A. An Overview of LEDA

> Libraries are great; use them whenever they do the job. Start with your system library, then search other libraries for appropriate functions.
>
> —Jon Bentley [31]

The detailed exposition of algorithms on trees and graphs made in this book is based on a full C++ implementation of all of the algorithms. These implementations are, in turn, based on the LEDA library of efficient C++ data structures and algorithms. A brief overview of the small subset of LEDA used throughout this book is provided in this appendix, in order to facilitate reading and is aimed at making the book self-contained. The reader is referred to [234, 236] for a more comprehensive description of LEDA.

A.1 Introduction

LEDA is a library of efficient C++ data structures and algorithms, developed over the last decade at the Max-Plank-Institute für Informatik in Saarbrücken, Germany, which is becoming a de facto standard for upper-undergraduate and graduate courses on graph algorithms throughout the world. As a matter of fact, LEDA allows the student, lecturer, researcher, and practitioner to complement algorithmic graph theory with actual implementation and experimentation, building upon a thorough library of efficient implementations of modern data structures and fundamental algorithms.

Another widely used library of C++ data structures and algorithms is STL, the ANSI C++ Standard Template Library [243]. STL includes efficient implementations of sequential container (*deque*, *list*, *stack*,

vector) and associative container (*map*, *set*) classes as well as generic algorithms, providing a large number of operations that can be applied to these classes and to the built-in *array* class as well. In much the same spirit, BGL, the Boost Graph Library [209, 302] that is being considered for adoption into the ANSI C++ Standard, includes efficient implementations of several basic graph algorithms and data structures, designed in the generic programming style of STL.

When it comes to combinatorial and geometric computing, though, LEDA goes much beyond STL and BGL by providing efficient implementations of graphs and their supporting data structures, together with a large number of graph algorithms (including shortest path, minimum spanning tree, bipartite matching, maximum flow, minimum cut, and graph drawing algorithms), while still providing efficient implementations of sequential container (*list*, *queque*, *stack*) and associative container (*dictionary*, *map*, *set*) classes. Furthermore, LEDA provides the most efficient data structures known for implementing each container class, even allowing the user to choose among different implementations for the same class. For instance, the *dictionary* class includes a default implementation by $(2,4)$-trees [26], together with a choice of several alternative implementations including AVL trees [1], $BB[\alpha]$ trees [48, 248], red-black trees [143], randomized search trees [292], and skip lists [262]. Last, but not least, LEDA includes an interface for graphical input and output in several platforms, which facilitates the editing and visualization of graphs and the interactive demonstration and animation of graph algorithms.

A.2 Data Structures

LEDA consists of a set of *container* classes, a set of algorithms to operate over these container classes, and a set of *iterators* to access the contents of a container.

Containers are just objects that hold other objects. A *sequential container* maintains a first element, a second element, and so on through a last element. Examples of sequential containers are the *list* class, which provides a linear list; the *stack* class, which provides a stack; and the *queue* class, which provides a single-ended queue.

An *associative container*, on the other hand, supports fast lookup of an object held in the container. Examples of associative containers are the *dictionary*, *d_array*, *h_array*, *map*, and *map2* classes, which provide access to the information associated with a unique key; and the *set* class, which provides access to unique elements.

Some of the containers provided by LEDA do not fit, though, in the previous distinction between sequential and associative containers. These include the *p_queue* class, which provides a single-ended priority queue; the *two_tuple*, *three_tuple*, and *four_tuple* classes, which provide *n*-tuples of elements; the *array* and *array2* classes, which provide one-dimensional dynamic arrays and two-dimensional static arrays, respectively; the *sortseq* class, providing sorted sequences, which support most of the operations of the *p_queue* and *dictionary* classes together with further search operations and some operations of the *list* class; and the *graph* class, which provides graphs together with efficient implementations of most fundamental graph algorithms.

Further containers provided by LEDA include the *integer* and *rational* classes, which provide an exact realization of integer and rational numbers, respectively, where the C++ types *int* and *long int* are just an approximation of integer numbers; the *bigfloat* and *real* classes, which provide a better approximation of real numbers than the C++ types *float*, *double*, and *long double*; and the *vector*, *integer_vector*, *matrix*, and *integer_matrix* classes, which provide one-dimensional and two-dimensional arrays of numbers together with basic linear algebra operations.

Algorithms operate on containers, and include capabilities for initializing, sorting, searching, and transforming the content of containers. Most LEDA data structures are implemented as C++ generic classes, which are parametrized by the type of the container, and encapsulate initializing, sorting, searching, and transforming algorithms, which are also implemented as C++ generic functions, parametrized again by the container element type. Most LEDA graph algorithms are implemented as C++ generic functions and wrapped in a separate *graph_alg* class, though, as further discussed below.

The operations shown in Fig. A.1 are common to the sequential container classes *list*, *stack*, and *queue*, and the associative container classes *dictionary* and *set*, as well as the container classes *p_queue*

and *sort_seq*. Furthermore, the associative container classes *d_array* and *h_array* provide *size* and *clear* but no *empty* operation, the *map* and *map2* classes only provide the *clear* operation, the *array* class only provides the *size* operation, and the *graph* class provides all three operations, where the *number_of_nodes* operation gives the order and the *number_of_edges* operation gives the size of the graph.

size()	returns the number of elements in the container
empty()	returns true if the container is empty and false otherwise
clear()	makes the container empty

Fig. A.1. Operations common to most LEDA container classes.

A.2.1 Linear Lists

The *list* class provides a linear list, that is, a sequence of elements. The *list*() constructor creates an empty list. Some of the operations provided by the *list* class are shown in Fig. A.2.

The *list* class includes a default implementation by doubly linked linear lists. Operations *front*, *back*, *push*, *append*, *pop*, *Pop*, *conc*, *size*, *empty* take $O(1)$ time; *rank*, *reverse*, *remove*, *permute*, *merge*, *unique*, *clear* take $O(n)$ time; *bucket_sort* takes $O(n+j-i)$ time; and *sort*, *merge_sort* take $O(n \log n)$ time, where n is the size or length of the list, and i and j are the minimal and maximal values of a given function on the elements of the list. The representation uses $O(n)$ space.

The efficient implementation of some of these operations requires the use of *items*, which are, roughly, addresses of containers. Most of the previous operations have a counterpart which acts upon list items. For instance, *first* and *last* are the counterpart of *front* and *back*, respectively, and return the first and the last item in the list, assuming the list is not empty. Furthermore, operation *contents*(*it*) returns the element contained at item *it* of the list.

Iteration over the items or the elements of a list is implemented by LEDA macros. In a macro call of the form *forall_items*(*it*,*L*), the items of list *L* are successively assigned to item *it*, and in a macro call of the form *forall*(*x*,*L*), the elements of *L* are successively assigned to variable *x*.

front()	returns the first element in the list, assuming the list is not empty
back()	returns the last element in the list, assuming the list is not empty
rank(*x*)	returns the rank of element *x* in the list, that is, the integer position of the first occurrence of *x* in the list, or zero is *x* does not occur in the list
push(*x*)	inserts element *x* at the front of the list
append(*x*)	inserts element *x* at the rear of the list
pop()	deletes and returns the first element in the list, assuming the list is not empty
Pop()	deletes and returns the last element in the list, assuming again that the list is not empty
remove(*x*)	removes all occurrences of element *x* from the list
conc(*L*)	deletes the elements of another list *L* and inserts them at the rear of the list
reverse()	reverses the sequence of elements of the list
permute()	performs a random permutation of the elements of the list
sort()	sorts the list by quick sort using the default linear ordering on the elements
merge_sort()	sorts the list by merge sort using the default linear ordering on the elements
bucket_sort(*f*)	sorts the list by bucket sort according to the integer function f defined on the elements
merge(*L*)	merges the list with *L* using the default linear ordering on the elements
unique()	removes duplicate elements from the list, assuming the list is already sorted according to the default linear ordering on the elements

Fig. A.2. Some of the operations provided by the *list* class.

A.2.2 Stacks

The *stack* class provides a stack, that is, a sequence of elements which are inserted and deleted at the same end (the top) of the sequence. The *stack*() constructor creates an empty stack. The operations provided by the *stack* class are shown in Fig. A.3.

top()	returns the top element in the stack, assuming the stack is not empty
pop()	deletes and returns the top element in the stack, assuming the stack is not empty
push(*x*)	inserts element *x* at the top of the stack

Fig. A.3. Operations provided by the *stack* class.

