Contents

2 Foundations

We discuss the foundations of the LEDA system. We introduce some key concepts, such as type, object, variable, value, item, and linear order, we relate these concepts to our implementation base C++, and we put forth our major design decisions. A superficial knowledge of this chapter suffices for a first use of LEDA. We recommend that you read it quickly and come back to it as needed.

The chapter is structured as follows. We first discuss the specification of data types. Then we treat the concept "copy of an object" and its relation to assignment and parameter passing by value. The other kinds of parameter passing come next and sections on iteration statements follow. We then tie data types to the class mechanism of C++. Type parameters, linear orders, equality, hashed types, and implementation parameters are the topics of the next sections. Finally, we discuss some helpful small functions, management, error handling, header and implementation files, compilation flags, and program checking.

2.1 Data Types

The most important concept is that of a *data type* or simply *type*. A type T consists of a set of *values*, which we denote $val(T)$, a set of *objects*, which we denote $obj(T)$, and a set of functions that can be applied to the objects of the type. An object may or may not have a *name*. A named object is also called a *variable* and an object without a name is called an *anonymous object*. An object is a region of storage that can hold a value of the corresponding type.

The set of objects of a type varies during execution of a program. It is initially empty, it grows as new objects are created (either by variable definitions or by applications of the *new* operator), and it shrinks as objects are destroyed.

The values of a type form a set that exists independently of any program execution. We define it using standard mathematical concepts and notation. When we refer to the values of a type without reference to an object, we also use *element* or *instance*, e.g., we say that the number 5 is a value, an element, or an instance of type *int*.

An object always holds a value of the appropriate type. The object is initialized when it is created and the value may be modified by functions operating on the object. For an object x we use x also to denote the value of x. This is a misuse of notation to which every programmer is accustomed to.

In LEDA the specification (also called definition) of a data type consists of four parts: a definition of the instances of the type, a description of how to create an object of the type, the definition of the operations available on the objects of the type, and information about the implementation. In the LEDA manual the four parts appear under the headers *Definition*, *Creation*, *Operations*, and *Implementation*, respectively. Sometimes, there is also a fifth section illustrating the use of the data type by an example. As an example we give the complete specification of the parameterized data type *stack*<E> in Figure 2.1.

2.1.1 ***Definition***

The first section of a specification defines the instances of the data type using standard mathematical concepts and notation. It also introduces notation that is used in later sections of the specification. We give some examples:

- An instance of type *string* is a finite sequence of characters. The length of the sequence is called the *length* of the string.
- An instance of type *stack*<E> is a sequence of elements of type E. One end of the sequence is designated as its *top* and all insertions into and deletions from a stack take place at its top end. The length of the sequence is called the *size* of the stack. A stack of size zero is called *empty*.
- An instance of type *array*<E> is an injective mapping from an interval $I = [a\,..\,b]$ of integers into the set of variables of type E. We call I the index set and E the element type of the array. For an array A we use $A(i)$ to denote the variable indexed by i, $a \le i \le b$.
- An instance of type *set*<E> is a set of elements of type E. We call E the element type of the set; E must be linearly ordered. The number of elements in the set is called the *size* of the set and a set of size zero is called *empty*.
- An instance of type *list*<E> is a sequence of list items (predefined item type *list_item*). Each item contains an element of type E. We use $\langle x \rangle$ to denote an item with content x.

Most data types in LEDA are *parameterized*, e.g., stacks, arrays, lists, and sets can be used for an arbitrary element type E and we will later see that dictionaries are defined in terms of a key type and an information type. A concrete type is obtained from a parameterized

Stacks (stack)

1. Definition

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

2. Creation

stack<E> S;	declares a variable S of type *stack*<E>. S is initialized with the empty stack.

3. Operations

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

4. Implementation

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

Figure 2.1 The specification of the type *stack*<E>.

type by substituting concrete types for the type parameter(s); this process is called *instantiation of the parameterized type*. So *array*<*string*> are arrays of strings, *set*<*int*> are sets of integers, and *stack*<*set*<*int*> $*$ > are stacks of pointers to sets of integers. Frequently, the actual type parameters have to fulfill certain conditions, e.g., the element type of sets must be linearly ordered. We discuss type parameters in detail in Section 2.8.

2.1.2 *Creation*

We discuss how objects are created and how their initial value is defined. We will see that an object either has a name or is anonymous. We will also learn how the lifetime of an object is determined.

A *named object* (also called variable) is introduced by a C++ *variable definition*. We give some examples.

```
string s;
```

introduces a variable *s* of type *string* and initializes it to the empty string.

```
stack<E> S;
```

introduces a variable *S* of type *stack*$<E>$ and initializes it to the empty stack.

```
b_stack<E> S(int n);
```

introduces a variable *S* of type *b_stack*$<E>$ and initializes it to the empty stack. The stack can hold a maximum of *n* elements.

```
set<E> S;
```

introduces a variable *S* of type *set*$<E>$ and initializes it to the empty set.

```
array<E> A(int l,int u);
```

introduces a variable *A* of type *array*$<E>$ and initializes it to an injective function $a : [l \,..\, u] \longrightarrow obj(E)$. Each object in the array is initialized by the default initialization of type E; this concept is defined below.

```
list<E> L;
```

introduces a variable *L* of type *list*$<E>$ and initializes it to the empty list.

```
int i;
```

introduces a variable of type *int* and initializes it to some value of type *int*.

We always give variable definitions in their generic form, i.e., we use formal type names for the type parameters (E in the definitions above) and formal arguments for the arguments of the definition (*int a*, *int b*, and *int n* in the definitions above). Let us also see some concrete forms.

```
string s("abc");        // initialized to "abc"
set<int> S;             // initialized to empty set of integers
array<string> A(2,5);   // array with index set [2..5],
                        // each entry is set to the empty string
b_stack<int> S(100);    // a stack capable of holding up to 100
                        // ints; initialized to the empty stack
```

The most general form of a variable definition in C++ is

```
T<T1,...,Tk> y(x1,...,xl).
```

It introduces a variable with name *y* of type *T<T1, ..., Tk>* and uses arguments *x1, ..., xl* to determine the initial value of *y*. Here *T* is a parameterized type with *k* type parameters and *T1, ..., Tk* are concrete types. If any of the parameter lists is empty the corresponding pair of brackets is to be omitted.

Two kinds of variable definitions are of particular importance: the definition with default initialization and the definition with initialization by copying. A *definition with default initialization* takes no argument and initializes the variable with the *default value* of the type. The default value is typically the “simplest” value of the type, e.g., the empty string, the empty set, the empty dictionary, ... We define the default value of a type in the section with header Creation. Examples are:

```
string s;         // initialized to the empty string
stack<int> S;     // initialized to the empty stack
array<string> A;  // initialized to the array with empty index set
```

The built-in types such as *char*, *int*, *float*, *double*, and all pointer types are somewhat an exception as they have no default value, e.g., the definition of an integer variable initializes it with some integer value. This value may depend on the execution history. Some compilers will initialize *i* to zero (more generally, 0 casted to the built-in type in question), but one should not rely on this[1].

We can now also explain the definition of an array. Each variable of the array is initialized by the default initialization of the element type. If the element type has a default value (as is true for all LEDA types), this value is taken and if it has no default value (as is true for all built-in types), some value is taken. For example, *array<list<E> > A*(1, 2) defines *A* as an array of lists of element type *E*. Each entry of the array is initialized with the empty list.

A *definition with initialization by copying* takes a single argument of the same type and initializes the variable with a copy of the argument. The syntactic form is

```
T<T1,...,Tk> y(x)
```

where *x* refers to a value of type *T<T1, ..., Tk>*, i.e., *x* is either a variable name or more generally an expression of type *T<T1, ..., Tk>*. An alternative syntactic format is

```
T<T1,...,Tk> y = x.
```

We give some examples.

```
stack<int> P(S);    // initialized to a copy of S
set<string> U(V);   // initialized to a copy of V
string s = t;       // initialized to a copy of t
int i = j;          // initialized to a copy of j
int h = 5;          // initialized to a copy of 5
```

We have to postpone the general definition of what constitutes a copy to Section 2.3 and give only some examples here. A copy of an integer is the integer itself and a copy of a string is the string itself. A copy of an array is an array with the same index set but new variables. The initial values of the new variables are copies of the values of the corresponding old variables.

LEDA Rule 1 *Definition with initialization by copying is available for every LEDA type. It initializes the defined variable with a copy of the argument of the definition.*

[1] The C++ standard defines that variables specified static are automatically zero-initialized and that variables specified automatic or register are not guaranteed to be initialized to a specified value.

How long does a variable live? The *lifetime* of a named variable is either tied to the block containing its definition (this is the default rule) or is the execution of the entire program (if the variable is explicitly defined to be static). The first kind of variable is called *automatic* in C++ and the second kind is called *static*. Automatic variables are created and initialized each time the flow of control reaches their definition and destroyed on exit from their block. Static variables are created and initialized when the program execution starts and destroyed when the program execution ends.

We turn to *anonymous objects* next. They are created by the operator *new*; the operator returns a pointer to the newly created object. The general syntactic format is

```
new T<T1,...,Tk> (x1,...,xl);
```

where T is a parameterized type, *T1*, ..., *Tk* are concrete types, and *x1*, ..., *xl* are the arguments for the initialization. Again, if any of the argument lists is empty then the corresponding pair of brackets is omitted. The expression returns a pointer to a new object of type T<*T1*, ..., *Tk*>. The object is initialized as determined by the arguments *x1*, ..., *xl*. We give an example.

```
stack<int> *sp = new stack<int>;
```

defines a pointer variable *sp* and creates an anonymous object of type *stack*<*int*>. The stack is initialized to the empty stack and *sp* is initialized to a pointer to this stack.

The lifetime of an object created by *new* is not restricted to the scope in which it is created. It extends till the end of the execution of the program unless the object is explicitly destroyed by the *delete* operator; *delete* can only be applied to pointers returned by *new* and if it is applied to such a pointer, it destroys the object pointed to. We say more about the destruction of objects in Section 2.3.

2.1.3 *Operations*

Every type comes with a set of operations that can be applied to the objects of the type. The definition of an operation consists of two parts: the definition of its interface (= syntax) and the definition of its effect (= semantics).

We specify the *interface of an operation* essentially by means of the C++ function declaration syntax. In this syntax the result type of the operation is followed by the operation name which in turn is followed by the argument list specifying the type of each argument. The result type of an operation returning no result is *void*. We extend this syntax by prefixing the operation name by the name of an object to which the operation is being applied. This facilitates the definition of the semantics. For example

```
void S.insert(E x);
```

defines the interface of the insert operation for type *set*<E>; *insert* takes an argument x of type E and returns no result. The operation is applied to the set (with name) S.

```
E& A[int i];
```

defines the interface of the access operation for type *array*<E>. Access takes an argument i of type *int* and returns a variable of type E. The operation is applied to array A.

```
E S.pop();
```

defines the interface of the pop operation for type *stack*<*E*>. It takes no argument and returns an element of type E. The operation is applied to stack S.

```
int s.pos(string s1);
```

defines the interface of the *pos* operation for type *string*. It takes an argument *s1* of type *string* and returns an integer. The operation is applied to string s.

