it == nil) D.insert(s,1);
          else D.change_inf(it,D.info(it)+1);
        }
  forall_dic_items(it, D) cout << D.info(it) << " " << D.key(it) << "\n";
}
```

4.4 Dictionary Arrays (d_array)

An instance A of the data type *d_array* (dictionary array) is an injective mapping from a linearly ordered data type I, called the index type of A, to a set of variables of a data type E, called the element type of A.

1. Declaration of a d_array type

declare2(d_array, I, E)

introduces a new data type with name $d\_array(I, E)$ consisting of all d_arrays with index type I and element type E. *Precondition*: I is linearly ordered.

2. Creation of a d_array

$d\_array(I, E)\ \ A(x);$

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 on a d_array A

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 red black trees. 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>

declare2(d_array,string,int)

main()
{
   d_array(string,int) N(0);
   string s;
   while (cin >> s && s != "stop") 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>

declare2(d_array,string,string)

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)

An instance A of the data type *h_array* (hashing array) is an injective mapping from a data type I, called the index type of A, to a set of variables of a data type E, called the element type of A. I must be *char*, *int*, a pointer type, or an item type.

1. Declaration of an hashing array type

declare2(h_array, I, E)

introduces a new data type with name $h\_array(I, E)$ consisting of all h_arrays with index type I and element type E.

2. Creation of a h_array

$h\_array(I, E)\quad A(x);$

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 on a h_array A

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)

An instance S of the data type *sortseq* is a sequence of items (*seq_item*). Every item contains a key from a linearly ordered data type K , called the key type of S, and an information from a 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.

1. Declaration of a sorted sequence type

declare2(*sortseq*, K, I)

introduces a new data type with name *sortseq*(K, I) consisting of all sorted sequences with key type K and information type I.

2. Creation of a sorted sequence

sortseq(K, I) S;

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

3. Operations on a sortseq S

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_item(*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.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*: Item a appears before item b in S.
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. Iteration

forall_seq_items(i, S) { "the items of S are successively assigned to i" }

5. Implementation

Sorted sequences are implemented by (2,4)-trees. Operations lookup, locate, insert, del 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.

6. 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>

declare2(sortseq,string,int);

main()
{
   sortseq(string,int) S;
   string s, s1, s2;
   while ( cin >> s && s != "stop" ) S.insert(s,0);
   while ( cin >> s1 >> s2 )
      { seq_item it1 = S.locate(s1);
        seq_item it2 = S.locate(s2);
        while (it1! = it2)
          { cout << S.key(it1) << "\n";
            it1 = S.succ(it1);
           }
      }
}
```

5. Graphs and Related Data Types

5.1 Directed graphs (graph)

An instance G of the data type *graph* consists of a set of nodes V and a set 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 $adj\_list(v) = \{ e \in E \mid source(e) = v \}$, called the adjacency list of v, is associated.

1. Creation of a graph

graph G;

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

2. Operations on a graph G

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 of all nodes of G
list(*edge*)	G.all_edges()	returns the list of all edges of G
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_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)

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*.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 the adjacency iterator, to move it to the successor or predecessor list item, to access its contents (an edge) and to test if it is defined, ($\neq$ nil). Adjacency iterators 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(*string s*)	writes a compressed representation of G to the file with name s
void	*G*.read(*string s*)	read a compressed representation of G from the file with name s

void	G.print_node(*node* v)	writes a readable representation of node v to the standard output
void	G.print_edge(*edge* e)	writes a readable representation of edge e to the standard output
void	G.print()	writes a readable representation of G to the standard output

e) Operators

graph&	$G = G1$	makes a copy of $G1$ and assigns it to G.

3. 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" }

4. Implementation

Graphs are implemented by 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. Examples

See section 8.1.

5.2 Undirected graphs (ugraph)

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 $adj\_list(v) = \{\ e \in E\ |v \in e\ \}$.

1. Creation of an undirected graph

ugraph G;

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

2. Operations on a ugraph G

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.

3. 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)

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

1. Creation of a planar_map

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.