The *stack* class includes a default implementation by singly linked linear lists. Operations *top*, *pop*, *push*, *size*, *empty* take $O(1)$ time; and *clear* takes $O(n)$ time, where n is the size of the stack. The representation uses $O(n)$ space.

A.2.3 Queues

The *queue* class provides a single-ended queue, that is, a sequence of elements which are inserted at one end (the rear) and deleted at the other end (the front) of the sequence. The *queue*() constructor creates an empty queue. The operations provided by the *queue* class are shown in Fig. A.4.

top()	returns the front element in the queue, assuming the queue is not empty
pop()	deletes and returns the front element in the queue, assuming the queue is not empty
append(*x*)	inserts element *x* at the rear end of the queue

Fig. A.4. Operations provided by the *queue* class.

The *queue* class includes a default implementation by singly linked linear lists. Operations *top*, *pop*, *append*, *size*, *empty* take $O(1)$ time; and *clear* takes $O(n)$ time, where n is the size of the queue. The representation uses $O(n)$ space.

A.2.4 Priority Queues

The *p_queue* class provides a single-ended priority queue, that is, a queue of elements with both information and a priority associated with each element, where there is a linear order defined on the priorities. The *p_queue*() constructor creates an empty priority queue, based on the linear order defined by the global *compare* function. The operations provided by the *p_queue* class are shown in Fig. A.5.

The *p_queue* class includes a default implementation by Fibonacci heaps [119]. Operations *prio*, *inf*, *find_min*, *change_inf*, *size*, *empty* take $O(1)$ time; *insert*, *decrease_p* take amortized $O(1)$ time; *del_item*, *del_min* take amortized $O(\log n)$ time; and *clear* takes $O(n)$ time,

prio(*x*)	returns the priority of element *x*
inf(*x*)	returns the information associated with element *x*
insert(*i*,*p*)	inserts and returns an element with information *i* and priority *p* in the priority queue
find_min()	returns an element with the minimum priority, or *nil* if the priority queue is empty
del_item(*x*)	deletes element *x* from the priority queue
del_min()	deletes and returns the priority of an element with the minimum priority, assuming the priority queue is not empty
change_inf(*x*,*i*)	makes *i* the new information associated with element *x* in the priority queue
decrease_p(*x*,*p*)	makes *p* the new priority of element *x*, assuming *x* already belongs to the priority queue with a priority not smaller than *p*

Fig. A.5. Operations provided by the *p_queue* class.

where n is the size of the priority queue. The representation uses $O(n)$ space.

Alternative implementations include pairing heaps [118, 308], k-ary and binary heaps, lists, buckets, redistributive heaps [3], monotone heaps, and Emde-Boas trees [335]. The default implementation can also be selected by the implementation parameter *f_heap*, and the alternative implementations are selected by *p_heap*, *k_heap*, *bin_heap*, *list_pq*, *b_heap*, *r_heap*, *m_heap*, and *eb_tree*, respectively.

A.2.5 Tuples

The *two_tuple*, *three_tuple*, and *four_tuple* classes provide respectively 2-tuples, 3-tuples, and 4-tuples. The operations provided by the *tuple* classes are shown in Fig. A.6.

first()	returns the first component in the 2-tuple, 3-tuple, or 4-tuple
second()	returns the second component in the 2-tuple, 3-tuple, or 4-tuple
third()	returns the third component in the 3-tuple or 4-tuple
fourth()	returns the fourth component in the 4-tuple

Fig. A.6. Operations provided by the *two_tuple*, *three_tuple*, and *four_tuple* classes.

The comparison operator $\equiv$ is defined for 2-tuples, 3-tuples, and 4-tuples, together with a *compare* lexicographic comparison operator and a *Hash* hashing function. The *tuple* classes include an obvious

default implementation by wrapping the components as private data members of the class. Operations *first*, *second*, *third*, and *fourth* take $O(1)$ time. The representation uses $O(1)$ space times the total size of the components.

A.2.6 Arrays

The *array* class provides one-dimensional dynamic arrays. The *array*(*a*,*b*) constructor creates an array of $b-a+1$ elements indexed by integers in the range $[a..b]$; *array*(*n*) creates an array of *n* elements indexed by $[0..n-1]$; and *array*() creates an empty array of elements. Some of the operations provided by the *array* class are shown in Fig. A.7.

[*i*]	returns a reference to element $A(i)$ of the array, assuming that $a \leqslant i \leqslant b$
resize(*a*,*b*)	redefines the array to be indexed by integers in the range $[a..b]$
low()	returns the minimal index *a* of the array
high()	returns the maximal index *b* of the array
init(*x*)	sets all elements in the array to *x*
swap(*i*,*j*)	permutes elements $A(i)$ and $A(j)$ in the array
permute()	performs a random permutation of the elements of the array
sort()	sorts the array by quick sort using the default linear ordering on the elements
binary_search(*x*)	returns the index *i* such that $A(i) = x$, or $a-1$ if element *x* does not belong to the array, assuming the array is already sorted according to the default linear ordering on the elements

Fig. A.7. Some of the operations provided by the *array* class.

The *array* class includes a default implementation by C++ vectors. Operations [ ], *low*, *high*, *swap* take $O(1)$ time; *binary_search* takes $O(\log n)$ time; *init*, *permute* take $O(n)$ time; and *sort* takes $O(n \log n)$ time, where $n = b-a+1$. Further, operation *resize* takes time linear in the maximum between the old and the new size of the array. The representation uses $O(n)$ space times the size of the elements.

Further, the *array2* class provides two-dimensional static arrays. The *array2*(*a*,*b*,*c*,*d*) constructor creates a two-dimensional array of $b-a+1$ times $d-c+1$ elements indexed by integers in the range $[a..b]$ and $[c..d]$; and *array2*(*n*,*m*) creates a two-dimensional array of

$n \times m$ elements indexed by $[0..n-1]$ and $[0..m-1]$. The operations provided by the *array2* class are shown in Fig. A.8.

[*i*,*j*]	returns a reference to element $A(i,j)$ of the array, assuming that $a \leqslant i \leqslant b$ and $c \leqslant j \leqslant d$
low1()	returns the minimal index a of the array
high1()	returns the maximal index b of the array
low2()	returns the minimal index c of the array
high2()	returns the maximal index d of the array

Fig. A.8. Operations provided by the *array2* class.

The *array2* class includes a default implementation by C++ vectors. Operations [ ], *low1*, *high1*, *low2*, *high2* take $O(1)$ time. The representation uses $O(nm)$ space times the size of the elements, where $n = b - a + 1$ and $m = d - c + 1$.

A.2.7 Dictionaries

The *dictionary* class provides an associative container, consisting of a set of elements with both information and a unique key associated with each element, where there is a linear order defined on the keys and the information associated with an element is retrieved on the basis of its key. The *dictionary*() constructor creates an empty dictionary, based on the linear order defined by the global *compare* function. The operations provided by the *dictionary* class are shown in Fig. A.9.

The *dictionary* class includes a default implementation by $(2,4)$-trees [26]. Operations *key*, *inf*, *change_inf*, *empty*, *size* take $O(1)$ time; *insert*, *lookup*, *access*, *del*, *del_item* take $O(\log n)$ time; and *clear* takes $O(n)$ time, where n is the size of the dictionary. The representation uses $O(n)$ space.

Alternative implementations include AVL trees [1], $BB[\alpha]$ trees [48, 248], red-black trees [143], randomized search trees [292], and skip lists [262]. The default implementation can also be selected by the implementation parameter *ab_tree*, and the alternative implementations are selected by *avl_tree*, *bb_tree*, *rb_tree*, *rs_tree*, and *skiplist*, respectively.

key(*x*)	returns the key associated with element *x*
inf(*x*)	returns the information associated with element *x*
[*x*]	returns a reference to the information associated with element *x*
insert(*k*,*i*)	inserts and returns an element with key *k* and information *i* in the dictionary, replacing the element (if any) with key *k*
lookup(*k*)	returns the element with key *k* in the dictionary, or *nil* if there is no such element
access(*k*)	returns the information associated with key *k* in the dictionary, assuming there is such an element
del(*k*)	deletes the element with key *k* from the dictionary, if there is such an element
del_item(*x*)	deletes element *x* from the dictionary, assuming *x* belongs to the dictionary
change_inf(*x*,*i*)	makes *i* the information associated with element *x*, assuming *x* belongs to the dictionary

Fig. A.9. Operations provided by the *dictionary* class.