The *semantics of an operation* is defined using standard mathematical concepts and notation. The complete definitions of our four example operations are:

void	*S.insert*(E x)	adds x to S.
E&	*A*[*int* i]	returns the variable $A(i)$. *Precondition*: $a \leq i \leq b$.
E	*S.pop*( )	removes and returns the top element of S. *Precondition*: S is not empty.
int	*s.pos*(*string s1*)	returns -1 if *s1* is not a substring of s and returns the minimal i, $0 \leq i \leq$*s.length*( )-1, such that *s1* occurs as a substring of s starting at position i, otherwise.

In the definition of the semantics we make use of the notation introduced in sections Definition and Creation. For example, in the case of arrays the section Definition introduces $A(i)$ as the notation for the variable indexed by i and introduces a and b as the array bounds.

Frequently, an operation is only defined for a subset of all possible arguments, e.g., the *pop* operation on stacks can only be applied to a non-empty stack. The *precondition* of an operation defines which conditions the arguments of an operation must satisfy. If the precondition of an operation is violated then the effect of the operation is undefined. This means that *everything can happen*. The operation may terminate with an error message or with an arbitrary result, it may not terminate at all, or it may result in abnormal termination of the program. Does LEDA check preconditions? Sometimes it does and sometimes it does not. For example, we check whether an array index is out of bounds or whether a pop from an empty stack is attempted, but we do not check whether item *it* belongs to dictionary D in *D.inf*(*it*). Checking the latter condition would increase the running time of the operation form constant to logarithmic and is therefore not done. More generally, we do not check preconditions that would change the order of the running time of an operation. All checks can be turned off by the compile-time flag `-DLEDA_CHECKING_OFF`.

All types offer the assignment operator. For type T this is the operator

```
T& operator=(const T&).
```

The assignment operator is not listed under the operations of a type since all types have it and since its semantics is defined in a uniform way as we will see in Section 2.3.

Our implementation base C++ allows overloading of operation and function names and it allows optional arguments. We use both mechanisms. An *overloaded function name* denotes different functions depending on the types of the arguments. For example, we have two translate operations for points:

```
point p.translate(vector v);
point p.translate(double alpha,double dist);
```

The first operation translates p by vector v and the second operation translates p in direction *alpha* by distance *dist*.

An *optional argument* of an operation is given a default value in the specification of the operation. C++ allows only trailing arguments to be optional, i.e., if an operation has k arguments, $k \geq 1$, then the last l, $l \geq 0$, may be specified to be optional. An example is the insert operation into lists. If L is a *list*<E> then

```
list_item L.insert(E x,list_item it, int dir = after)
```

inserts x before (*dir* $==$ *before*) or after (*dir* $==$ *after*) item *it* into L. The default value of *dir* is *after*, i.e., $L.insert(x, it)$ is equivalent to $L.insert(x, it, after)$.

2.1.4 ***Implementation***

Under this header we give information about the implementation of the data type. We name the data structure used, give a reference, list the *running time* of the operations, and state the *space requirement*. Here is an example.

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 $O(n)$, $item(i)$ takes time $O(i)$, *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. The space requirement is $16 + 12n$ bytes.

It should be noted that the time bounds do not include the time needed for parameter passing. The cost of passing a reference parameter is bounded by a constant and the cost of passing a value parameter is the cost of copying the argument. We follow the custom to account for parameter passing at the place of call.

Similarly, the space bound does not include the extra space needed for the elements contained in the set, it only accounts for the space required by the data structure that realizes the set. The extra space needed for an element is zero if the element fits into one machine word and is the space requirement of the element otherwise. This reflects how parameterized data types are implemented in LEDA. Values that fit exactly into one word are stored directly in the data structure and values that do not fit exactly are stored indirectly through a pointer. The details are given in Section 13.1.

The information about the space complexity allows us to compute the exact space requirement of a list of size n. We give some examples. A set of type *list*<*int*> and *list*<*list*<*int*> $*$ >

requires $16 + 12n$ bytes since integers and pointers fit exactly into a word. A list of type *list<list<int> >* where the i-th list has n_i elements, $1 \le i \le n$, requires $16 + 12n + \sum_{1 \le i \le n}(16 + 12n_i)$ bytes.

The information about time complexity is less specific than that for space. We only give *asymptotic bounds*, i.e., bounds of the form $O(f(n))$ where f is a function of n. A bound of this form means that there are constants c_1 and c_2 (independent of n) such that the running time on an instance of size n is bounded by $c_1 + c_2 \cdot f(n)$. The constants c_1 and c_2 are not explicitely given. An asymptotic bound does predict the actual running time on a particular input (as c_1 and c_2 are not available), it gives however a feeling for the behavior of an algorithm as n grows. In particular, if the running time is $O(n)$ then an input of twice the size requires at most twice the computing time, if the running time is $O(n^2)$ then the computing time at most quadruples, and if it is $O(\log n)$ then the computing time grows only by an additive constant as n doubles. Thus asymptotic bounds allow us to extrapolate running times from smaller to larger problem instances.

Why do we not give explicit values for the constants c_1 and c_2? The answer is simple, we do not know them. They depend on the machine and compiler which you use (which we do not know) and even for a fixed machine and compiler it is very difficult to determine them, as machines and compilers are complex objects with complex behavior, e.g., machines have pipelines, multilevel memories, and compilers use sophisticated optimization strategies. It is conceivable that program analysis combined with a set of simple experiments allows one to determine good approximations of the constants, see [FM97] for a first step in this direction.

Our usual notion of running time is worst-case running time, i.e., if an operation is said to have running time $O(f(n))$ then it is guaranteed that the running time is bounded by $c_1 + c_2 \cdot f(n)$ for every input of size n and some constants c_1 and c_2. Sometimes, running times are classified as being expected (also called average) or amortized. We give some examples.

The expected access time for maps is constant. This assumes that a random set is stored in the map.

The expected time to construct the convex hull of n points in 3-dimensional space is $O(n \log n)$. The algorithm is randomized.

The amortized running time of *insert* and *decrease_prio* in priority queues is constant and the amortized running time of *delete_min* is $O(\log n)$.

In the remainder of this section we explain the terms expected and amortized. An *amortized* time bound is valid for a sequence of operations but not for an individual operation. More precisely, assume that we execute a sequence op_1, op_2, $\ldots$, op_m of operations on an object D, where op_1 constructs D. Let n_i be the size of D before the i-th operation and assume that the i-th operation has amortized cost $O(T_i(n_i))$. Then the total running time for the sequence op_1, op_2, $\ldots$,op_m is

$$O(m + \sum_{1 \le i \le m} T_i(n_i)),$$

i.e., a summation of the amortized time bounds for the individual operations yields a bound for the sequence of the operations. Note that this does not preclude that the i-th operation takes much longer than $T_i(n_i)$ for some i, it only states that the entire sequence runs in the bound stated. However, if the i-th operation takes longer than $T_i(n_i)$ then the preceding operations took less than their allowed time.

We give an example: in priority queues (with the Fibonacci heap implementation) the amortized running time of *insert* and *decrease_prio* is constant and the amortized cost of *delete_min* is $O(\log n)$. Thus an arbitrary sequence of n *insert*, n *delete_min*, and m *decrease_prio* operations takes time $O(m + n \log n)$.

We turn to *expected* running times next. There are two ways to compute expected running times. Either one postulates a probability distribution on the inputs or the algorithm is randomized, i.e., uses random choices internally.

Assume first that we have a probability distribution on the inputs, i.e., if x is any conceivable input of size n then $prob(x)$ is the probability that x actually occurs as an input. The expected running time $\bar{T}(n)$ is computed as a weighted sum $\bar{T}(n) = \sum_x prob(x) \cdot T(x)$, where x ranges over all inputs of size n and $T(x)$ denotes the running time on input x. We refer the reader to any of the textbooks [AHU83, CLR90, Meh84] for a more detailed treatment. We usually assume the *uniform distribution*, i.e., if x and y are two inputs of the same size then $prob(x) = prob(y)$. It is time for an example.

The expected access time for maps is constant. A *map*<I, E> realizes a partial function m from some type I to some other type E; the index type I must be either the type *int* or a pointer or item type. Let D be the domain of m, i.e., the set of arguments for which m is defined. The uniform distribution assumption is then that all subsets D of I of size n are equally likely. The average running time is computed with respect to this distribution.

Two words of caution are in order at this point. Small average running time does not preclude the possibility of outliers, i.e., inputs for which the actual running time exceeds the average running time by a large amount. Also, average running time is stated with respect to a particular probability distribution on the inputs. This distribution is probably not the distribution from which your inputs are drawn. So be careful.

A *randomized* algorithm uses random choices to control its execution. For example, one of our convex hull algorithms takes as input a set of points in the plane, permutes the points randomly, and then computes the hull in an incremental fashion. The running time and maybe also the output of a randomized algorithm depends on the random choices made. Averaging over the random choices yields the expected running time of the algorithm. Note that we are only averaging with respect to the random choices made by the algorithm, and do not average with respect to inputs. In fact, time bound of randomized algorithms are worst-case with respect to inputs. As of this writing all randomized algorithms in LEDA are of the so-called *Las Vegas* style, i.e., their output is independent of the random choices made. For example, the convex hull algorithm always computes the convex hull. If the output of a randomized algorithm depends on the random choices then the algorithm is called *Monte Carlo* style. An example of a Monte Carlo style randomized algorithm is the primality tests of Solovay and Strassen [SS77] and Rabin [Rab80]. They take two integers n

and s and test the primality of n. If the algorithms declare n non-prime then n is non-prime. If they declare n prime then this answer is correct with probability at least $1-2^{-s}$, i.e., there is chance that the answer is incorrect. However, this chance is miniscule (less than 2^{-100} for $s = 100$). The expected running time is $O(s \log^3 n)$.

2.2 Item Types

Item types are ubiquitous in LEDA. We have dic_items (= items in dictionaries), pq_items (= items in priority queues), nodes and edges (= items in graphs), points, segments, and lines (= basic geometric items), and many others. What is an item?

Items are simply addresses of containers and item variables are variables that can store items. In other words, item types are essentially C++ pointer types. We say essentially, because some item types are not implemented as pointer types. We come back to this point below.