2. Operations on a planar_map M

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.
list(*edge*)	M.triangulate()	triangulates all faces of M by inserting new edges. The list of inserted edges is is returned.
void	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.

3. Iteration

Additional iteration macros are

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" }

4. 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)

A parameterized graph G is a graph whose nodes and edges contain additional (user defined) informations. 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 informations 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* or *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 informations are hidden. This allows the design of generic graph algorithms, i.e., algorithms accepting instances of any parametrized graph type as argument.

1. Declaration of a parameterized graph type

declare2(*GRAPH*, *vtype*, *etype*)

introduces a new data type with name *GRAPH*(*vtype*, *etype*) consisting of all parameterized graphs with node type *vtype* and edge type *etype*.

2. Creation of a parameterized graph

GRAPH(*vtype*, *etype*) *G*;

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

3. Operations on a GRAPH G

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'.

4. Operators

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

5. Implementation

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

5.5 Parameterized undirected graphs (UGRAPH)

A parameterized undirected graph G is an undirected graph whose nodes and edges contain additional (user defined) informations. 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.

1. Declaration of a parameterized undirected graph type

declare**2**(*UGRAPH*, *vtype*, *etype*)

introduces a new data type with name *UGRAPH*(*vtype*, *etype*) consisting of all undirected parameterized graphs with node type *vtype* and edge type *etype*.

2. Creation of a parameterized undirected graph

UGRAPH(*vtype*, *etype*) G;

creates an instance G of type *UGRAPH*(*vtype*, *etype*) and and initializes it to the empty graph.

3. Operations on a UGRAPH G

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

vtype/etype G.inf(*node/edge a*) — returns the information of node/edge a

void G.assign(*node/edge a*, *vtype/etype x*) — makes x the information of node/edge a

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*, *rel_pos dir1 = after*, *edge e2*, *rel_pos dir2 = after*, *etype x*) — 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 informations take constant time.

5.6 Parameterized planar maps (PLANAR_MAP)

A parameterized planar map M is a planar map whose nodes and faces contain additional (user defined) informations. 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 informations to access or update the node and face informations.

1. Declaration of a parameterized planar_map type

declare2$(PLANAR\_MAP, vtype, ftype)$

introduces a new data type with name $PLANAR\_MAP(vtype, ftype)$ consisting of all parameterized planar maps with node type *vtype* and face type *ftype*. *Precondition*: The data type $GRAPH(vtype, ftype)$, i.e., the parameterized directed graph type with node entries of type *vtype* and edge entries of type *ftype*, has been declared before.

2. Creation of a parameterized planar map

$PLANAR\_MAP(vtype, ftype)$ $M(GRAPH(vtype, ftype)\ G)$;

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

3. Operations on a PLANAR_MAP M

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 informations take constant time.

5.7 Node and edge arrays (node_array, edge_array)

An instance A of the data type *node_array* (*edge_array*) is a partial mapping from the node set V (edge set E) of a (u)graph G to a set of variables of a 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.

1. Declaration of node and edge array types

declare(*node/edge_array*, E)

introduces a new data type with name *node_array*(E) (*edge_array*(E)) consisting of all node (edge) arrays with element type E.

2. Creation of a node array (edge array)

a) *node/edge_array* A;

b) *node/edge_array* A(*graph* G);

c) *node/edge_array* A(*graph* G, 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.

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
E&	A [*node/edge* i]	access the variable $A(i)$. *Precondition:* A must be valid for node (edge)

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 $O(n)$.

Important: A node (edge) array is only valid for the nodes (edges) contained in G at the moment of the array declaration or initialization (*init*). Access operations for later added nodes (edges) are not allowed. Node and edge arrays for dynamic graphs can be realized using hashing arrays (cf. section 4.5), e.g. by **declare2**(*h_array*, *edge*, *int*).

5.8 Two dimensional node arrays (node_matrix)

An instance M of the data type *node_matrix* is a partial mapping from the set of node pairs $V \times V$ of a graph to a set of variables of a 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))).