A.2.8 Dictionary Arrays

The *d_array* class provides a second kind of associative container, also consisting of a set of elements with both information and a unique key associated with each element, where there is a linear order defined on the keys. The *d_array*() constructor creates an empty dictionary array. The operations provided by the *d_array* class are shown in Fig. A.10.

[*k*]	returns a reference to the information associated with key *k* in the dictionary array
[*k*]=*i*	inserts an element with key *k* and information *i* in the dictionary array, replacing the element (if any) with key *k*, and returns a reference to the information associated with the element
defined(*k*)	returns true if there is an element with key *k* in the dictionary array and false otherwise
undefine(*k*)	deletes the element with key *k* from the dictionary array and leaves key *k* undefined

Fig. A.10. Operations provided by the *d_array* class.

The *d_array* class includes a default implementation by randomized search trees [292]. Operation *size* takes $O(1)$ time; [ ], *defined*, *undefine* take $O(\log n)$ time; and *clear* takes $O(n)$ time, where n is the size of the domain. The representation uses $O(n)$ space.

Alternative implementations include $(2,4)$-trees [26], AVL trees [1], $BB[\alpha]$ trees [48, 248], unbalanced binary trees, red-black trees

[143], and skip lists [262]. The default implementation can also be selected by the implementation parameter *rs_tree*, and the alternative implementations are selected by *ab_tree*, *avl_tree*, *bb_tree*, *bin_tree*, *rb_tree*, and *skiplist*, respectively.

Iteration over defined elements is implemented by LEDA macros. In a macro call of the form *forall_defined*(*k*,*A*), the keys associated with the elements in the dictionary array are successively assigned to variable *k*, and in a macro call of the form *forall*(*i*,*A*), the information associated with the elements of the dictionary array are successively assigned to variable *i*.

A.2.9 Hashing Arrays

The *h_array* class provides a third kind of associative container, consisting of a set of elements with both an information and a unique key associated with each element, where there is a hashing function defined on the keys. The *h_array*() constructor creates an empty hashing array. The operations provided by the *h_array* class are shown in Fig. A.11.

[*k*]	returns a reference to the information associated with key *k* in the hashing array
[*k*]=*i*	inserts an element with key *k* and information *i* in the hashing array, replacing the element (if any) with key *k*, and returns a reference to the information associated with the element
defined(*k*)	returns true if there is an element with key *k* in the hashing array and false otherwise
undefine(*k*)	deletes the element with key *k* from the hashing array and leaves key *k* undefined

Fig. A.11. Operations provided by the *h_array* class.

The *h_array* class includes a default implementation by hashing with chaining. Operation *size* takes $O(1)$ time; [ ], *defined*, *undefine* take expected $O(1)$ time; and *clear* takes $O(n)$ time, where n is the size of the hashing array. The representation uses $O(n)$ space.

There is an alternative implementation by dynamic perfect hashing [93, 117]. The default implementation can also be selected by the implementation parameter *ch_hash*, and the alternative implementation is selected by *dp_hash*.

Again, iteration over defined elements is implemented by LEDA macros. In a macro call of the form *forall_defined*(*k*,*A*), the keys associated with the elements in the hashing array are successively assigned to variable *k*, and in a macro call of the form *forall*(*i*,*A*), the information associated with the elements of the hashing array are successively assigned to variable *i*.

A.2.10 Maps

The *map* class provides still a fourth kind of associative container, consisting of a set of elements with both information and a unique key associated with each element, where keys are either of the type *int* or of a *pointer* or *item* type. The *map*() constructor creates an empty map. The operations provided by the *map* class are shown in Fig. A.12.

[*k*]	returns a reference to the information associated with key *k* in the map
[*k*]=*i*	inserts an element with key *k* and information *i* in the map, replacing the element (if any) with key *k*, and returns a reference to the information associated with the element
defined(*k*)	returns true if there is an element with key *k* in the map and false otherwise

Fig. A.12. Operations provided by the *map* class.

The *map* class includes a default implementation by hashing with chaining and table doubling. Operations [ ], *defined* take expected $O(1)$ time; and *clear* takes $O(n)$ time, where n is the number of elements in the map. The representation uses $O(n)$ space.

Iteration over defined elements is also implemented by LEDA macros. In a macro call of the form *forall_defined*(*k*,*A*), the keys associated with the elements in the map are successively assigned to variable *k*, and in a macro call of the form *forall*(*i*,*A*), the information associated with the elements of the map are successively assigned to variable *i*.

Further, the *map2* class provides an associative container consisting of a set of elements with both information and a unique ordered pair of keys associated with each element. The *map2*() constructor creates an empty two-dimensional map. The operations provided by the *map2* class are shown in Fig. A.13.

(*k*,*l*)	returns a reference to the information associated with key pair (k, l) in the two-dimensional map
(*k*,*l*)=*i*	inserts an element with key pair (k, l) and information *i* in the two-dimensional map, replacing the element (if any) with key pair (k, l), and returns a reference to the information associated with the element
defined(*k*,*l*)	returns true if there is an element with key pair (k, l) in the two-dimensional map and false otherwise

Fig. A.13. Operations provided by the *map2* class.

The *map2* class includes a default implementation by hashing with chaining and table doubling. Operations ( ), *defined* take expected $O(1)$ time; and *clear* takes $O(n)$ time, where *n* is the number of elements in the two-dimensional map. The representation uses $O(n)$ space.

A.2.11 Sets

The *set* class provides a set of elements. The *set*() constructor creates an empty set. The operations provided by the *set* class are shown in Fig. A.14.

insert(*x*)	inserts element *x* in the set
del(*x*)	deletes element *x* from the set
member(*x*)	returns true if *x* belongs to the set and false otherwise
choose()	returns some element of the set, assuming the set is not empty
join(*T*)	returns the union of the set with set *T*
diff(*T*)	returns the difference of the set minus set *T*
intersect(*T*)	returns the intersection of the set with set *T*
symdiff(*T*)	returns the symmetric difference of the set and set *T*

Fig. A.14. Operations provided by the *set* class.

The comparison operators $\leqslant$, $\geqslant$, $\equiv$, $\neq$, $<$, and $>$ are defined for the *set* class. The *set* class is implemented by randomized search trees [292]. Operations *empty*, *size*, *choose* take $O(1)$ time; *insert*, *del*, *member* take expected $O(\log n)$ time; *clear* takes $O(n)$ time; and *join*, *diff*, *intersect*, *symdiff* take expected $O(n \log n)$ time, where *n* is the size of the set.

Iteration over the elements of a set is also implemented by LEDA macros. In a macro call of the form *forall*(*x*,*S*), the elements of set *S* are successively assigned to variable *x*.

A.2.12 Partitions

The *partition* class provides a partition of a finite set of items into disjoint subsets, called *blocks*. The *partition*() constructor creates an empty partition of an empty set of elements. The operations provided by the *partition* class are shown in Fig. A.15.

make_block()	returns a new item and inserts a singleton block containing the item in the partition
find(*p*)	returns a canonical representative item of the block that contains item *p*
size(*p*)	returns the size of the block containing item *p*
number_of_blocks()	returns the number of blocks in the partition
same_block(*p*,*q*)	returns true if items *p* and *q* belong to the same block of the partition and false otherwise
union_blocks(*p*,*q*)	combines the blocks of the partition containing items *p* and *q*
split(*L*)	splits all blocks containing items in a list of items *L* of the partition into singleton blocks

Fig. A.15. Operations provided by the *partition* class.

The *partition* class includes a default implementation by the union-find data structure with weighted union and path compression [318, 321]. Operation *number_of_blocks* takes $O(1)$ time; a sequence of n operations *make_block* and a total of $m > n$ operations *find*, *size*, *same_block*, *union_blocks* takes $O(m\alpha(m,n))$ time, where the value of $\alpha(m,n)$, the inverse Ackermann function, is less than or equal to 4 for all practical purposes; and operation *split* takes time linear in the size of the blocks. The representation uses $O(n)$ space, where n is now the size of the set.