A (value of type) *dic_item* is the address of a dic_container and a (value of type) *point* is the address of a point_container. A dic_container has a key and an information field and additional fields that are needed for the data structure underlying the dictionary and a point_container has fields for the x- and y-coordinate and additional fields for internal use. In C++ notation we have as a first approximation (the details are different):

```
class dic_container
{ K key;
  I inf;
  // additional fields required for the underlying data structure
}
typedef dic_container* dic_item;
class point_container
{ double x, y;
  // additional fields required for internal use
}
typedef point_container* point;
   // Warning: this is NOT the actual definition of point
```

We distinguish between *dependent* and *independent* item types. The containers corresponding to a dependent item type can only live as part of a collection of containers, e.g., a dictionary-container can only exist as part of a dictionary, a priority-queue-container can only exists as part of a priority queue, and a node-container can only exists as part of a graph. A container of an independent item type is self-sufficient and needs no “parent type” to justify its existence. Points, segments, and lines are examples of independent item types. We discuss the common properties of all item types now and treat the special properties of dependent and independent item types afterwards. We call an item of an independent or dependent item type an independent or dependent item, respectively.

An item is the address of a container. We refer to the values stored in the container as *attributes* of the item, e.g., a point has an x- and a y-coordinate and a dic_item has a key and an information. We have functions that allow us to read the attributes of an item. For a point p, *p.xcoord*( ) returns the x-coordinate of the point, for a segment s, *s.start*( ) returns the start point of the segment, and for a dic_item *it* which is part of a dictionary D, *D.key*(*it*) returns the key of the item. Note the syntactic difference: for dependent items the parent object is the main argument of the access function and for independent items the item itself is the main argument.

We will systematically blur the distinction between items and containers. The previous paragraph was the first step. We write "a point has an x-coordinate" instead of the more verbose "a point refers to a container which stores an x-coordinate" and "a dic_item has a key" instead of the more verbose "a dic_item refers to a container that stores a key". We also say "a dic_item which is part of a dictionary D" instead of the more verbose "a dic_item that refers to a container that is part of a dictionary D". We will see more examples below. For example, we say that an insert *D.insert*(k, i) into a dictionary "adds an item with key k and information i to the dictionary and returns it" instead of the more verbose "adds a container with key k and information i to the dictionary and returns the address of the container". Our shorthand makes many statements shorter and easier to read but can sometimes cause confusion. Going back to the longhand should always resolve the confusion.

We said above that item types are essentially C++ pointer types. The actual implementation may be different and frequently is. In the current implementation of LEDA all dependent item types are realized directly as pointer types, e.g., the type *dic_item* is defined as *dic_container*$*$, and all independent item types are realized as classes whose only data member is a pointer to the corresponding container class.

The reason for the distinction is storage management which is harder for containers associated with independent item types. For example, a dictionary-container can be returned to free store precisely if it is either deleted from the dictionary containing it or if the lifetime of the dictionary containing it ends. Both situations are easily recognized. On the other hand, a point-container can be returned to free store if no point points to it anymore. In order to recognize this situation we make every point-container know how many points point to it. This is called a reference count. The count is updated by the operations on points, e.g., an assignment $p = q$ increases the count of the container pointed to by q and decreases the count of the container pointed to by p. When the count of a container reaches zero it can be returned to free store. In order to make all of this transparent to the user of type *point* it is necessary to encapsulate the pointer in a class and to redefine the pointer operations assignment and access. This technique is known under the name *handle types* and is discussed in detail in Section 13.7.

All item types offer the assignment operator and the equality predicate. Assume that T is an item type and that *it1* and *it2* are variables of type T. The assignment

```
it1 = it2;
```

assigns the value of *it2* to *it1* and returns a reference to *it1*. This is simply the assignment between pointers. In the case of handle types the assignment has the side effect of updating the reference counters of the objects pointed to by *it1* and *it2*.

The equality predicate (operator *bool operator* $==$ (*const* T&, *const* T&)) is more subtle. For dependent item types it is the equality between values (i.e., pointers) but for independent item types it is usually defined differently. For example, two points in the Euclidean plane are equal if they agree in their Euclidean coordinates.

```
point p(2.0,3.0);  // a point with coordinates 2.0 and 3.0
point q(2.0,3.0);  // another point with the same coordinates
p == q;            // evaluates to true
```

Note that p and q are not equal as pointers. They point to distinct point-containers. However, they agree in their Euclidean coordinates and therefore the two points are said to be equal. For independent item types we also have the *identity* predicate (realized by function *bool identical*(*const* T&, *const* T&)). It tests for equality of values (i.e., pointers). Thus *identical*(p, q) evaluates to false. We summarize in:

LEDA Rule 2

(a) *For independent item types the identity predicate is equality between values. The equality predicate is defined individually for each item type. It is usually equality between attributes.*

(b) *For dependent item types the equality predicate is equality between values.*

2.2.1 *Dependent Item Types*

Many advanced data types in LEDA are defined as collections of items, e.g., a dictionary is a collection of dic_items and a graph is defined in terms of nodes and edges. This collection usually has some combinatorial structure imposed on it, e.g., it may be arranged in the form of a sequence, or in the form of a tree, or in the form of a general graph. We give some examples.

An instance of type *dictionary*<K, I> is a collection of dic_items, each of which has an associated key of type K and an associated information of type I. The keys of distinct items are distinct. We use $\langle k, i\rangle$ to denote an item with key k and information i.

An instance of type *list*<E> is a sequence of list_items, each of which has an associated information of type E. We use $\langle e\rangle$ to denote an item with information e.

An instance of type *sortseq*<K, I> is a sequence of seq_items, each of which has an associated key of type K and an associated information of type I. The key type K must be linearly ordered and the keys of the items in the sequence increase monotonically from front to rear. We use $\langle k, i\rangle$ to denote an item with key k and information i.

An instance of type *graph* is a list of nodes and a list of edges. Each edge has a source node and a target node. We use (v, w) to denote an edge with source v and target w.

An instance of type *partition* is a collection of partition_items and a partition of these items into so-called *blocks*.

In all examples above an instance of the complex data type is a collection of items. This collection has some combinatorial structure: lists and sorted sequences are sequences of items, the items of a partition are arranged into disjoint blocks, and the nodes and edges of a graph form a graph. The items have zero or more attributes: dic_items and seq_items have a key and an information, an edge has a source and a target node, whereas a partition_item has no attribute. An attribute either helps to define the combinatorial structure, as in the case of graphs, or associates additional information with an item, as in the case of dictionaries, lists, and sorted sequences. The combinatorial structure is either defined by referring to standard mathematical concepts, such as set, sequence, or tree, or by using attributes, e.g., an edge has a source and a target. The values of the attributes belong to certain types; these types are usually type parameters. The type parameters and the attribute values may have to fulfill certain constraints, e.g., sorted sequences require their key type to be linearly ordered, dictionaries require the keys of distinct items to be distinct, and the keys of the items in a sorted sequence must be monotonically increasing from front to rear.

Many operations on dictionaries (and similarly, for the other complex data types of LEDA) have items in their interface, e.g., an *insert* into a dictionary returns an item, and a *change_inf* takes an item and a new value for its associated information. Why have we chosen this design which deviates from the specifications usually made in data structure text books? The main reason is efficiency.

Consider the following popular alternative. It 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 memory. It is frequently useful that the pair (k, i) cannot only be accessed through the key k but also through the location where it is stored, e.g., we may 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, it can be done by direct access. Of course, the second solution is more efficient and we therefore wanted to support it in LEDA.

We provide direct access through dic_items. A dic_item is the address of a dictionary container and can be stored in a dic_item variable. The key and information stored in a dictionary container can be accessed directly through a dic_item variable.

Doesn't this introduce all the dangers of pointers, e.g., the potential to change information which is essential to the correct functioning of the underlying data structure? The answer is no, because the access to dictionary containers through dictionary items is restricted, e.g., the access to a key of a dictionary container is read-only. In this way, items give the efficiency of pointers but exclude most of their misuse, e.g., given a dic_item its associated key and information can be accessed in constant time, i.e., we have the efficiency of pointer access, but the key of a dic_item cannot be changed (as this would probably corrupt the underlying data structure), i.e., one of the dangers of pointers is avoided. The wish to have the efficiency of pointer access without its dangers was our main motivation for introducing items into the signatures of operations on complex data types.

Let us next see some operations involving items. We use dictionaries as a typical example. The operations

```
dic_item D.lookup(K k);
I        D.inf(dic_item it);
void     D.change_inf(dic_item it,I j);
```

have the following semantics: *D.lookup*(k) returns the item[2], say *it*, with key k in dictionary D, *D.inf*(it) extracts the information from *it*, and a new information j is associated with *it* by *D.change_inf*(it, j). Note that only the first operation involves a search in the data structure realizing D and that the other two operations access the item directly.

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* whose key is equal to k then the information associated with *it* is changed to i and *it* is returned. If D contains no such item, then a *new* container, i.e., a container which is not part in any dictionary, is added to D, this container is made to contain (k, i), and its address is returned. In the specification of dictionaries 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 $\langle k, j\rangle$ in D then j is replaced by i, else a new item $\langle k, i\rangle$ is added to D. In both cases the item is returned.

For any dependent item type the set of values of the type contains the special value *nil*[3]. This value never belongs to any collection and no attributes are ever defined for it. We use it frequently as the return value for function calls that fail in some sense. For example *D.lookup*(k) returns *nil* if there is no item with key k in D.

Containers corresponding to dependent item types cannot exist outside collections. Assume, for example, that the container referred to by dic_item *it* belongs to some dictionary D and is deleted from D by *D.del_item*(it). This removes the container from D and destroys it. It is now illegal[4] to access the fields of this container.

LEDA Rule 3 *It is illegal to access the attributes of an item which refers to a container that has been destroyed or to access the attributes of the item nil.*

In the definition of operations involving items this axiom frequently appears in the form of a precondition.

I	*D.inf*$(dic\_item\ it)$	returns the information of item *it*. *Precondition*: *it* must belong to dictionary D.

[2] The operation returns *nil* if there is no item with key k in D.

[3] Recall that all dependent item types are pointer types internally.

[4] Of course, as in ordinary life, illegal actions can be performed anyway. The outcome of an illegal action is hard to predict. You may be lucky and read the values that existed before the container was destroyed, or you may be unlucky and read some random value, or you might get caught and generate a segmentation fault.

2.2.2 *Independent Item Types*

We now come to independent item types. Points, lines, segments, integers, rationals, and reals are examples of independent item types. We discuss points.

A point is an item with two attributes of type double, called the x- and y-coordinate of the point, respectively[5]. We use (a, b) to denote a point with x-coordinate a and y-coordinate b.