1. Declaration of a node matrix type

declare(*node_matrix*, E)

introduces a new data type with name *node_matrix*(E) consisting of all node matrices with element type E.

2. Creation of a node_matrix

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 on a node_matrix M

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)

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.

1. Creation of a node or edge set

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.

2. 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

3. 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)

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

1. Creation of a node partition

node_partition $P(G)$;

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

2. 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 node of the block that contains node v.

3. 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)

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

1. Declaration of a node priority queue type

declare($node\_pq, I$)

introduces a new data type with name $node\_pq(I)$ consisting of all node priority queues with information type I.

2. Creating a node priority queue

$node\_pq(I)\ \ Q(G);$

creates an instance Q ot type $node\_pq(I)$ for the nodes of graph G with $dom(Q) = \emptyset$.

3. Operations on a node_pq Q

void	Q.insert(*node* v, I i)	adds the node v with information i to Q. *Precondition*: $v \notin dom(Q)$.
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)
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 NQ. 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 $1 \leq ord[v] \leq |V|$ and $ord[v] < ord[w]$ for all edges $(v, w) \in E$. TOPSORT returns true if G is acyclic and false otherwise.

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.

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).

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 (length of a shortest path) $dist[v]$ from s to v. BFS returns the list of all reached nodes.

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.

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.

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.

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(real) *cost*, *node_array(real) 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.

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(real) cost*,
node_array(real) 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

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(real)& cost*,
node_matrix(real)& 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).

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(real)& cap,*
edge_array(real)& 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. MAXFLOW returns the total flow from s to t.

Running Time: $O(|V|^3)$

- **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 computes a maximum cardinality bipartite matching of G, i.e., a maximal set of edges M such that no two edges in M share an end point (target or source). MAX_CARD_BIPARTITE_MATCHING returns M as a list of edges.

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(*real*)& *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.

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.

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*(*real*)& *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.

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.

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.

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.

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 (if present). Returns true if every edge has a reversal and false otherwise.

void eliminate_parallel_edges(*graph& G*)

removes all parallel edges from G.

5.13.2 Predefined parameterized types

list(node) — *list(edge)*

node_array(int) — *edge_array(int)*
node_array(bool) — *edge_array(bool)*
node_array(real) — *edge_array(real)*
node_array(node) — *edge_array(node)*
node_array(edge) — *edge_array(edge)*

node_matrix(int)
node_matrix(bool)
node_matrix(real)

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**>. The corresponding list types list(point), list(segment), list(line), list(circle), and list(polygon) are also declared in this file. Furthermore, some basic algorithms (section 6.1.6) are included.

Note: This section is preliminary and will probably change in future versions of the library. Above all, there is missing a hierarchy of the data types and a general concept for infinite objects.

6.1.1 Points (point)

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.

1. Creation of a point

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

b) *point* p;

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

2. Operations on a point p

real	p.xcoord()	returns the first coordinate of point p
real	p.ycoord()	returns the second coordinate of point p
real	p.distance(*point* q)	returns the euclidean distance between p and q.
real	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(*real* α, *real* 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, *real* α)	returns the point created by a rotation of p about point q by angle α.

3. Operators

point&	*point* = *point*	assignment
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 reals) from an input stream

6.1.2 Segments (segment)

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.

1. Creation of a segment

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

b) *segment* s(*point* p, *real* α, *real* d);

c) *segment* s;

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

2. Operations on a segment s

point	s.start()	returns the start point of segment s.
point	s.end()	returns the end point of segment s.
real	s.xcoord1()	returns the x-coordinate of s.start().
real	s.ycoord1()	returns the y-coordinate of s.start().
real	s.xcoord2()	returns the x-coordinate of s.end().
real	s.ycoord2()	returns the y-coordinate of s.end().
real	s.length()	returns the length of s.
real	s.direction()	returns the direction of s as an angle in the intervall $(-\pi, \pi]$.
real	s.angle(*segment* t)	returns the angle between s and t, i.e., t.direction() - s.direction().
real	s.angle()	returns s.direction().
bool	s.horizontal()	returns true iff s is horizontal.
bool	s.vertical()	returns true iff s is vertical.
real	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, *real* α)	returns the segment created by a rotation of s about point q by angle α.
segment	s.rotate(*real* 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(*real alpha*, *real* d)	returns the segment created by a translation of s in direction α by distance d.