A.2.13 Sorted Sequences

The *sortseq* class provides a sorted sequence, that is, a sequence of elements with both information and a unique key associated with each

element, where there is a linear order defined on the keys. The *sortseq*() constructor creates an empty sorted sequence, based on the linear order defined by the global *compare* function. Some of the operations provided by the *sortseq* class are shown in Fig. A.16.

key(*it*)	returns the key associated with item *it*
inf(*it*)	returns the information associated with item *it*
insert(*k*,*i*)	inserts and returns an element with key *k* and information *i* in the sorted sequence, replacing the element (if any) with key *k*
lookup(*k*)	returns the element with key *k* in the sorted sequence, or *nil* if there is no such element
del(*k*)	deletes the element with key *k* from the sorted sequence, if there is such an element
del_item(*it*)	deletes item *it* from the sorted sequence, assuming *it* belongs to the sequence
change_inf(*it*,*i*)	makes *i* the information associated with item *it*, assuming *it* belongs to the sorted sequence

Fig. A.16. Some of the operations provided by the *sortseq* class.

Sorted sequences also offer so-called finger search operations, as well as operations for splitting and merging sorted sequences. The *sortseq* class includes a default implementation by skip lists [262]. Operations *empty*, *size*, *key*, *inf*, *del_item*, *change_inf* take $O(1)$ time; *insert*, *lookup*, *del* take expected $O(\log n)$ time; and *clear* takes $O(n)$ time, where n is the size of the sorted sequence. The representation uses $O(n)$ space times the size of the elements.

Alternative implementations of sorted sequences include $(2,4)$-trees [26], $BB[\alpha]$ trees [48, 248], red-black trees [143], and randomized search trees [292]. The default implementation can also be selected by the implementation parameter *skiplist*, and the alternative implementations are selected by *ab_tree*, *bb_tree*, *rb_tree*, and *rs_tree*, respectively.

A.2.14 Graphs

The *graph* class provides directed graphs, and also undirected graphs represented by bidirected graphs. The *graph*() constructor creates an empty graph. Some of the operations provided by the *graph* class are shown in Fig. A.17.

outdeg(*v*)	returns the number of arcs going out of vertex *v*
indeg(*v*)	returns the number of arcs coming into vertex *v*
source(*e*)	returns the source vertex of arc *e*
target(*e*)	returns the target vertex of arc *e*
opposite(*v*,*e*)	returns *target*(*e*) if vertex *v* is the source of arc *e* and *source*(*e*) otherwise
number_of_nodes()	returns the order of the graph
number_of_edges()	returns the size of the graph
choose_node()	returns a vertex of the graph at random, or *nil* if the graph is empty
choose_edge()	returns an arc of the graph at random, or *nil* if the graph is empty
all_nodes()	returns a *list* of all the vertices of the graph
all_edges()	returns a *list* of all the arcs of the graph
adj_edges(*v*)	returns a *list* of all the arcs going out of vertex *v*
in_edges(*v*)	returns a *list* of all the arcs coming into vertex *v*
adj_nodes(*v*)	returns a *list* of all the vertices adjacent to vertex *v*
new_node()	inserts and returns a vertex in the graph
new_edge(*v*,*w*)	inserts and returns an arc in the graph going out of vertex *v* and coming into vertex *w*
del_node(*v*)	deletes vertex *v* from the graph, together with all those arcs going out of or coming into vertex *v*
del_edge(*e*)	deletes arc *e* from the graph
del_all_nodes()	deletes all vertices from the graph
del_all_edges()	deletes all arcs from the graph
sort_nodes(*cmp*)	sorts the graph by quick sort according to the linear ordering *cmp* defined on the vertices
bucket_sort_nodes(*f*)	sorts the graph by bucket sort according to the integer function f defined on the vertices
sort_edges(*cmp*)	sorts the graph by quick sort according to the linear ordering *cmp* defined on the arcs
bucket_sort_edges(*f*)	sorts the graph by bucket sort according to the integer function f defined on the arcs
empty()	returns true if the graph is empty and false otherwise
clear()	makes the graph empty

Fig. A.17. Some of the operations provided by the *graph* class.

The *graph* class includes a default implementation by doubly linked lists of vertices and arcs. Operations *outdeg*, *indeg*, *source*, *target*, *opposite*, *number_of_nodes*, *number_of_edges*, *choose_node*, *choose_edge*, *new_node*, *new_edge*, *del_node*, *del_edge*, *empty* take $O(1)$ time; *adj_edges*, *adj_nodes* take time linear in the outdegree of the vertex; *in_edges* takes time linear in the indegree of the vertex; *all_nodes*, *all_edges*, *del_all_nodes*, *del_all_edges*, *bucket_sort_nodes*, *bucket_sort_edges*, *clear* take $O(n+m)$ time; operation *sort_nodes* takes $O(n \log n)$ time; and *sort_edges* takes $O(m \log m)$ time, where

n is the order and m is the size of the graph. The representation uses $O(n+m)$ space.

There are further operations supporting iteration over the vertices and arcs of a graph. An alternative form of iteration is implemented by LEDA macros. In a macro call of the form *forall_nodes*(*v*,*G*), the vertices of graph *G* are successively assigned to variable *v*; in a macro call *forall_edges*(*e*,*G*), the arcs of graph *G* are successively assigned to variable *e*; in a macro call *forall_rev_nodes*(*v*,*G*), the vertices of graph *G* are successively assigned to variable *v* in reverse order; in a macro call *forall_rev_edges*(*e*,*G*), the arcs of graph *G* are successively assigned to variable *e* in reverse order; in a macro call *forall_out_edges*(*e*,*v*), the arcs going out of vertex *v* are successively assigned to variable *e*; in a macro call *forall_in_edges*(*e*,*w*), the arcs coming into vertex *w* are successively assigned to variable *e*; and in a macro call of the form *forall_adj_nodes*(*w*,*v*), the vertices adjacent to vertex *v* are successively assigned to variable *w*.

Information can be associated with the vertices and arcs of a graph by defining appropriate arrays, matrices, or maps of vertices and arcs. While the former are static data structures, valid only for the vertices and arcs contained in the graph at the moment of creation of the array or matrix, the latter are dynamic data structures, which are also valid for vertices and arcs inserted later in the graph.

The *node_array* class provides a static array of information associated with the vertices of a graph. The *node_array*() constructor creates an empty static array of information to be associated with the vertices of a graph, the *node_array*(*G*) constructor creates a static array of information indexed by the vertices of graph *G*, and the *node_array*(*G*,*x*) constructor creates a static array of information indexed by the vertices of graph *G* and initializes all entries to the value of variable *x*. Some of the operations provided by the *node_array* class are shown in Fig. A.18.

The *node_array* class includes a default implementation by C++ vectors and an internal numbering of the vertices of the graph. Operations *get_graph*, [ ] take $O(1)$ time; and *init* takes $O(n)$ time, where n is the order of the graph. The representation uses $O(n)$ space times the size of the information associated with the vertices.

get_graph()	returns a reference to the graph which the array of vertices is associated with
[*v*]	returns a reference to the information associated with vertex *v*
init(*G*)	makes the array valid for all vertices of graph *G*
init(*G*,*x*)	makes the array valid for all vertices of graph *G* and initializes all entries to the value of variable *x*

Fig. A.18. Some of the operations provided by the *node_array* class.

The *edge_array* class provides a static array of information associated with the arcs of a graph. The *edge_array*() constructor creates an empty static array of information to be associated with the arcs of a graph, the *edge_array*(*G*) constructor creates a static array of information indexed by the arcs of graph *G*, and the *edge_array*(*G*,*x*) constructor creates a static array of information indexed by the arcs of graph *G* and initializes all entries to the value of variable *x*. Some of the operations provided by the *edge_array* class are shown in Fig. A.19.

get_graph()	returns a reference to the graph which the array of arcs is associated with
[*e*]	returns a reference to the information associated with arc *e*
init(*G*)	makes the array valid for all arcs of graph *G*
init(*G*,*x*)	makes the array valid for all arcs of graph *G* and initializes all entries to the value of variable *x*

Fig. A.19. Some of the operations provided by the *edge_array* class.