Note that we are not saying that a point is a pair of doubles. We say: a point is an item and this item has two double attributes, namely the coordinates of the point. In other words, a point is logically a pointer to a container that contains two doubles (and additional fields for internal use). This design has several desirable implications:

- Assignment between points takes constant time. This is particularly important for types where the attributes are large, e.g., arbitrary precision integers.
- Points can be tested for identity (= same pointer value) and for equality (= same attribute values). The identity test is cheap.
- The storage management for points and all other independent item types is transparent to the LEDA user.

We have functions to query the attributes of a point: *p.xcoord*( ) returns the x-coordinate and *p.ycoord*( ) returns the y-coordinate. We also have operations to construct new points from already constructed points, e.g.,

```
point p.translate(double a,double b);
```

returns a new point $(p.xcoord() + a, \; p.ycoord() + b)$, i.e., it returns an item with attributes $p.xcoord() + a$ and $p.ycoord() + b$. It is important to note that *translate* does not change the point p. In fact, there is no operation on points that changes the attributes of an already existing point. This is true for all independent item types.

LEDA Rule 4 *Independent item types offer no operations that allow to change attributes; the attributes are immutable.*

We were led to this rule by programs of the following kind (which is not a LEDA program):

```
q = p;
p.change_x(a); // change x-coordinate of p to a
```

After the assignment q and p point to the same point-container and hence changing p's x-coordinate also changes q's x-coordinate, a dangerous side-effect that can lead to errors that are very hard to find[6]. We therefore wanted to exclude this possibility of error. We explored two alternatives. The first alternative redefines the semantics of the assignment statement to mean component-wise assignment and the second alternative forbids operations that change

[5] There are also points with rational coordinates and points in higher dimensional space.
[6] Both authors spent many hours finding errors of this kind.

attributes. We explored both alternatives in a number of example programs, adopted the second alternative[7], and casted it into the rule above.

A definition of an independent item always initializes all attributes of the item. For example,

```
point p(2.0,3.0);
point q;           // q has coordinates but it is not known which.
```

defines a point p with coordinates $(2.0, 3.0)$ and a point q. The coordinates of point q are defined but their exact value is undetermined. This is the same convention as for built-in types.

LEDA Rule 5 *The attributes of an independent item are always defined. In particular, definition with default initialization initializes all attributes. A type may specify the initial values but it does not have to.*

We explored alternatives to this rule. For example, we considered the rule that the initial value of an attribute is always the default value of the corresponding type. This rule sounds elegant but we did not adopt it because of the following example. We mentioned already that the default value of type *double* is undefined and that the default value of type *rational* is zero. Thus a point with rational coordinates (type *rat_point*) would be initialized to the origin and a point with floating point coordinates (type *point*) would be initialized to some unspecified point. This would be confusing and a source of error. The rule above helps to avoid this error by encouraging the practice that objects of an independent item type are to be initialized explicitly.

2.3 Copy, Assignment, and Value Parameters

We now come to a central concept of C++ and hence LEDA, the notion of a *copy*. Its importance stems from the fact that several other key concepts are defined in terms of it, namely assignment, creation with initialization by copying, parameter passing by value, and function value return. We give these definitions first and only afterwards define what it means to copy a value. At the end of the section we also establish a relation between destruction and copying.

We distinguish between primitive types and non-primitive types. All built-in types, all pointer types, and all item types are primitive. For primitive types the definition of a copy is trivial, for non-primitive types the definition is somewhat involved. Fortunately, most LEDA users will never feel the need to copy a non-primitive object and hence can skip the non-trivial parts of this section.

We start by defining assignment and creation with initialization by copying in terms of copying. This will also reveal a close connection between assignment and creation with

[7] This does not preclude the possibility that other examples would have led us to a different conclusion.

initialization. The designers of C++ decided that definition with initialization is defined in terms of copy and we decided that assignment should also be defined in terms of copy. Observe that C++ allows one to implement the assignment operator for a class in an arbitrary way. We decided that the assignment operator should have a uniform semantics for all LEDA types.

LEDA Rule 6 *An assignment* `x = A` *assigns a copy of the value of expression A to the variable x.*

C++ Axiom 1 *A definition* `T x = A` *creates a new variable x of type T and initializes it with a copy of the value of A. An alternative syntactic form is* `T x(A)`. *The statement* `new T(A)` *returns a pointer to a newly created anonymous object of type T. The object is initialized with a copy of the value of A.*

The axioms above imply that the code fragments `T x; x = A` and `T x = A` are equivalent, i.e., creation with default initialization followed by an assignment is equivalent to creation with initialization by copying[8]. The next axiom ties parameter passing by value and value return to definition with initialization and hence to copying.

C++ Axiom 2
a) A value parameter of type T and name x is specified as `T x`. *Let A be an actual parameter, i.e., A is an expression of type T. Parameter passing is equivalent to the definition* `T x = A`.
b) Let f be a function with return type T and let `return A` *be a return statement in the body of f; A is an expression of type T. Function value return is equivalent to the definition* `T x = A` *where x is a name invented by the compiler. x is called a temporary variable.*

Now that we have seen so many references to the notion of copy of a value, it is time to define it. A copy of a natural number is simply the number itself. More generally, this is true for all so-called *primitive* types.

LEDA Rule 7
(a) *All built-in types, all pointer types, and all item types are primitive.*
(b) *A copy of a value of a primitive type is the value itself.*

We conclude, that the primitive types behave exactly like the built-in types and hence if you understand what copy, assignment, parameter passing by value, and function value return mean for the built-in types, you also understand them for all primitive types. For non-primitive types the definition of a copy is more complex and making a copy is usually a non-constant time operation. Fortunately, the copy operation for non-primitive types is rarely needed. We give the following advice.

Advice: Avoid assignment, initialization by copying, parameter passing by value, and

[8] This assumes that both kinds of creations are defined for the type T.

function value return for non-primitive types. Also exercise care when using a non-primitive type as an actual type parameter.

```
// read on, if you plan to use any of the statements below
L1 = L2;                         // L1 and L2 are lists
int f(list<int> A);              // non-primitive value parameter
list<int> f();                   // non-primitive return value
dictionary<string,list<int> > D; // non-primitive type parameter
```

The values of non-primitive types exhibit structure, e.g., a value of type *stack*$<E>$ is a sequence of elements of type E, a value of type *array*$<E>$ is a set of variables of type E indexed by an interval of integers, and a value of type *list*$<E>$ is a sequence of list items each with an associated element of type E. Therefore, non-primitive types are also called *structured.* A copy of a value of a structured type is similar but not identical to the original in the same sense as the Xerox-copy of a piece of paper is similar but not identical to the original; it has the same content but is on a different piece of paper.

We distinguish two kinds of structured types, *item-based* and *non-item-based.* A structured type is called item-based if its values are defined as collections of items. Dictionaries, sorted sequences, and lists are examples of item-based structured types, and arrays and sets are examples of non-item-based structured types. We also say *simple-structured* type instead of non-item-based structured type.

LEDA Rule 8

(a) *A value x of a simple-structured type is a set or sequence of elements or variables of some type E. A copy of x is a component-wise copy.*

(b) *A copy of a variable is a new variable of the same type, initialized with a copy of the value of the original.*

We give some examples. Copying the stack (1, 4, 2) produces the stack (1, 4, 2), copying an *array*<*int*> with index set $[1\,..\,3]$ means creating three new integer variables indexed by the integers one to three and initializing the variables with copies of the values of the corresponding variable in the original, and copying a *stack*<*dictionary*$<K, I> * >$ produces a stack with the same length and the same pointer values. The following code fragment shows that a copy of a value of a structured type is distinct from the original.

```
array<int> A(0,2);
array<int> B = A;
int* p = A[0];
int* q = B[0];
p == q;          // evaluates to false
```

We next turn to item-based structured types.

LEDA Rule 9 *A value of an item-based structured type is a structured collection of items each of which has zero or more attributes. A copy of such a value is a collection of new items, one for each item in the original. The combinatorial structure imposed on the new items is isomorphic to the structure of the original. Every attribute of a new item which* *does not encode combinatorial structure is set to a copy of the corresponding attribute of the corresponding item in the original.*

Again we give some examples. Copying a *list*<E> of length 5 means creating five new list items, arranging these items in the form of a list, and setting the contents of the i-th new item, $1 \leq i \leq 5$, to a copy of the contents of the i-th item in the original. To copy a graph (type *graph*) with n nodes and m edges means creating n new nodes and m new edges and creating the isomorphic graph structure on them. To copy a *GRAPH*<*E1*, *E2*>[9] means copying the underlying graph and associating with each new node or edge a copy of the variable associated with the corresponding original node or edge. According to LEDA Rule 8 this means creating a new variable and initializing it with a copy of the value of the old variable.

The programming language literature sometimes uses the notions of *shallow* and *deep copy*. We want to relate these notions to the LEDA concept of a copy. Consider a structure *node_container* consisting of a pointer to a node container and a pointer to some other type.

```
class node_container
{ node_container* succ;
  E*              content;
}
```

Such a structure may, for example, arise in the implementation of a singly linked list; one pointer is used for the successor node and the other pointer is used for the the content, i.e., the list has type *list*<$E*$> for some type E. A shallow copy of a node is a new node whose two fields are initialized by component-wise assignment. A deep copy of a node is a copy of the entire region of storage reachable from the node, i.e., both kinds of pointers are followed when making a deep copy. In other words, a shallow copy follows no pointer, a deep copy follows all pointers. Our notion of copying is more semantically oriented. Copying a *list*<$E*$> of n items means creating n new items (this involves following the successor pointers), establishing a list structure on them, and setting the content attribute of each item to a copy of the contents of the corresponding item in the original. Since the type $E*$ is primitive (recall that all pointer types are primitive) this is tantamount to setting the contents of any new item to the contents of the corresponding old item. In particular, no copying of values of type E takes place. In other words, when making a copy of a *list*<$E*$> we follow successor pointers as if making a deep copy, but we do not follow the $E*$ pointers as if making a shallow copy.

Parameter passing by value involves copying. Since most arguments to operations on complex data types have value parameters, this has to be taken into account when reading the specifications of operations on data types. Consider, for example, the operation *D.insert*(k, i) for dictionaries. It takes a key k and an information i, adds a new item $\langle k, i\rangle$ to D and returns the new item[10]. Actually, this is not quite true. The truth is that the new

[9] A *GRAPH*<*E1*, *E2*> is a graph where each node and edge has an associated variable of type *E1* and *E2*, respectively.