3. Operators

segment&	*segment* = *segment*	assignment
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 reals) from an input stream.

6.1.3 Straight Lines (line)

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.

1. Creation of a line

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

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

c) *line* l(*point* p, *real* 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).

2. Operations on a line l

real	l.direction()	returns the direction of l.
real	l.angle(*line* g)	returns the angle between l and g, i.e., g.direction() - l.direction().
real	l.angle()	returns l.direction().
bool	l.horizontal()	returns true iff l is horizontal.
bool	l.vertical()	returns true iff l is vertical.
real	l.slope()	returns the slope of l.

		Precondition: l is not vertical.
real	l.y_proj(*real* x)	returns p.ycoord(), where $p \in l$ with p.xcoord() $= x$. *Precondition*: l is not vertical.
real	l.x_proj(*real* y)	returns p.xcoord(), where $p \in l$ with p.ycoord() $= y$. *Precondition*: l is not horizontal.
real	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(*real alpha*, *real* d)	returns the line created by a translation of l in direction α by distance d.
line	l.rotate(*point* q, *real* α)	returns the line created by a rotation of l about point q by angle α.
segment	l.perpendicular(*point* p)	returns the nromal of p with respect to l.

3. Operators

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

6.1.4 Polygons (polygon)

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.

1. Creation of a polygon

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 assings it to P.

2. Operations on a polygon P

list(*point*)	P.vertices()	returns the vertex sequence of P.
list(*segment*)	P.segments()	returns the sequence of bounding segments of Pin 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(*real* α, *real* d)	returns the polygon created by a translation of of P in direction α by distance d
polygon	P.rotate(*point* q, *real* α)	returns the polygon created by a rotation of P about point q by angle α.
real	P.size()	returns the size of P.
real	P.empty()	returns true if P is empty, false otherwise.

3. Operators

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

6.1.5 Circles (circle)

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.

1. Creation of a circle

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

b) *circle* C;

introduces a variable C of type *circle*. C is initialized to the circle with center p and radius r. Variant b) creates the empty circle and assigns it to C.

2. Operations on a circle C

real	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 $[p, C.center()]$.
real	C.distance(*point* p)	returns the distance between C and p (negative if p inside C).
real	C.distance(*line* l)	returns the distance between C and l (negative if l intersects C).
real	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(*real* α, *real* d)	returns the circle created by a translation of C in direction α by distance d.
circle	C.rotate(*point* q, *real* α)	returns the circle created by a rotation of C about point q by angle α.

3. Operators

circle&	*circle* = *circle*	assignment
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.

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 segments.

• **Voronoi Diagrams**

```
void  VORONOI(list(point)&  sites,
              real  R,
              graph&  G,
              node_array(point)&  P,
              edge_array(point)&  C );
```

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. Node_array P stores for each node of G the corresponding Voronoi vertex (point) and edge_array C gives for each edge e of G the site (point) whose Voronoi region is bound by e.

6.1.7 Predefined parameterized data types

list(point), *list(segment)*, *list(line)*, *list(polygon)*, *list(circle)*

GRAPH(point, int), *node_array(point)*, *edge_array(point)*

6.2 Two-dimensional dictionaries (d2_dictionary)

An instance D of the data type *d2_dictionary* is a collection of items (*dic2_item*). Every item in D contains a key from a linearly ordered data type $K1$, a key from a linearly ordered data type $K2$, and an information from a 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 $K1 \times K2$.

1. Declaration of a two-dimensional dictionary type

declare2(*d2_dictionary*, $K1, K2, I$)

introduces a new data type with name *d2_dictionary*$(K1, K2, I)$ consisting of all two-dimensional dictionaries with key types $K1$ and $K2$ and information type I. *Precondition*: $K1$ and $K2$ are linearly ordered.