The *edge_array* class includes a default implementation by C++ vectors and an internal numbering of the arcs of the graph. Operations *get_graph*, [ ] take $O(1)$ time; and *init* takes $O(m)$ time, where m is the size the graph. The representation uses $O(m)$ space times the size of the information associated with the arcs.

The *node_matrix* class provides a static two-dimensional array of information associated with the vertices of a graph. The *node_matrix*() constructor creates an empty static two-dimensional array of information to be associated with the vertices of a graph, the *node_matrix*(*G*) constructor creates a static two-dimensional array of information indexed by the vertices of graph *G*, and the *node_matrix*(*G*,*x*) constructor creates a static two-dimensional array of information indexed by the vertices of graph *G* and initializes all entries to the value of vari-

able *x*. Some of the operations provided by the *node_matrix* class are shown in Fig. A.20.

get_graph()	returns a reference to the graph which the two-dimensional array of vertices is associated with
[*v*]	returns a reference to the *node_array* associated with vertex *v*
(*v*,*w*)	returns a reference to the information associated with vertices *v* and *w*
init(*G*)	makes the two-dimensional array valid for all vertex pairs of graph *G*
init(*G*,*x*)	makes the two-dimensional array valid for all vertex pairs of graph *G* and initializes all entries to the value of variable *x*

Fig. A.20. Some of the operations provided by the *node_matrix* class.

The *node_matrix* class includes a default implementation by *node_array* vectors and an internal numbering of the vertices of the graph. Operations *get_graph*, [ ], ( ) take $O(1)$ time; and *init* takes $O(n^2)$ time, where n is the order of the graph. The representation uses $O(n^2)$ space times the size of the information associated with the vertices.

The *node_map* and *edge_map* classes provides dynamic arrays of information associated with the vertices and the arcs of a graph, respectively. The *node_map*() and *edge_map*() constructors create empty dynamic arrays of information to be associated with the vertices and arcs of a graph, the *node_map*(*G*) and *edge_map*(*G*) constructors create dynamic arrays of information indexed by the vertices and arcs of graph *G*, and the *node_map*(*G*,*x*) and *edge_map*(*G*,*x*) constructors create dynamic arrays of information indexed by the vertices and arcs of graph *G* and initialize all entries to the value of variable *x*. Some of the operations provided by the *node_map* and *edge_map* classes are shown in Fig. A.21.

The *node_map* and *edge_map* classes includes a default implementation by hashing based on an internal numbering of the vertices and arcs of the graph. Operations *get_graph*, *init* take $O(1)$ time; and [ ] takes expected $O(1)$ time. The representation uses $O(n)$ and $O(m)$ space times the size of the information associated with the vertices and arcs, respectively, where n is the order and m is the size of the graph.

get_graph()	returns a reference to the graph which the dynamic array is associated with
[*v*]	returns a reference to the information associated with vertex *v*
[*e*]	returns a reference to the information associated with arc *e*
init(*G*)	makes the dynamic array valid for all vertices or arcs of graph *G*
init(*G*,*x*)	makes the dynamic array valid for all vertices or arcs of graph *G* and initializes all entries to the value of variable *x*

Fig. A.21. Some of the operations provided by the *node_map* and *edge_map* classes.

The *node_map2* class provides a dynamic two-dimensional array of information associated with the vertices of a graph. The *node_map2*() constructor creates an empty dynamic two-dimensional array of information to be associated with the vertices of a graph, constructor *node_map2*(*G*) creates a dynamic two-dimensional array of information indexed by the vertices of graph *G*, and the *node_map2*(*G*,*x*) constructor creates a dynamic two-dimensional array of information indexed by the vertices of graph *G* and initializes all entries to the value of variable *x*. Some of the operations provided by the *node_map2* class are shown in Fig. A.22.

get_graph()	returns a reference to the graph which the two-dimensional array of vertices is associated with
(*v*,*w*)	returns a reference to the information associated with vertices *v* and *w*
defined(*v*,*w*)	returns true if there is an entry in the two-dimensional array for vertices *v* and *w* and false otherwise
init(*G*)	makes the two-dimensional array valid for all vertex pairs of graph *G*
init(*G*,*x*)	makes the two-dimensional array valid for all vertex pairs of graph *G* and initializes all entries to the value of variable *x*

Fig. A.22. Some of the operations provided by the *node_map2* class.

The *node_map2* class includes a default implementation by hashing based on an internal numbering of the vertices of the graph. Operations *get_graph*, *init* take $O(1)$ time; and ( ), *defined* take expected $O(1)$ time. The representation uses $O(n+m)$ space times the size of the information associated with the vertices, where n is the order and m is the size of the graph.

An alternative form of associating information with vertices and arcs consists of using graphs parametrized by the type of the information associated with the vertices and arcs of the graph. The *GRAPH* class provides parametrized directed graphs, and also undirected graphs represented by bidirected graphs. The *GRAPH*(*V*,*E*) constructor creates an empty graph parametrized by vertex information type *V* and arc information type *E*. The *GRAPH* class is derived from the *graph* class, and inherits all of the operations provided by the *graph* class. Further operations provided by the *GRAPH* class are shown in Fig. A.23.

inf(*v*)	returns the information associated with vertex *v*
[*v*]	returns a reference to the information associated with vertex *v*
inf(*e*)	returns the information associated with arc *e*
[*e*]	returns a reference to the information associated with arc *e*
node_data()	makes the information associated with the vertices of the graph available as a *node_array*
edge_data()	makes the information associated with the arcs of the graph available as an *edge_array*
assign(*v*,*x*)	makes *x* the information associated with vertex *v*
assign(*e*,*x*)	makes *x* the information associated with arc *e*
new_node(*x*)	inserts and returns a vertex in the graph with information *x* associated
new_edge(*v*,*w*,*x*)	inserts and returns an arc in the graph going out of vertex *v* and coming into vertex *w* with information *x* associated

Fig. A.23. Some of the additional operations provided by the *GRAPH* class.

The *GRAPH* class includes a default implementation by doubly linked lists of vertices and arcs. Operations *inf*, [ ], *node_data*, *edge_data*, *assign*, *new_node*, *new_edge* take $O(1)$ time. The representation uses $O(n)$ space times the size of the information associated with the vertices plus $O(m)$ space times the size of the information associated with the arcs, where n is the order and m is the size of the graph.

LEDA also provides linear lists, priority queues, and partitions of the set of vertices of a graph, which have a more efficient implementation than the corresponding generic classes. The *node_list* class provides a linear list of the vertices of a graph that can contain each vertex of the graph at most once, with the additional restriction that each vertex of a graph is contained in at most one linear list of vertices. The

node_list() constructor creates an empty list of vertices. Some of the operations provided by the *node_list* class are shown in Fig. A.24.

The *node_list* class includes a default implementation by a doubly linked list of vertices, together with a map of vertices indexed by the vertices of the graph. Operations *append*, *push*, *insert*, *pop*, *del*, *head*, *tail*, *succ*, *pred*, *empty* take $O(1)$ time; *member* takes expected $O(1)$ time; and *clear* takes $O(n)$ time, where n is the size or length of the linear list of vertices.

Iteration over the vertices in a linear list of vertices is also implemented by LEDA macros. In a macro call of the form *forall*(*x*,*L*), the vertices of *L* are successively assigned to variable *x*.

append(*v*)	appends vertex *v* to the list of vertices
push(*v*)	inserts vertex *v* at the front of the list of vertices
insert(*v*,*w*)	inserts vertex *v* right after vertex *w*, assuming *w* belongs to the list
pop()	deletes and returns the first vertex from the list, assuming the list of vertices is not empty
del(*v*)	deletes vertex *v* from the list of vertices, assuming there is such a vertex
member(*v*)	returns true if vertex *v* belongs to the list of vertices and false otherwise
head()	returns the first vertex in the list of vertices, or *nil* if the list is empty
tail()	returns the last vertex in the list of vertices, or *nil* if the list is empty
succ(*v*)	returns the successor of vertex *v* in the list of vertices, assuming the list is not empty
pred(*v*)	returns the predecessor of vertex *v* in the list of vertices, assuming the list is not empty
empty()	returns true if the list of vertices is empty and false otherwise
clear()	makes the list of vertices empty

Fig. A.24. Some of the operations provided by the *node_list* class.