[10] We assume for simplicity, that D contains no item with key k.

item contains a copy of k and a copy of i. For primitive types a value and a copy of it are identical and hence the sentence specifying the semantics of *insert* can be taken literally. For non-primitive types copies and originals are distinct and hence the sentence specifying the semantics of *insert* is misleading. We should say "adds a new item $\langle$copy of k, copy of $i\rangle$ to D" instead of "adds a new item $\langle k, i\rangle$ to D". We have decided to suppress the words "copy of" for the sake of brevity[11]. The following example shows the effect of copying.

```
dictionary<string,dictionary<int,int> > M;
dictionary<int,int> D;
dic_item it = D.insert(1,1);

M.insert("Ulli",D);
M.lookup("Ulli").inf(it);   // illegal
D.change_inf(it,2);
M.lookup("Ulli").access(1); // returns 1
D.insert(2,2);
M.lookup("Ulli").lookup(2); // returns nil
```

The insertion of D into M stores a copy of D in M. The item *it* belongs to D but not to the copy of D. Thus querying its *inf*-attribute in the copy of D returned by *M.lookup*("*Ulli*") is illegal. The operation *D.change_inf*(*it*, 2) changes the *inf*-attribute of *it* to 2; this has no effect on the copy of D stored in M and hence the access operation in the next line returns 1. Similarly, the second insertion into D has no effect on the copy and hence the lookup in the last line returns *nil*.

When the lifetime of an object ends it is *destructed*. The lifetime of a named object ends either at the end of the block where it was defined (this is the default rule) or when the program terminates (if declared static). The life of an anonymous object is ended by a call of *delete*. We need to say what it means to destruct an object. For LEDA-objects there is a simple rule.

LEDA Rule 10 *When a LEDA-object is destructed the space allocated for the object is freed. This is exactly the space that would be copied when a copy of the object were made.*

2.4 More on Argument Passing and Function Value Return

C++ knows two kinds of parameter passing, by value and by reference. Similarly, a function may return its result by value or by reference. We have already discussed value arguments and value results. We now review reference arguments and reference results and at the end of the section discuss functions as arguments. This section contains no material that is

[11] In the early versions of LEDA only primitive types were allowed as type parameters and hence there was no need for the words "copy of". When we allowed non-primitive types as type parameters we decided to leave the specification of *insert* and many other operations unchanged and to only make one global remark.

specific for LEDA; it is just a short review of reference parameters, reference results, and function arguments in C++.

The specification of a formal parameter has one of the three forms:

```
T x          (value parameter of type T),
T& x         (reference parameter of type T),
const T& x   (constant reference parameter of type T).
```

The qualifier `const` in the last form specifies that it is illegal to modify the value of the parameter in the body of the procedure. The compiler attempts to verify that this is indeed the case. Let A be the actual parameter corresponding to formal parameter x. Parameter passing is tantamount to the definition `T x = A` in the case of a value parameter and to the definition `T& x = A` in the case of a reference parameter. We already know the semantics of `T x = A`: a new variable x of type T is created and initialized with a copy of the value of expression A. The definition `T& x = A` does not define a new variable. Rather it introduces x as an additional name for the object denoted by A. Note that the argument A must denote an object in the case of a reference parameter. In either case the lifetime of x ends when the function call terminates.

Argument passing by reference must be used for parameters whose value is to be changed by the function. For arguments that are not to be changed by the function one may use either a value parameter[12] or a constant reference parameter. Note, however, that passing by value makes a copy of the argument and that copying a "large" value, e.g., a graph, list, or array, is expensive. Moreover, we usually want the function to work on the original of a value and not on a copy. We therefore advice to specify arguments of non-primitive types either as reference parameters or as constant reference parameters and to use value parameters only for primitive types. In our own code we very rarely pass objects of non-primitive type by value. If we do then we usually add the comment: "Yes, we actually want to work on a copy".

An example for the use of a constant reference parameter is

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

This function[13] takes a graph G, a node s of G, a non-negative cost function on the edges of G, and computes the distance of each vertex from the source (in *dist*). Also for each vertex $v \neq s$, *pred*$[v]$ is the last edge on a shortest path from s to v. The constant qualifiers ensure that *DIJKSTRA* does not change G and *cost* (although they are reference parameters). What would happen if we changed G to a value parameter? Well, we would pass a copy of G instead of G itself. Since a copy of a graph has new nodes and edges, s is not a node of the copy and *cost* is not defined for the edges of a copy. The function would fail if G was passed by value. Thus, it is essential that G is passed by reference.

Parameter passing moves information into a function and function value return moves

[12] It is legal to assign to a variable that is defined as a value parameter. Such an assignment does not affect the value of the actual parameter.

[13] See Section 6.6 for a detailed discussion of this function.

information out of a function. Consider the call of a function f with return type T or $T\&$ for some type T and assume that the call terminates with the return statement `return A`. The call is equivalent to the definition of a temporary t which is initialized with A, i.e., `return A` amounts to either `T t = A` or `T& t = A`. The temporary replaces the function call.

Let us go through an example. Let T be any type. We define four functions with the four combinations of return value and parameter specification.

```
T f1(T x)   {  return x; }
T f2(T& x)  {  return x; }
T& f3(T& x) {  return x; }
T& f4(T x)  {  return x; }
     // illegal, since a reference to a local variable is returned
```

Let y and z be objects of type T. The statement

```
z = f1(y);
```

copies y three times, first from y to the formal parameter x (value argument), then from x to a temporary t (value return), and finally from t to z (assignment). In

```
z = f2(y);
```

y is copied only twice, first from y to a temporary (value return) and then from the temporary into z (assignment).

```
z = f3(y);
```

copies y once, namely from y into z (assignment). Since *f3* returns a reference to an object of type T it can also be used on the left-hand side of an assignment. So

```
f3(y) = z;
```

assigns z to y.

Some operations take *functions as arguments*. A function argument f with result type T and argument types *T1*, ..., *Tk* is specified as

```
T(*f)(T1,T2,...,Tk)
```

The $*$ reflects the fact that a pointer to the function is passed. As a concrete example let us look at the bucket sort operation on lists with element type E:

```
void L.bucket_sort(int i,int j,int(*f)(E&));
```

requires a function f with a reference parameter of type E that maps each element of L into $[i\,..\,j]$. It sorts the items of L into increasing order according to f, i.e., item $\langle x\rangle$ is before $\langle y\rangle$ after the call if either $f(x) < f(y)$ or $f(x) = f(y)$ and $\langle x\rangle$ precedes $\langle y\rangle$ before the call.

2.5 Iteration

For many data types, LEDA offers *iteration macros* that allow to iterate over the elements of a collection. These macros are similar to the C++ *for*-statement. We give some examples. For all item-based types we have

```
forall_items(it,D)
  { /* the items in D are successively assigned to it */ }
```

This iteration successively assigns all items in *D* to *it* and executes the loop body for each one of them. For lists and sets we also have iteration statements that iterate over elements.

```
// L is a list<point>
point p;
forall(p,L)
  { /* the elements of L are successively assigned to p */ }.
```

For graphs we have statements to iterate over all nodes, all edges, all edges adjacent to a given node, . . . , for example:

```
forall_nodes(v,G)
  { /* the nodes of G are successively assigned to v*/ }
forall_edges(e,G)
  { /* the edges of G are successively assigned to e*/ }
forall_adj_edges(e,v)
  { /* all edges adjacent to v are successively assigned to e */ }
```

It is dangerous to modify a collection while iterating over it. We have

LEDA Rule 11 *An iteration over the items in a collection C must not add new items to C. It may delete the item under the iterator, but no other item. The attributes of the items in C can be changed without restriction.*

We give some examples:

```
// L is a list<int>
// delete all occurrences of 5
forall(it,L)
  if ( L[it] == 5 ) L.del(it);
forall(it,L)
  if ( L[it] == 5 ) L.del(L.succ(it));  // illegal
// add 1 to the elements following a 5
forall(it,L)
  if ( L[it] == 5 ) L[L.succ(it)]++;
forall(it,L)
  L.append(1);  // infinite loop
// G is a graph;
//add a new node s and edges (s,v) for all nodes of G
node s = G.new_node();
node v;
forall_nodes(v,G) if (v != s) G.new_edge(s,v);
```

The iterations statements in LEDA are realized by macro expansion. This will be discussed in detail in Section 13.9. We give only one example here to motivate the rule above and the rules to follow. The *forall_items* loop for lists

```
forall_items(it,L) { <<body>> }
```

expands into a C++ for-statement. The expansion process introduces a new variable; a distinct variable is generated for every loop by the expansion process. *loop_it* of type *list_item* and initializes it with the first item of L. In each iteration of the loop, *loop_it* is assigned to *it*, *loop_it* is advanced, and the loop body is executed. The loop terminates when *it* has the value *nil*.

```
for (list_item loop_it = (L).first_item();
     it = loop_it, loop_it = (L).next_item(loop_it), it;  )
{ <<body>> }
```

The fact that we use macro expansion to reduce the forall-loop to a C++ for-loop has two consequences.

LEDA Rule 12 *Break and continue statements can be used in forall-loops.*

We give an example.

```
list_item it;
forall_items(it,L) if ( L[it] == 5 ) break;
if ( it ) // there is an occurrence of 5 in L
else      // there is no occurrence of 5 in L
```

There is second consequence which is less pleasing. Consider

```
edge e;
forall(e,G.all_edges()) { <<body>> }
```

where the function *G.all_edges*( ) returns a list of all edges of G. The expansion process will generate

```
for (list_item loop_it = (G.all_edges()).first_item();
     it=loop_it,loop_it=(G.all_edges()).next_item(loop_it),it;)
{ <<body>> }
```

and hence the function *G.all_edges*( ) is called in every iteration of the loop. This is certainly not what is intended.

LEDA Rule 13 *The data type argument in an iteration statement must not be a function call that produces an object of the data type but an object of the data type itself.*

The correct way to write the loop above is

```
list<edge> E = G.all_edges();
edge e;
forall(e,E) { <<body>> }
```

or even simpler

```
forall_edges(e,G) { <<body>> }
```

2.6 STL Style Iterators

STL (Standard Template Library [MS96]) is a library of basic data types and algorithms that is part of the C++ standard. STL has a concept called *iterators* that is related to, but different from LEDA's item concept. In STL the forall-items loop for a *list*<*int*> is written as

```
for (list<int>::iterator it = L.begin(); it != L.end(); it++)
{ <<body>> }
```

In the loop body the content of the iterator can be accessed by $*it$; in LEDA one writes $L[it]$ to access the content of *it*.

Many LEDA data structures offer also STL style iterators. This feature is still experimental and we refer the user to the manual for details.

2.7 Data Types and C++

LEDA's implementation base is C++. We show in this section how abstract data types can be realized by the *class mechanism* of C++. We do so by giving a complete implementation of the data type stack which we specified at the beginning of this chapter. We also give the reader a first impression of LEDA's structure and we introduce the reader to Lweb and noweb.

A C++ class consists of *data members* and *function members*. The data members define how the values of the class are represented and the function members define the operations available on the class. Classes may be parameterized. We now define a parameterized class *stack*<E> that realizes the LEDA data type with the same name.