2. Creation of a two-dimensional dictionary

d2_dictionary$(K1, K2, I)$ D;

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

3. Operations on a d2_dictionary D

$K1$	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 $a \leq k_1 \leq b$ and $c \leq k_2 \leq d$.
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. Iteration

forall_dic2_items(i, D) { "the items of D are successively assigned to i " }

5. Implementation

Two-dimensional dictionaries are implemented by dynamic two-dimensional range trees 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)

An instance S of the data type *point_set* 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 a 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.

1. Declaration of a two-dimensional point set type

declare(*point_set*, I)

introduces a new data type with name *point_set*(I) consisting of all two-dimensional point sets with information type I.

2. Creation of a two-dimensional point set

point_set(I) S;

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

3. Operations on a point_set S

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).
list(*ps_item*)	S.range_search(*real* x_0, *real* x_1, *real* y_0, *real* y_1)	returns all items $< p, i > \in S$ with $x_0 \leq p$.xcoord() $\leq x_1$ and $y_0 \leq p$.ycoord() $\leq y_1$

ps_item	S.nearest_neighbor(*point q*)	returns the item $< p, i > \in S$ such that the distance between p and q is minimal.
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. Iteration

forall_ps_items(i, S) { "the items of S are successively assigned to i " }

5. Implementation

Point sets are implemented by a combination of two-dimensional range trees 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)

An instance S of the data type *interval_set* 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 a 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 $< x, y, i > \in S$.

1. Declaration of an interval set type

declare(*interval_set*, I)

introduces a new data type with name *interval_set*(I) consisting of all interval sets with information type I.

2. Creation of an interval set

interval_set(I) S;

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

3. Operations on a interval_set S

real	S.left(*is_item it*)	returns the left boundary of item *it*. *Precondition*: *it* is an item in S.
real	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(*real* x, *real* 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(*real* x, *real* y)	returns the item with interval $[x, y]$ (nil if no such item exists in S).
list(*is_item*)	S.intersection(*real* a, *real* b)	returns all items $< x, y, i > \in S$ with $[x, y] \cap [a, b] \neq \emptyset$.

void	S.del(*real* x, *real* 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. Iteration

forall_is_items(i, S) { "the items of S are successively assigned to i " }

5. Implementation

Interval sets are implemented by two-dimensional range trees. 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)

An instance S of the data type *segment_set* 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 a 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 $< s, i > \in S$.

1. Declaration of a segment set type

declare(*segment_set*, I)

introduces a new data type with name *segment_set*(I) consisting of all segment sets with information type I.

2. Creation of a segment set

a) *segment_set*(I) S(*real* α);

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 on a segment_set S

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. Iteration

forall_seg_items(i, S) { "the items of S are successively assigned to i " }

5. Implementation

Segment sets are implemented by dynamic segment trees based on BB[α] 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)

An instance S of the data type *subdivision* 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 a data type I, called the information type of S.

1. Declaration

declare(*subdivision*, I)

introduces a new data type with name *subdivision*(I) consisting of all planar subdivisions with information type I. *Precondition*: The data type $GRAPH$(*point*, I) has been declared before.

2. Creation of a subdivision

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 on a subdivision S

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. 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 (gwindow)

The data type *gwindow* provides an interface for the input and output of basic geometric objects in the plane (see section 5.1) through a graphic window on a SUN workstation. In the current implementation only the sunview (suntools) window system is supported, the include file is **<LEDA/sunview.h>**. Application programs must be started from a sunview (suntools) window and have to be linked with the *libP.a*, *libG.a*, *libL.a*, *suntool*, *sunwindow*, *pixrect*, and *m* libraries (see section 1.9).