The *node_pq* class provides a priority queue of the vertices of a graph that can contain each vertex of the graph at most once, with the additional restriction that only one priority queue of vertices may be used for a graph. The *node_pq*(*G*) constructor creates a priority queue of the vertices of graph *G*. The operations provided by the *node_pq* class are shown in Fig. A.25.

The *node_pq* class includes a default implementation by priority queues—implemented, in turn, by binary heaps—and arrays of ver-

tices. Operations *prio*, *inf*, *member*, *find_min*, *size*, *empty* take $O(1)$ time; *insert*, *del*, *del_min*, *decrease_p* take $O(\log m)$ time; and *clear* takes $O(m)$ time, where m is the size of the priority queue. The representation uses $O(n)$ space, where n is the order of the graph.

The *node_partition* class provides a partition of the vertices of a graph. The *node_partition*(G) constructor creates a trivial partition of the vertices of graph G, containing a singleton block for each vertex of the graph. The operations provided by the *node_partition* class are shown in Fig. A.26.

prio(v)	returns the priority of vertex v
inf(v)	returns the information associated with vertex v
member(v)	returns true if vertex v belongs to the priority queue and false otherwise
insert(v,p)	inserts vertex v with priority p in the priority queue
find_min()	returns a vertex with the minimum priority, or *nil* if the priority queue is empty
del(v)	deletes vertex v from the priority queue
del_min()	deletes and returns a vertex with the minimum priority, assuming the priority queue is not empty
decrease_p(v,p)	makes p the new priority of vertex v, assuming v already belongs to the priority queue with a priority not smaller than p

Fig. A.25. Operations provided by the *node_pq* class.

find(v)	returns a canonical representative vertex of the block containing vertex v
size(v)	returns the size of the block containing vertex v
same_block(v,w)	returns true if vertices v and w belong to the same block of the partition and false otherwise
union_blocks(v,w)	combines the blocks of the partition containing vertices v and w
split(L)	splits all blocks containing vertices in a list of vertices L of the partition into singleton blocks
make_rep(v)	makes vertex v the canonical representative of the block containing it

Fig. A.26. Operations provided by the *node_partition* class.

The *node_partition* class includes a default implementation by a partition—implemented in turn by the union-find data structure with weighted union and path compression—together with a static array of items of the partition indexed by the vertices of the graph. Initialization takes $O(n)$ time; operations *find*, *size*, *same_block*, *union_blocks*,

make_rep take amortized $O(\alpha(n))$ time, where n is the order of the graph and the value of $\alpha(n)$, the inverse Ackermann function, is less than or equal to 4 for all practical purposes; and *split* takes time linear in the size of the blocks. The representation uses $O(n)$ space.

Further data structures provided by LEDA include compressed Boolean arrays; strings; random bits, characters, and numbers; data structures for graphical input and output; and computational geometry data structures. The reader is referred to [236] for details.

BFS(*G*,*s*,*dist*)
: computes the distance *dist* in graph *G* of all vertices reachable from vertex *s* during a breadth-first traversal of the graph in $O(n+m)$ time [232] [235, Sect. 7.3], and returns a list of all visited vertices

BICONNECTED_COMPONENTS(*G*,*compnum*)
: computes the biconnected components of an undirected graph *G* in $O(n+m)$ time [74] and returns the number of biconnected components, together with the component number *compnum* to which each edge belongs

COMPONENTS(*G*,*compnum*)
: computes the connected components of an undirected graph *G* in $O(n+m)$ time [232] and returns the number of connected components, together with the component number *compnum* to which each edge belongs

DFS_NUM(*G*,*dfsnum*,*compnum*)
: computes the number *dfsnum* and *compnum* in which the vertices are first and last visited, respectively, during a depth-first traversal of graph *G* in $O(n+m)$ time [235, Sect. 7.3] [317] and returns a list of tree edges in the corresponding depth-first forest of the graph

STRONG_COMPONENTS
: computes the strong components of graph *G* in $O(n+m)$ time [232] [235, Sect. 7.4.2] and returns the number of strong components, together with the strong component number *compnum* to which each vertex belongs

TOPSORT(*G*,*ord*)
: computes a topological sort *ord* of an acyclic graph *G* in $O(n+m)$ time [176] and returns true if the graph is indeed acyclic and false otherwise

TRANSITIVE_CLOSURE(*G*)
: computes the transitive closure of graph *G* in $O(n+m)$ time [134]

Fig. A.27. Some basic graph algorithms provided by LEDA.

A.3 Fundamental Graph Algorithms

LEDA provides efficient implementations of a large number of graph algorithms, including traversal, connectivity, shortest path, minimum spanning tree, bipartite matching, maximum flow, minimum cut, algorithms for planar graphs, and graph drawing algorithms. These graph

algorithms are implemented as C++ generic functions, and accept both graphs and parametrized graphs as arguments.

Basic graph algorithms provided by LEDA include topological sorting; depth-first and breadth-first traversal; connected, biconnected, and strongly connected components; and transitive closure algorithms. Some of them are summarized in Fig. A.27.

Further graph algorithms provided by LEDA include several single-source shortest path and all-pairs shortest paths algorithms, as well as minimum spanning tree algorithms. Some of them are summarized in Fig. A.28.

ACYCLIC_SHORTEST_PATH(*G*,*s*,*cost*,*dist*,*pred*)
: computes the distance *dist* and the shortest path arcs *pred* in an acyclic graph G with arc costs *cost* of all vertices reachable from vertex s in $O(n+m)$ time [235, Sect. 7.5.4]

ALL_PAIRS_SHORTEST_PATHS(*G*,*cost*,*dist*)
: computes the distance *dist* between all reachable pairs of vertices in graph G with no negative-cost cycles in $O(nm+n^2 \log n)$ time [235, Sect. 7.5.10] and returns true if the graph has indeed no negative-cost cycles and false otherwise

BELLMAN_FORD(*G*,*s*,*cost*,*dist*,*pred*)
: computes the distance *dist* and the shortest path arcs *pred* in graph G with arc costs *cost* of all vertices reachable from vertex s in $O(nm)$ time [29] [235, Sect. 7.5.7–7.5.9] and returns false if there is a negative-cost cycle reachable from s in the graph and true otherwise

DIJKSTRA(*G*,*s*,*cost*,*dist*,*pred*)
: computes the distance *dist* and the shortest path arcs *pred* in graph G with nonnegative arc costs *cost* of all vertices reachable from vertex s in $O(m+n \log n)$ time [94] [235, Sect. 6.6]

MIN_SPANNING_TREE(*G*,*cost*)
: computes a spanning tree of minimum cost of an undirected graph G with edge costs *cost* in $O(n+m)$ time [204] [235, Sect. 6.8] and returns a list of the edges in the minimum spanning tree

SPANNING_TREE(*G*)
: computes a spanning tree of an undirected graph G in $O(n+m)$ time [232] and returns a list of the edges in the spanning tree

Fig. A.28. Further graph algorithms for shortest paths and minimum spanning trees provided by LEDA.

Additional graph algorithms provided by LEDA include several matching algorithms in bipartite and general graphs, as well as combinatorial optimization algorithms for computing a maximum or minimum flow in a network, and minimum cut algorithms. Some of them are summarized in Fig. A.29.

MAX_CARD_BIPARTITE_MATCHING(*G*)
: computes a matching of maximum cardinality in a bipartite graph G in $O(\sqrt{n}m)$ time [7, 165] or in $O(nm)$ time [114], and returns the list of edges in the matching

MAX_CARD_MATCHING(*G*)
: computes a matching of maximum cardinality in an undirected graph G in $O(nm\alpha(n,m))$ time [103, 121], and returns the list of edges in the matching

MAX_FLOW(*G*,*s*,*t*,*cap*)
: computes a maximum (s,t)-flow in a network (G,s,t,cap) with arc capacities *cap* in $O(n^2\sqrt{m})$ time [128] [235, Sect. 7.10] and returns the value of the flow

MAX_WEIGHT_ASSIGNMENT(*G*,*cost*,*pot*)
: computes a perfect matching of maximum cost in an undirected graph G with edge costs *cost* in $O(n(m+n\log n))$ time [235, Sect. 7.8] and returns the list of edges in the matching and a correctness certificate *pot*

MAX_WEIGHT_BIPARTITE_MATCHING(*G*,*cost*,*pot*)
: computes a matching of maximum cost in a bipartite graph G with edge costs *cost* in $O(n(m+n\log n))$ time [235, Sect. 7.8] and returns the list of edges in the matching, together with a correctness certificate *pot*

MIN_CUT(*G*,*weight*)
: computes a cut of minimum weight in graph G with arc weights *weight* in $O(nm+n^2\log n)$ time [11, 310] and returns the value of the cut

Fig. A.29. Further graph algorithms for flows in networks and cuts and matchings in graphs provided by LEDA.