⟨*stack.c*⟩≡

```
template <class E> // E is the type parameter of stack
class stack
{ private:
      ⟨data members⟩
  public:
      ⟨function members⟩
};
```

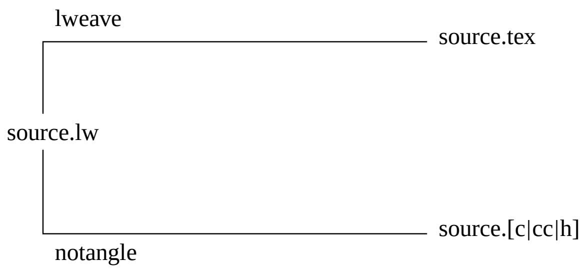

Figure 2.2 Lweb: lweave transforms a file source.lw into a file source.tex; notangle extracts program files. Lweb is a dialect of noweb [Ram94].

The definition of a class consists of a private part and a public part; the private part is only visible within the class and the public part is also visible outside the class. We declare the data members private to the class and hence invisible outside the class. This emphasizes the fact that we are defining an abstract data type and hence it is irrelevant outside the class how a value is represented in the machine and how the operations are implemented. To further emphasize this fact we give an implementation of stacks in this section that is different from the one actually used in LEDA. The function members are the interface of the class and hence public.

It is time to give more information about Lweb. *Lweb* is the literate programming tool which we use to produce manual pages, implementation reports, and which we used to produce this book. It is dialect of *noweb* [Ram94]. It allows us to write a program and its documentation into a single file (usually with extension .lw) and offers two utilities to produce two views of this file, one for a human reader and one for the C++ compiler: *lweave* typesets program and documentation and creates a file with extension .tex which can then be further processed using TEX and LATEXand *notangle* extracts the program and puts it into a file (usually with extension .c or .cc or .h). Figure 2.2 visualizes the process.

We postpone the discussion of lweave to Chapter 14 and only discuss notangle here. A noweb-file[14] consists of documentation chunks and code chunks. A documentation chunk starts with `@` followed by a blank or by a carriage return in column one of a line and a code chunk starts with ⟨*name of chunk*⟩`=` in column one of a line. Code chunks are given names. If several chunks are given the same name they are concatenated. Code chunks are referred to by ⟨*name of chunk*⟩.

In this section we have already defined a chunk ⟨*stack.c*⟩. It refers to chunks ⟨*data members*⟩ and ⟨*function members*⟩ which will be defined below. The command

```
notangle -Rstack.c Foundations.lw > stack.c
```

will extract the chunk `stack.c` (the “R” stands for root) from the file Foundations.lw (the name of the file containing this chapter) and write it into stack.c.

We come back to stacks. We represent a *stack*<*E*> by a C++ array A of type E and two integers sz and n with $n < sz$. The array A has size sz and the stack consists of elements

[14] As far as notangle is concerned there is no difference between a noweb-file (usually with extension .nw) and a Lweb-file.

$A[0]$, $A[1]$, ..., $A[n]$ with $A[n]$ being the top element of the stack. The stack is empty if $n = -1$.

⟨*data members*⟩≡

```
E* A;
int sz;
int n;
```

The function members correspond to the operations available on stacks. We start with the constructors. There are two ways to create a stack: *stack<E> S* creates an empty stack and *stack<E>* $S(X)$ creates a stack whose initial value is a copy of X. The corresponding function members are the so-called *default constructor* and so-called *copy constructor*, respectively. In C++ a constructor has the same name as the class itself, i.e., the constructors of class T have name T. The default constructor has no argument and the copy constructor has a constant reference argument of type T.

⟨*function members*⟩≡

```
stack()                    // default constructor
{ /* we start with an array of ten elements */
  A = new E[10];
  sz = 10;
  n = -1;
}
stack(const stack<E>& X) // copy constructor
{ sz = X.sz;
  A = new E[sz];
  n = X.n;
  for(int i = 0; i <= n; i++) A[i] = X.A[i];
}
```

We give some more functions: *empty* returns *true* if the stack is empty, *top* returns the top element of a non-empty stack, *push* adds an element to a stack, *pop* deletes an element from a non-empty stack and returns it, and $=$ performs assignment. We let *top* check its precondition and call an error-handler when it is violated. However, *pop* does not check its precondition. Recall that LEDA does not promise to check all preconditions.

⟨*function members*⟩+≡

```
int empty()  { return (n == -1); }
E top()
{ if ( n == -1) error_handler(1,"stack::top: stack is empty");
  return A[n];
}
E pop()  { return A[n--]; }
```

A *push* first checks whether there is still room in the array. If not, it doubles the size of A. In either case it increases n and assigns x to the new top element of the stack.

⟨*function members*⟩+≡
```
void push(const E& x)
{ if (n + 1 == sz)
  { sz = 2 * sz;
    E* B = A;
    A = new E[sz];
    for (int i = 0; i <= n; i++) A[i] = B[i];
    delete[] B;
  }
  A[++n] = x;
}
```

An assignment first checks for the trivial assignment $S = S$, then destroys the old value of the left-hand side, copies the right-hand side into the left-hand side, and finally returns a reference to the left-hand side.

⟨*function members*⟩+≡
```
stack<E>& operator=(const stack<E>& X)
{ if (this != &X)
  { delete[] A;
    sz = X.sz;
    A = new E[sz];
    n = X.n;
    for (int i=0; i<=n; i++) A[i] = X.A[i];
  }
  return (*this);
}
```

When the lifetime of a stack ends the array A needs to be deleted.

⟨*function members*⟩+≡
```
~stack()  {  delete[] A; }
```

This completes the definition of class *stack*<*E*>. The class essentially realizes the data type *stack*<*E*> as defined on page 4; we invite the reader to complete the implementation by writing the code for *clear*.

Our implementation of the stack data type wastes space. Imagine that we perform 1000 pushes followed by 1000 pops. The pushes will increase the size of A to at least 1000 but A does not shrink again during the pops. The LEDA implementation of stacks uses space in a more thrifty way; its space requirement is proportional to the number of elements in the stack.

In this section we gave the reader a first impression of how the data types of LEDA are implemented in C++. Chapter 13 gives the details.

2.8 Type Parameters

Most data types in LEDA are parameterized. We have lists over an arbitrary element type E and dictionaries over any linearly ordered key type K and any information type I. Any class that provides a certain small set of functions can be used as an actual type argument: one must be able to create a variable of the type and initialize it either with the default value (default constructor) or with a copy of an already existing value (copy constructor). One must be able to perform assignment (operator =), to read a value of the type from an input stream (function *Read*), and to print a value onto an output stream (function *Print*). Finally, when the lifetime of an object ends one must be able to destruct it (destructor). Sometimes, type arguments need to have additional abilities. Linearly ordered types have to support comparisons between their elements, hashed types have to support hashing, and numerical types have to support arithmetic.

LEDA Rule 14 *Any actual type argument must provide the following six functions:*

a default constructor	`T::T()`
a copy constructor	`T::T(const T&)`
an assignment operator	`T& T::operator=(const T&)`
a read function	`void Read(T&,istream&)`
a print function	`void Print(const T&,ostream&)`
a destructor	`T::~T().`

A linearly ordered type must in addition provide

a compare function	`int compare(const T&,const T&).`

A hashed type must in addition provide

a hash function	`int Hash(const T&)`
an equality operator	`bool operator ==(const T&,const T&).`

A numerical type must in addition have the basic arithmetic functions addition, subtraction, and multiplication, and the standard comparison operators.

We have already discussed the default constructor, the copy constructor, the destructor, and the assignment operator. The functions *Read* and *Print* read an object of type T from an input stream and print it to an output stream, respectively. Equality and the functions *compare, Hash* are discussed in the next section and number types are discussed in Chapter 4. We next give the complete definition of a linearly ordered class *pair*.

⟨*definition of class pair*⟩≡

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

```
int compare(const pair& p,const pair& q)
{ if (p.x < q.x) return - 1;
  if (p.x > q.x) return  1;
  if (p.y < q.y) return - 1;
  if (p.y > q.y) return  1;
  return 0;
}
```

We need to make two remarks about the definition of the class *pair*. (1) The functions *Read*, *Print*, and *compare* are not member functions of the class, but global functions. They are declared as friends of *pair* so that they can access the private data of the class. (2) We did not define two of the required functions, namely the assignment operator and the destructor ∼*pair*. The reason is that C++ will generate them automatically. More precisely, if no copy constructor, assignment operator, or destructor is defined then the default version is used. The default version copies component-wise, assigns component-wise, and destructs component-wise, respectively. Thus the definition of the copy constructor could also be omitted from class *pair*.

The type *pair* can be used as the key type in a dictionary, i.e., we may define

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

What happens if one uses a class T as an actual type parameter without defining one of the required functions (that are not generated automatically)? The C++ compiler will produce an error message that it cannot match certain functions. For example, the compiler used by the first author produces

```
LEDA/dictionary.h:52: no match for
‘_IO_ostream_withassign & << const pair & ’
```

when given the following program

⟨*parameterized_data_type_test.c*⟩≡

```
#include <LEDA/dictionary.h>
class pair
{ double x;
  double y;
public:
  pair() {  x = y = 0; }
  pair(const pair& p)  {  x = p.x; y = p.y; }
};
main(){
  dictionary<pair,int> D;
}
```

2.9 Memory Management

LEDA provides an efficient *memory management system* that is used for all node, edge, and item types and that can easily be customized for user-defined classes by means of the `LEDA_MEMORY` macro. One simply has to add the macro call `LEDA_MEMORY(T)` to the definition of class T. This call creates *new* and *delete* operators for the class T that rely on LEDA's memory manager. The main advantages over the built-in *new* and *delete* operators are:

- Memory is allocated in big chunks and thus frequent and costly calls to the memory allocator are avoided.
- Memory returned by the *delete* operator is reused by later calls of the *new* operator, i.e., the manager provides garbage collection.

The implementation of LEDA's memory manager is discussed in Section 13.8. The definition of our class *pair* now reads as follows. We advise the reader to follow this scheme in the definition of his classes.

⟨*refined definition of class pair*⟩≡
```
class pair
{ private:
    double x, y;

  public:
    pair() {  x = y = 0; }
    /* pair uses the default versions of copy constructor,
       assignment operator, and destructor */
    friend void Read(pair& p,istream& is) { is >> p.x >> p.y; }
    friend void Print(const pair& p,ostream& os)
      { os << p.x << " " << p.y; }
    friend int compare(const pair&,const pair&);

  LEDA_MEMORY(pair);
};
```

2.10 Linearly Ordered Types, Equality and Hashed Types

Algorithms frequently need to compare objects: a geometric algorithm may have to determine whether one line is above another line at a certain x-value, a sorting algorithm needs to compare the objects it is supposed to sort, and a shortest path algorithm needs to compare the lengths of two paths. Also, many data types such as dictionaries, priority queues, and sorted sequences need to compare the objects of their key type. The appropriate mathematical concept is a linear order.