An instance W of the data type *gwindow* is an iso-oriented rectangular window in the two-dimensional plane. Its size and position are defined by three real numbers: 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. W is displayed on the screen as a sunview window, initially a 800×800 pixel square positioned in the upper right corner. 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)();

1. Creation of a graphic window

a) *gwindow* W(*real* x_0, *real* x_1, *real* y_0);

b) *gwindow* W(*real* x_0, *real* x_1, *real* y_0, *int* d);

c) *gwindow* W;

creates a graphic window W with lower left corner (x_0, y_0) and lower right corner (x_1, y_0) . Variant b) takes an additional integer argument d to define a rectangular grid with integer coordinates of distance d. In this case the mouse cursor can only take grid point positions. Variant c) initializes W to a default sized window ($x_0 = 0$, $x_1 = 100$, $y_0 = 0$). The *init* operation (see below) can always be used to change the window coordinates.

2. Operations

2.1 Initialization

void	W.init(*real* x_0, *real* x_1, *real* y_0)	W is made a gwindow with lower left corner (x_0, y_0) and lower right corner (x_1, y_0) (like creation a).
void	W.init(*real* x_0, *real* x_1, *real* y_0, *int* d)	W is made a gwindow with lower left corner (x_0, y_0) and lower right corner (x_1, y_0) with a rectangular grid with integer coordinates of distance d (like creation b).
void	W.clear()	W is erased.

2.2 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.

draw_mode *W*.set_mode(*draw_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*.

2.3 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.
draw_mode	*W*.get_text_mode()	returns the current text mode.
draw_mode	*W*.get_mode()	returns the current drawing mode.
real	*W*.xmin()	returns x_0, the minimal x-coordinate of *W*.
real	*W*.ymin()	returns y_0, the minimal y-coordinate of *W*.
real	*W*.xmax()	returns x_1, the maximal x-coordinate of *W*.
real	*W*.ymax()	returns y_1, the maximal y-coordinate of *W*.
real	*W*.scale()	returns the number of pixels of a unit length line segment.

2.4 Drawing points

void *W*.draw_point(*real x*, *real 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()).

void *W*.draw(*point p*, $c = black$)

same as draw_point(*p*,*c*).

2.5 Drawing line segments

void W.draw_segment(*real* x_1, *real* y_1, *real* x_2, *real* 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.

void W.draw(*segment* s, $c = black$)
same as draw_segment(s,c).

2.6 Drawing lines

void W.draw_line(*real* x_1, *real* y_1, *real* x_2, *real* 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_hline(*real* y, *color* $c = black$)
draws a horizontal line with y-coordinate y.

void W.draw_vline(*real* x, *color* $c = black$)
draws a vertical line with x-coordinate x.

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

void W.draw(*line* l, $c = black$)
same as draw_line(l,c).

2.7 Drawing arrows

void W.draw_arrow(*real* x_1, *real* y_1, *real* x_2, *real* 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().

2.8 Drawing circles

void W.draw_circle(*real* x, *real* y, *real* r, *color* $c = black$)
draws the circle with center (x, y) and radius r.

void W.draw_circle(*point* p, *real* 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.

void W.draw(*circle* C, $c = black$)
same as draw_circle(C,c).

2.9 Drawing discs

void W.draw_disc(*real* x, *real* y, *real* r, *color* $c = black$)
draws a filled circle with center (x, y) and radius r.

void W.draw_disc(*point* p, *real* 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.

2.10 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(*polygon* P, $c = black$)
same as draw_polygon(P,c).

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.

2.11 Drawing functions

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

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

2.12 Drawing text

void W.draw_text(*real* x, *real* 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(*real* x, *real* 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.

2.13 Drawing nodes

Nodes are circles of diameter *node_width*.

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

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

draws a node filled with string s at position p.

2.14 Drawing edges

Edges are straigth line segments or arrows with a clearance of *node_width*/2 at each end.

void W.draw_edge(*real* x_1, *real* y_1, *real* x_2, *real* 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(*real* x_1, *real* y_1, *real* x_2, *real* 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().

2.15 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(*real&* x, *real&* 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(*real* x_0, *real* y_0, *real&* x, *real&* 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(*real* x_0, *real* y_0, *real*& x, *real*& 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(*real* x_0, *real* y_0, *real*& x, *real*& 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".

bool W.acknowledge(*string* s)
displays string s and asks for acknowledgement.

bool W.message(*string* s) displays s (each call adds a new line).

bool W.del_message() deletes the text written by all previous message operations.