Beware that the maximum cardinality bipartite matching algorithms actually transform the representation of the graph by reordering adjacency lists, thereby modifying the combinatorial embedding of the graph. Beware also that some of the layout algorithms for planar graphs included in LEDA transform first the graph into a planar map, modifying again the combinatorial embedding of the graph. In particular, applying a straight-line layout or a visibility representation layout to an ordered tree transforms it into a *different* ordered tree.

LEDA also provides efficient implementations of algorithms for planar graphs and graph drawing algorithms. The reader is referred to [236, Chap. 7–8] for details.

A.4 A Simple Representation of Trees

The LEDA representation of graphs may be used for representing trees as well, although LEDA graphs offer many operations that do not make much sense for trees—for instance, most of those dealing with embedded graphs, because a tree has only one face—and more space-efficient representations for trees could be adopted, as already

discussed in Sect. 1.4. Nevertheless, the choice in this book has been to define a simple *tree* class, which inherits the representation and operations from the *graph* class and also provides additional operations for trees.

The *tree* class provides nonempty rooted trees. The *tree*() constructor creates an empty tree, and the *tree*(G) constructor creates a tree representation of graph G, assuming G is the graph representation of a nonempty rooted tree. All operations provided by the *graph* class can be applied to trees as well. Further operations provided by the *tree* class are shown in Fig. A.30.

parent(v)	returns the parent of node v, or *nil* if v is the root of the tree
is_root(v)	returns true if node v is the root of the tree and false otherwise
is_leaf(v)	returns true if node v is a leaf of the tree and false otherwise
root()	returns the root node of the tree
first_child(v)	returns the first child of node v in the tree, or *nil* if v is a leaf node
last_child(v)	returns the last child of node v in the tree, or *nil* if v is a leaf node
next_sibling(v)	returns the next sibling of node v in the tree, or *nil* if v is either the root node or a last child of the tree
previous_sibling(v)	returns the previous sibling of node v in the tree, or *nil* if v is either the root node or a first child of the tree
is_first_child(v)	returns true if node v is a first child in the tree and false otherwise
is_last_child(v)	returns true if node v is a last child in the tree and false otherwise
number_of_children(v)	returns the number of children of node v in the tree

Fig. A.30. Operations provided by the *tree* class.

The *tree* class includes a default implementation derived from the LEDA *graph* class. Initialization from a LEDA graph takes $O(n)$ time; operations *parent*, *is_root*, *is_leaf*, *first_child*, *last_child*, *next_sibling*, *previous_sibling*, *is_first_child*, *is_last_child*, *number_of_children* take $O(1)$ time; *root* takes time linear in the depth of the tree; and *clear* takes $O(n)$ time, where n is the size of the tree. The representation uses $O(n)$ space.

Recall from Sect. 1.1 that a connected undirected graph $G = (V,E)$ with n vertices and m edges is a nonempty undirected tree if and only if $n = m + 1$. As a matter of fact, in an empty undirected tree both $n = 0$ and $m = 0$.

In the case of directed, rooted trees, the previous condition is still necessary but no longer sufficient for a graph to be a tree, because a node could have more than one parent. Recall from Definition 1.36 that a connected graph $T = (V,E)$ is a tree if the underlying undirected graph has no cycles and for all nodes $v \in V$, there is a path in the graph from a distinguished node root$[T]$ to node v.

Lemma A.1. *A connected graph $T = (V,E)$ with n vertices and m arcs is a tree if and only if $n = m+1$ and* $\mathrm{indeg}(v) \leqslant 1$ *for all vertices $v \in V$.*

Proof. Let $T = (V,E)$ be a connected graph with n vertices and $m = n-1$ arcs such that $\mathrm{indeg}(v) \leqslant 1$ for all vertices $v \in V$. Suppose that $\mathrm{indeg}(v) = 1$ for all vertices $v \in V$. Then, $\sum_{v \in V} \mathrm{indeg}(v) = \sum_{v \in V} 1 = n$. But $\sum_{v \in V} \mathrm{indeg}(v) = m$ by Theorem 1.5, contradicting the assumption that $n = m+1$. Therefore, there is a vertex $u \in V$ with $\mathrm{indeg}(u) = 0$.

Now, since the graph is connected there is, for all vertices $v \in V$, an undirected path from vertex u to vertex v. Suppose that the path from vertex u to some vertex $v \in V$ is not a directed path. Then, there must be some vertex $w \in V$ along the path with $\mathrm{indeg}(w) \geqslant 2$, contradicting the assumption that $\mathrm{indeg}(v) \leqslant 1$ for all vertices $v \in V$. Therefore, there is a directed path from vertex u to vertex v, for all vertices $v \in V$.

Suppose now that there is a cycle $[v_1, \ldots, v_2, \ldots, v_1]$ in the underlying undirected graph. If the cycle is directed, vertex u cannot belong to the cycle, because $\mathrm{indeg}(u) = 0$, and there must be some vertex $w \in V$ in the cycle such that the directed path from vertex u up to, but not including, vertex w is disjoint from the cycle. Then, $\mathrm{indeg}(w) \geqslant 2$, contradicting the assumption that $\mathrm{indeg}(v) \leqslant 1$ for all vertices $v \in V$. Otherwise, if the cycle is not directed, there must be some vertex $w \in V$ along either the undirected path $[v_1, \ldots, v_2]$ or the undirected path $[v_2, \ldots, v_1]$ with $\mathrm{indeg}(w) \geqslant 2$, contradicting again the assumption that $\mathrm{indeg}(v) \leqslant 1$ for all vertices $v \in V$. Therefore, there are no cycles in the underlying undirected graph, and T is indeed a tree.

Conversely, let $T = (V,E)$ be a tree with n nodes and m arcs. By Theorem 1.43, $n = m+1$. Suppose now that there is a node $w \in V$ with $\mathrm{indeg}(w) > 1$. Let $v_1, v_2 \in V$ be nodes such that $(v_1,w), (v_2,w) \in E$, and let v be the least common ancestor of nodes v_1 and v_2 in T. Then, the paths $[v, \ldots, v_1, w]$ and $[v, \ldots, v_2, w]$ together constitute a cycle in

the underlying undirected graph, contradicting the assumption that T is a tree. Therefore, $\mathrm{indeg}(v) \leqslant 1$ for all nodes $v \in V$. □

The following procedure *is_tree* returns true if the graph is the representation of a rooted tree and false otherwise. Since the LEDA procedure *Is_Connected* takes $O(n)$ time to determine whether the graph is connected, procedure *is_tree* also takes $O(n)$ time.

429a ⟨tree graph representation 429a⟩≡

```
bool is_tree(
 const graph& G)
{
 node v;
 forall_nodes(v,G)
  if ( G.indeg(v) > 1 ) return false;    // nonunique parent
 return ( G.number_of_nodes() ≡ G.number_of_edges() + 1
  ∧ Is_Connected(G) );
}
```

The following *tree* class is derived from the LEDA *graph* class. The destructor, copy constructor, and copy assignment operator, as well as the operations on LEDA graphs, are all inherited from the *graph* class.

429b ⟨tree graph representation 429a⟩+≡

```
class tree : public graph {
 public:
  tree() : graph() { }

  tree( const graph& G ) : graph( G )
  {
   if ( ¬is_tree(G) )
    error_handler(1,"Graph is not a tree");
  }

  node parent( const node v ) const;
  bool is_root( const node v ) const;
  bool is_leaf( const node v ) const;
  node root() const;
  node first_child( const node v ) const;
  node last_child( const node v ) const;
  node next_sibling( const node v ) const;
  node previous_sibling( const node v ) const;
  bool is_first_child( const node v ) const;
  bool is_last_child( const node v ) const;
  int number_of_children( const node v ) const;
};
```

In the graph representation of a tree $T = (V,E)$, the parent of a node $v \in V$ is the unique node $u \in V$ such that $(u,v) \in E$.