A binary relation $\leq$ (less than or equal) on a set S is called a *linear order* if the following three conditions hold for all $x, y, z \in S$:

- $x \le x$ (reflexivity).
- $x \le y$ and $y \le z$ implies $x \le z$ (transitivity).
- $x \le y$ or $y \le x$ (anti-symmetry).

Note that the "or" in the third condition is not exclusive. We may have $x \le y$ and $y \le x$ even if x and y are distinct. Here is an example. For non-vertical lines g and h, define $g \le h$ if the intersection of g with the y-axis is below or equal to the intersection of h with the y-axis. Then $g \le h$ and $h \le g$ iff g and h intersect the y-axis in the same point.

We call x and y *equivalent* if $x \le y$ and $y \le x$ and we say that x is *strictly less than* y and write $x < y$ or $y > x$ if $x \le y$ and x and y are not equivalent. Note that for any two elements x and y exactly one of the following three relations holds: x is strictly less than y, x is equivalent to y, or y is strictly less than x.

In LEDA, a function *int cmp*(*const T*&, *const T*&) is said to realize a linear order on the type T if there is a linear order $\le$ on T such that for all x and y in T

$$cmp(x, y) \begin{cases} < 0, & \text{if } x < y \\ = 0, & \text{if } x \text{ is equivalent to } y \\ > 0, & \text{if } x > y \end{cases}$$

LEDA Rule 15 *A type T is called* linearly ordered *if the function*

```
int compare(const T&,const T&)
```

is defined for the type T and realizes a linear order on T. If compare(x, y) returns zero for two objects x and y then they are called compare-equivalent *or simply* equivalent.

Note that we have adopted the syntactic convention that the function with the name *compare* defines the order on T. This is in line with similar conventions already used in C++, e.g., that constructors have the same name as the type.

For many primitive data types a function *compare* is predefined and defines the so-called *default ordering* of the type. The default ordering is the usual "less than or equal" for the numerical types, the lexicographic ordering for strings, and the lexicographic ordering of the Cartesian coordinates for points. For all other types T there is no default ordering, and the user has to define the function *compare* if a linear order on T is required. We already gave an example in the preceding section.

A weaker concept than linear orders is equivalence relation. A binary relation R defines an *equivalence relation* on a set S if the following three conditions hold for all $x, y, z \in S$:

- xRx (reflexivity).
- xRy and yRz implies xRz (transitivity).
- xRy implies yRx (symmetry).

We have already seen an equivalence relation, namely compare-equivalence. The relation R defined by $x\,R\,y$ if $compare(x, y) == 0$ defines an equivalence relation. We also require

LEDA Rule 16 *If the equality operator*

```
bool operator==(const T&,const T&)
```

is defined for a class T then it defines an equivalence relation on T. We call x and y equal if $x == y$ evaluates to true.

We require *no* relationship between equality and compare-equivalence, i.e., two objects may be equal but not compare-equivalent or compare-equivalent but not equal. However, for all LEDA types with predefined *compare* and == the two notions agree. On the other hand, there are applications where it is natural to distinguish between the two concepts. For example, a plane sweep algorithm for line segment intersection (cf. Section 10.7.2) compares segments by the y-coordinate of their intersection with a vertical sweep line and thus two segments can be compare-equivalent without being equal.

We next turn to hashed types. A hashed type T must provide the equality operator and the function *int Hash*(*const T*&). Of course, the hash function should not tell objects apart that are equal.

LEDA Rule 17 *For any hashed type and any objects x and y of type T: if $x == y$ then $Hash(x) == Hash(y)$.*

There is one further point that we have to make. Recall that, for example, a dictionary stores copies of keys (and informations) and that for structured types a copy of a value is distinct from the original. It is possible to write compare functions and equality operators that distinguish between a value and a copy of the value. This would lead to a disaster, e.g., a lookup in a dictionary would fail to find a stored key. We therefore have

LEDA Rule 18 *A value and a copy of a value must be compare-equivalent and equal.*

For primitive types, this axiom is trivially fulfilled since a copy is identical to the original.

In some situations it is useful to have more than one linear order for a type T. For example, we might want to have two dictionaries *D1* and *D2* with key type *pair*. In *D1* the pairs are to be ordered by the lexicographic ordering of their Cartesian coordinates and in *D2* by the lexicographic ordering of their polar coordinates. The dictionary *D1* is easy to define. We simply write

```
dictionary<pair,int> D1,
```

but how can we define the second dictionary? After all, we have the syntactic convention that the function with the name *compare* defines the order on a type. There are two solutions, one old and one added recently.

The first solution is to define an equivalent type with the alternative ordering. The code sequence

```
int pol_cmp(const point& x,const point& y)
{ /* compute lexicographic ordering by polar coordinates */ }
DEFINE_LINEAR_ORDER(point,pol_cmp,pol_point);
dictionary<pol_point,int> D2;
```

first defines the ordering by polar coordinates and then defines a type *pol_point* by a call to the DEFINE_LINEAR_ORDER macro. The type *pol_point* is equivalent to the type *point*, in particular, a pol_point can be assigned to a point and vice versa. However, the ordering on the type *pol_point* is given by the function *pol_cmp*. The last line defines the desired dictionary *D2*.

The second solution makes the linear order an additional argument of any data type that requires a linearly ordered type, e.g.,

```
dictionary<point,int> D(pol_cmp);
```

declares a dictionary D that uses the function *pol_cmp* for comparing points.

Instead of passing a function to the dictionary, one can also pass a class which has a function operator and is derived from the class *leda_cmp_base*. This variant is helpful when the compare function depends on a global parameter. We give an example. More examples can be found in Sections 10.7.2 and 10.3. Assume that we want to compare edges of a graph *GRAPH<point, int>* (in this type every node has an associated point in the plane; the point associated with a node v is accessed as $G[v]$) according to the distance of their endpoints. We write

⟨*compare_example*⟩≡

```
class cmp_edges_by_length: public leda_cmp_base<edge> {
  const GRAPH<point,int>& G;
public:
  cmp_edges_by_length(const GRAPH<point,int>& g): G(g){}
  int operator()(const edge& e, const edge& f) const
  { point pe = G[G.source(e)]; point qe = G[G.target(e)];
    point pf = G[G.source(f)]; point qf = G[G.target(f)];
    return compare(pe.sqr_dist(qe),pf.sqr_dist(qf));
  }
};
main(){
  GRAPH<point,int> G;
  cmp_edges_by_length cmp(G);
  list<edge> E = G.all_edges();
  E.sort(cmp);
}
```

The class *cmp_edges_by_length* has a function operator that takes two edges e and f of a graph G and compares them according to their length. The graph G is a parameter of the constructor. In the main program we define *cmp*(G) as an instance of *cmp_edges_by_length* and then pass *cmp* as the compare object to the sort function of *list<edge>*. In the implementation of the sort function a comparison between two edges is made by writing $cmp(e, f)$, i.e., for the body of the sort function there is no difference whether a function or a compare object is passed to it.

The example above illustrates a nice feature of literate programming. We gave a named program chunk that illustrates a concept of LEDA. Of course, we want to make sure that the program fragment is correct and hence we want to execute it. To this effect we enclose it into a larger program chunk which we can extract and compile. We usually do not show the enclosing program chunk, i.e., we enclose it into a LaTeX command `\ignore` that makes it invisible to LaTeXby expanding to the empty string. We show the construction once:

```
\ignore{
<<compare_test.c>>=

#include <LEDA/graph.h>
#include <LEDA/point.h>

<<compare_example>>

@ }%end ignore
```

2.11 Implementation Parameters

Some data types in LEDA, e.g., dictionary, priority queue, d_array, and sorted sequence, come with several implementations. A user of such a data type can choose a particular implementation by giving the name of the implementation as an additional parameter, e.g., *_d_array<I, E, skiplist>* selects the skiplist implementation of dictionary arrays. Note that the type name now starts with an underscore. This is necessary since C++ does not allow us to overload templates. The following program uses the skiplist implementation of dictionary arrays to count word occurrences in the input stream.

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

The types with and without implementation parameter are closely related.

Any type *_T<T1, ..., Tk, xyz_impl>* is derived (in the C++ sense of the word) from the corresponding "normal" parameterized type *T<T1, ..., Tk>*. This allows us, for example, to pass an instance of type *_T<T1, ..., Tk, xyz_impl>* as an argument to a function with a formal parameter of type *T<T1, ..., Tk>&*, a feature that allows us to execute even pre-compiled algorithms with different implementations of data types. We give an example. We define a procedure *word_count* that has a parameter of type *d_array<string, int>*.

```
void word_count(d_array<string,int>& N)
{ string s;
  while (cin >> s) N[s]++;
  forall_defined(s,N) cout << s << " " << N[s] << endl;
}
```

Any implementation of d_arrays can be passed to *word_count*.

```
d_array<string,int> N1(0);
word_count(N1);
_d_array<string,int,skiplist> N2(0);
word_count(N2);
```

The section "Implementation Parameters" of the LEDA manual surveys the implementation parameters currently available. Section 13.6 discusses the realization of implementation parameters. The latter section also describes how a LEDA user may add his own implementation of a data type to the system.

2.12 **Helpful Small Functions**

There are a number of small, but helpful, functions. We mention some of them here and refer the reader to the section "Miscellaneous Functions" of the LEDA manual for the full list.

```
int i = read_int("i = ");
```

prints "`i = `" (more generally, its string argument) on standard output and then reads an integer from standard input. Similar functions exist to read strings, character, and doubles.

The function *used_time* is very helpful for running time experiments. For example, the chunk

```
float T = used_time();  // sets T to the current cpu time
// an experiment
cout << used_time(T);
  // sets T to the current cpu time and returns the difference
  // to the previous value of T
// another  experiment
cout << used_time(T);
```

will print the cpu time used in each of two experiments.

The function

```
void  print_statistics();
```

prints a summary of the currently used memory. For example, the program

⟨*memory_statistic*⟩≡

```
list<point> L;
{ for (int i = 0; i < 100000; i++) L.append(point());
  list<point> L1 = L;
}
print_statistics();
```

produces

```
STD_MEMORY_MGR (memory status)
    +--------------------------------------------------+
    |   size     used     free     blocks     bytes    |
    +--------------------------------------------------+
    |    12    100000   100214        294    2402568    |
    |    20        27      381          1       8160    |
    |    40    100002       77        493    4003160    |
    +--------------------------------------------------+
    |   time:  0.53 sec             space: 6300.92 kb |
    +--------------------------------------------------+
```

The statistics tell us that space for a total of 100000 + 100214 records of size 12 bytes (= list nodes), for a total of 27 + 381 records of size 20, and for a total of 100002 + 77 records of size 40 (= points) was allocated. It also gives information on which of these records are currently used and which are free. In our example, the records for the nodes of L and the points in L are still allocated and the records for the nodes of *L1* have already been freed. Observe that the program allocates space for 200000 list nodes, but only for 100000 structures to contain representations of points; read Section 2.2.2 to understand why. Space is allocated in blocks of 8160 bytes. The next to last column shows the number of allocated blocks for the structures of the different sizes and the last column shows the space consumption in bytes. Our program required about 6.3 megabytes. It ran for 0.53 seconds.