2.16 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 gwindows have an internal flag 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 istream input by the eof-character ctrl-D). In conditional statements objects of type gwindow are automatically converted to boolean by simply returning this internal flag. Thus, they can be used in conditional statements exactly 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) { ... } ".

2.16.1 Output

gwindow&	W << *point* p	like W.draw_point(p)
gwindow&	W << *segment* s	like W.draw_segment(s)
gwindow&	W << *line* l	like W.draw_line(l)

gwindow&	$W <<$ *circle* C	like W.draw_circle(C)
gwindow&	$W <<$ *polygon* P	like W.draw_polygon(P)

2.16.2 Input

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

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 File input streams (file_istream)

An instance I of the data type *file_istream* is an C++ istream bound to a file F, i.e., all input operations or operators applied to I read from F.

1. Creation of a file input stream

file_istream   *in*(*string s*);

creates an instance *in* of type file_istream bound to the file with name s.

2. Operations

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

7.2 File output streams (file_ostream)

An instance O of the data type *file_ostream* is an C++ ostream bound to a file F, i.e., all output operations or operators applied to O write to F.

1. Creation of a file output stream

file_ostream   *out*(*string s*);

creates an instance *out* of type file_ostream bound to the file with name s.

2. Operations

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

7.3 Some useful functions

int	read_int(string s = "")	prints s and reads an integer
char	read_char(string s = "")	prints s and reads a character
real	read_real(string s = "")	prints s and reads a real number
string	read_string(string s = "")	prints s and reads a string
bool	Yes(string s = "")	returns (read_char(s) == 'y')
void	init_random()	initializes the random number generator.
real	random()	returns a real valued random number in $[0,1]$
int	random(int a, int b)	returns a random integer in $[a..b]$
real	used_time()	returns the currently used cpu time in seconds.
real	used_time(real& 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

7.4 Macros

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)	$((a > b)\ ?\ b\ :\ a)$

7.5 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 statement *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>


declare(stack,node)

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>

declare(queue,node)

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:

node_array(int) ord;

int cmp_node_ord(node& v, node& w)
{ return ord[v] - ord[w]; }

int cmp_edge_ord(edge& e1, edge& e2)
{ return cmp_node_ord(target(e1),target(e2)); }

bool TOPSORT1(graph& G)
{
   ord.init(G);
   if (TOPSORT(G,ord))
   { G.sort_nodes(cmp_node_ord);
     G.sort_edges(cmp_edge_ord);
     return true;
    }
   return false;
}
```

Strongly Connected Components

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

declare(array,node)

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>

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.delete_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>

declare(queue,node)

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(real)& cost,
                                              node_matrix(real)& 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;
  real 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(real) dist1(G);
  node_array(edge) pred(G);
  edge_array(real) 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(real) 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>

edge_array(real)* C;

int cmp_edges(edge& e1, edge& e2)
{ return (*C)[e1] - (*C)[e2]; }

void MIN_SPANNING_TREE(graph& G, edge_array(real)& cost, list(edge)& EL)
{
   node v, w;
   edge e;
   node_partition Q(G);
   list(edge) OEL = G.all_edges();
   C = &cost;
   OEL.sort(cmp_edges);
   EL.clear();
   forall(e, OEL)
      { v = G.source(e);
        w = G.target(e);
        if (!(Q.same_block(v, w))
          { Q.union_blocks(v, w);
            EL.append(e);
           }
      }
 }
```

9. Tables

9.1 Data Types

9.2 Algorithms

References

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

[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 IEEE Symposium on Foundations of Computer Science, 1988

[FT84] M.L. Fredman, and R.E. Tarjan: "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms", 25th Annual IEEE Symp. on Found. of Comp. Sci., 338-346, 1984

[L89] S.B. Lippman: "C++ Primer", Addison-Wesley, Publishing Company, 1989

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

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

[S86] B. Stroustrup: " The C++ Programming Language", Addison-Wesley Publishing Company, 1986

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