The functions

```
T    leda_min(const T& a, const T& b);
T    leda_max(const T& a, const T& b);
void leda_swap(T& a, T& b);
```

return the minimum, the maximum, and swap the values of their arguments, respectively. They can be used for any type T.

Finally, the function

```
double truncate(double x, int k = 10);
```

returns a double whose mantissa is truncated after $k - 1$ bits after the binary point, i.e., if $x \neq 0$ then the binary representation of the mantissa of the result has the form d.dddddddd, where the number of d's is equal to k.

2.13 **Error Handling**

The error handler

```
error_handler(int i, char* s);
```

writes s to the diagnostic output (cerr) and terminates the program abnormally if $i \neq 0$. The function

```
leda_assert(bool b, int i, char* s);
```

calls *error_handler*(i, s) if b is *false* and has no effect otherwise. Users can provide their own error handling function *handler* by calling

```
set_error_handler(handler);
```

After this function call *handler* is used instead of the default error handler. *handler* must be a function of type *void handler*(*int*, *char*∗). The parameters are replaced by the error number and the error message, respectively.

2.14 **Program Checking**

Programming is an error-prone task. How do we make sure that the programs in LEDA are correct? We take the following measures:

- We start from correct algorithms as described in the large literature on data structures and algorithms.
- We try to document our programs carefully. This book contains many examples of carefully documented programs. We try to document so carefully that we can show our programs around and give them to colleagues to read. Don Knuth coined the name "literate programming" for this style of programming.
- We test extensively and our large user community tests.
- We use program checking [SM90, BK89, BLR90, MNS$^+$96].

In this section we concentrate on the last item. Consider a program P that computes a function f. We call P *checkable* if for any input x it returns y, the alleged value of $f(x)$, and maybe additional information I that makes it easy to verify that indeed $y = f(x)$. By easy to verify we mean two things. Firstly, there must be a simple program C (a checking program) that, given x, y, and I, checks whether indeed $y = f(x)$. The program C should be so simple that its correctness is “obvious”. Secondly, the running time of C on inputs x, y, and I should be no larger than the running time of P on x. This guarantees that the checking program C can be used without severe penalty in running time.

We give some examples.

Consider a program that takes an $m \times n$ matrix A and an m vector b and is supposed to check whether the linear system $A \cdot x = b$ has a solution. As stated, the program is supposed to return a boolean value indicating whether the system is solvable or not. This program is not checkable. In order to make it checkable, we extend the interface.

On input A and b the program returns either:

- “the system is solvable” and a vector x such that $A \cdot x = b$ or
- “the system is unsolvable” and a vector c such that $c^T \cdot A = 0$ and $c^T \cdot b \neq 0$.

The extended program is easy to check. If it answers “the system is solvable”, we check that $A \cdot x = b$ and if it answers “the system is unsolvable”, we check that $c^T \cdot A = 0$ and $c^T \cdot b \neq 0$. Thus the check amounts to a matrix-vector and a vector-vector product which are fast and also easy to program. We leave it as an exercise to prove that the vector c exists, when the system is solvable, and only remark that Gaussian elimination will produce it.

The second example is planarity testing. The task is to decide whether a graph is planar. A witness of planarity is a planar embedding and a witness of non-planarity is a Kuratowski subgraph. The details can be found in Section 8.7. The planarity test played an important role in the development of LEDA. A first implementation of it was added to LEDA in 1991. The implementation had been tested on a small number of graphs. In 1993 we were sent a graph together with a planar drawing of it. However, our program declared the graph non-planar. It took us some days to discover the bug. More importantly, we realized that a complex question of the form ”is this graph planar” deserves more than a yes-no answer. We adopted the thesis that

a program should justify (prove) its answers in a way
that is easily checked by the user of the program.

By now many functions in LEDA justify their answers and come with checkers, see Sections 5.5.3, 10.3, 10.4.3, 10.5.3, and all sections in Chapter 7.

What do we gain by program checking?

First, the answer of a program can be verified for any single problem instance. This is much less than program verification which gives a guarantee for all problem instances, but it is assuring.

Second, a user of a program can develop trust in the program with little intellectual investment. A user of a linear systems solver does not need to understand the intricacies of Gaussian elimination. For any program run, she can convince herself of the correctness of the computation by a simple matrix-vector and vector-vector product. The program for the latter two tasks is so simple, that it is even conceivable to verify them formally. See [BSM97] for a first example of a verified checker.

Third, a developer of a program can give compelling evidence of its correctness without revealing any details of the implementation. It suffices to publish the interface of the functions, to define what constitutes a witness, and to publish the checking program.

Fourth, program checking allows us to use a potentially incorrect program as if it were correct. If a program operates correctly on a particular instance, fine, and if it operates incorrectly, it is caught by the checker. Thus, if all subroutines of a function f are checked, no checker of a subroutine fires, and an error occurs during the execution of f, the error must be in f. This feature of program checking is extremely useful during the debugging phase of program development.

Fifth, program checking supports testing. Traditionally testing is restricted to problem instances for which the solution is known by other means. Program checking allows one to test on *any* instance. For example, we use the following program (among others) to check our algorithm to compute maximal matching in graphs (see Section 7.7).

```
for (int n = 0; n < 100; n++)
  for (int m = 0; m < 100; m++)
  { random_graph(G,n,m); // random graph with n nodes and m edges
    list<edge> M = MAX_CARD_MATCHING(G,OSC);
    CHECK_MAX_CARD_MATCHING(G,M,OSC);
  }
```

Sixth, a checker can only be written if the problem at hand is rigorously defined. We noticed that some of our specifications contained hidden assumptions which were revealed during the design of the checker. For example, an early version of our biconnected components algorithm assumed that the graph contains no isolated nodes.

The papers [SM90, BS94, SM91, BSM97, BS95, BSM95, SWM95, BK89, BLR90, BW96, WB97, AL94, MNS$^+$96, DLPT97] contain further material on program checking.

2.15 Header Files, Implementation Files, and Libraries

The specifications of all LEDA types and algorithms are contained in the header files in directory LEDAROOT/incl/LEDA. In order to use a particular LEDA type or algorithms one must include the appropriate header file.

```
#include <LEDA/list.h>       // to use lists
#include <LEDA/dictionary.h> // to use dictionaries
#include <LEDA/point.h>      // to use points
#include <LEDA/graph_alg.h>  // to use the graph algorithms
#include <LEDA/geo_alg.h>    // to use the geometric algorithms
```

The implementations of all LEDA data types and algorithms are contained in the .c-files collected in the various subdirectories of LEDAROOT/src. They are pre-compiled into four libraries (libL.a, libG.a, libP.a, libWx.a) which can be linked with C++ application programs. The section “Using LEDA” of the LEDA manual describes how this is done.

2.16 Compilation Flags

The compilation flag -DLEDA_CHECKING_OFF turns off all checking of preconditions.

Bibliography

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

[AL94] N.M. Amato and M.C. Loui. Checking linked data structures. In *Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing (FTCS'94)*, pages 164–173, 1994.

[BK89] M. Blum and S. Kannan. Designing Programs That Check Their Work. In *Proceedings of the 21th Annual ACM Symposium on Theory of Computing (STOC'89)*, pages 86–97, 1989.

[BLR90] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing (STOC'90)*, pages 73–83, 1990.

[BS94] J.D. Bright and G.F. Sullivan. Checking mergeable priority queues. In *Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing (FTCS'94)*, pages 144–153, Los Alamitos, CA, USA, June 1994. IEEE Computer Society Press.

[BS95] J.D. Bright and G.F. Sullivan. On-line error monitoring for several data structures. In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing (FTCS'95)*, pages 392–401, Pasadena, California, 1995.

[BSM95] J.D. Bright, G.F. Sullivan, and G.M. Masson. Checking the integrity of trees. In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing (FTCS'95)*, pages 402–413, Pasadena, California, 1995.

[BSM97] J.D. Bright, G.F. Sullivan, and G.M. Masson. A formally verified sorting certifier. *IEEE Transactions on Computers*, 46(12):1304–1312, 1997.

[BW96] M. Blum and H. Wasserman. Reflections on the pentium division bug. *IEEE Transaction on Computing*, 45(4):385–393, 1996.

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

[DLPT97] O. Devillers, G. Liotta, F.P. Preparata, and R. Tamassia. Checking the convexity of polytopes and the planarity of subdivisions. Technical report, Center for Geometric Computing, Department of Computer Science, Brown Universi ty, 1997.

[FM97] U. Finkler and K. Mehlhorn. Runtime prediction of real programs on real machines. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 380–389, 1997.

[Meh84] K. Mehlhorn. *Data Structures and Algorithms 1,2, and 3*. Springer, 1984.

[MNS$^+$96] K. Mehlhorn, S. Näher, T. Schilz, S. Schirra, M. Seel, R. Seidel, and C. Uhrig. Checking Geometric Programs or Verification of Geometric Structures. In *Proceedings of the 12th Annual Symposium on Computational Geometry (SCG'96)*, pages 159–165, 1996.

[MS96] D.R. Musser and A. Saini. *STL tutorial and*

Reference Guide. Addison-Wesley, Reading, 1996.

[Rab80] M.O. Rabin. A probabilistic algorithm for testing primality. *Journal of Number Theory*, 12, 1980.

[Ram94] N. Ramsey. Literate programming simplified. *IEEE Software*, 11:97–105, 1994.

[SM90] G.F. Sullivan and G.M. Masson. Using certification trails to achieve software fault tolerance. In Brian Randell, editor, *Proceedings of the 20th Annual International Symposium on Fault-Tolerant Computing (FTCS '90)*, pages 423–433. IEEE, 1990.

[SM91] G.F. Sullivan and G.M. Masson. Certification trails for data structures. In *Proceedings of the 21st Annual International Symposium on Fault-Tolerant Computing(FTCS'91)*, pages 240–247, 1991.

[SS77] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal of Computing*, 6(1):84–85, 1977.

[SWM95] G.F. Sullivan, D.S. Wilson, and G.M. Masson. Certification of computational results. *IEEE Transactions on Computers*, 44(7):833–847, 1995.

[WB97] H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, 1997.

Index