430a ⟨tree graph representation 429a⟩ +≡
```
node tree::parent( const node v ) const
{
 if ( (*this).indeg(v) ≡ 0 ) return nil;
 return (*this).source((*this).first_in_edge(v));
}
```

The root of a tree $T = (V,E)$ is the only node $v \in V$ which is not the target of any arc of the form $(u,v) \in E$, that is, the only node $v \in V$ with $\mathrm{indeg}(v) = 0$.

430b ⟨tree graph representation 429a⟩ +≡
```
bool tree::is_root( const node v ) const
{
 return (*this).indeg(v) ≡ 0;
}
```

On the other hand, a node $v \in V$ is a leaf of a tree $T = (V,E)$ if there is no arc of the form $(v,w) \in E$, that is, if $\mathrm{outdeg}(v) = 0$.

430c ⟨tree graph representation 429a⟩ +≡
```
bool tree::is_leaf( const node v ) const
{
 return (*this).outdeg(v) ≡ 0;
}
```

The root of a tree can be found in time linear in the depth of the tree by starting off with some node—for instance, a node chosen at random—and then following the path up to the root of the tree.

Notice that operation *root* could be implemented to take $O(1)$ time instead, by just extending the graph representation of a tree with an explicit pointer to the root node. However, the root of a tree must still be found when reading the tree from a LEDA graph window; for instance, in the interactive demonstration of graph algorithms. Besides, all algorithms on trees discussed in this book take $\Omega(n)$ time anyway, the time needed to either generate or input the tree.

430d ⟨tree graph representation 429a⟩ +≡
```
node tree::root() const
{
 node v = (*this).choose_node();
 while ( ¬is_root(v) )
  v = parent(v);
```

```
  return v;
}
```

The first and the last child of a node in a tree are the target of the first and the last arc going out of the node, respectively, according to the relative order of the children of the node fixed by the graph representation of the tree.

431a ⟨tree graph representation 429a⟩+≡

```
node tree::first_child( const node v ) const
{
  if ( (*this).outdeg(v) ≡ 0 ) return nil;
  return (*this).target((*this).first_adj_edge(v));
}
```

431b ⟨tree graph representation 429a⟩+≡

```
node tree::last_child( const node v ) const
{
  if ( (*this).outdeg(v) ≡ 0 ) return nil;
  return (*this).target((*this).last_adj_edge(v));
}
```

In the same sense, the next and the previous sibling of a node in a tree are the target of the next and the previous arc going out of the parent of the node, respectively, according to the relative order of the arcs going out of the parent node fixed by the graph representation of the tree.

431c ⟨tree graph representation 429a⟩+≡

```
node tree::next_sibling( const node v ) const
{
  if ( (*this).indeg(v) ≡ 0 ) return nil;
  edge e = (*this).adj_succ((*this).first_in_edge(v));
  if ( e ≡ nil ) return nil;
  return (*this).target(e);
}
```

431d ⟨tree graph representation 429a⟩+≡

```
node tree::previous_sibling( const node v ) const
{
  if ( (*this).indeg(v) ≡ 0 ) return nil;
  edge e = (*this).adj_pred((*this).first_in_edge(v));
  if ( e ≡ nil ) return nil;
  return (*this).target(e);
}
```

Now, the first child of the parent of a node is said to be a *first child* node, and the last child of the parent of a node is said to be a *last child* node.

432a ⟨tree graph representation 429a⟩ +≡
```
bool tree::is_first_child( const node v ) const
{
 if ( (*this).is_root(v) ) return false;
 return (*this).first_child((*this).parent(v)) ≡ v;
}
```

432b ⟨tree graph representation 429a⟩ +≡
```
bool tree::is_last_child( const node v ) const
{
 if ( (*this).is_root(v) ) return true;
 return (*this).last_child((*this).parent(v)) ≡ v;
}
```

The number of children of a node in a tree is just the outdegree of the corresponding vertex in the graph representation of the tree.

432c ⟨tree graph representation 429a⟩ +≡
```
int tree::number_of_children( const node v ) const
{
 return (*this).outdeg(v);
}
```

Iteration over all children nodes $w \in V$ of node $v \in V$ in a tree $T = (V,E)$ is implemented by the LEDA macro for iterating over the target vertex w of all arcs of the form $(v,w) \in E$ in the graph representation of the tree. In a macro call of the form *forall_children*(*w*,*v*), the children of node *v* are successively assigned to node *w*.

432d ⟨tree graph representation 429a⟩ +≡
```
#define forall_children(w,v) forall_adj_nodes(w,v)
```

A *TREE* class is also defined, providing nonempty rooted trees parametrized by the type of the information associated with the nodes and edges of the tree. The *TREE* class is derived from the *GRAPH* class which is, in turn, derived from the *graph* class. The destructor, copy constructor, and copy assignment operator, as well as the operations on LEDA parametrized graphs, are all inherited from the *GRAPH* class. Further operations provided by the *TREE* class are identical to the additional operations provided by the *tree* class.

432e ⟨tree graph representation 429a⟩ +≡

```
template<class V,class E>
class TREE : public GRAPH<V,E> {
public:
  TREE<V,E>() { }

  TREE( const GRAPH<V,E>& G ) : GRAPH<V,E>( G )
  {
    if ( ¬is_tree(G) ) error_handler(1,"Graph is not a tree");
  }

  node parent( const node v ) const
  {
    if ( (*this).indeg(v) ≡ 0 ) return nil;
    return (*this).source((*this).first_in_edge(v));
  }

  bool is_root( const node v ) const
  {
    return (*this).indeg(v) ≡ 0;
  }

  bool is_leaf( const node v ) const
  {
    return (*this).outdeg(v) ≡ 0;
  }

  node root() const
  {
    node v = (*this).choose_node();
    while ( ¬is_root(v) )
      v = parent(v);
    return v;
  }

  node first_child( const node v ) const
  {
    if ( (*this).outdeg(v) ≡ 0 ) return nil;
    return (*this).target((*this).first_adj_edge(v));
  }

  node last_child( const node v ) const
  {
    if ( (*this).outdeg(v) ≡ 0 ) return nil;
    return (*this).target((*this).last_adj_edge(v));
  }

  node next_sibling( const node v ) const
  {
    if ( (*this).indeg(v) ≡ 0 ) return nil;
    edge e = (*this).adj_succ((*this).first_in_edge(v));
    if ( e ≡ nil ) return nil;
    return (*this).target(e);
  }
```

```
  node previous_sibling( const node v ) const
  {
   if ( (*this).indeg(v) == 0 ) return nil;
   edge e = (*this).adj_pred((*this).first_in_edge(v));
   if ( e == nil ) return nil;
   return (*this).target(e);
  }

  bool is_first_child( const node v ) const
  {
   if ( (*this).is_root(v) ) return false;
   return (*this).first_child((*this).parent(v)) == v;
  }

  bool is_last_child( const node v ) const
  {
   if ( (*this).is_root(v) ) return true;
   return (*this).last_child((*this).parent(v)) == v;
  }

  int number_of_children( const node v ) const
  {
   return (*this).outdeg(v);
  }

};
```

A.5 A Simple Implementation of Radix Sort

Some of the algorithms for tree isomorphism and related problems discussed in Chap. 4, required radix sorting a list of arrays of integers. Radix sort is based on bucket sort, and consists of sorting the list of arrays of integers in several stages, using bucket sort at each stage.

Bucket Sort

LEDA does not provide an implementation of radix sort, although it includes an efficient implementation of bucket sort which is, however, of a rather low level. The following, straightforward implementation of bucket sort is a more convenient starting point for a simple LEDA implementation of radix sort.

Let $\mathrm{ord} : E \to \mathrm{int}$ be a function with $\mathrm{ord}(x) \in [i..j]$ for all elements x of list L. Bucket sorting L consists of maintaining an array *bucket*

of $j - i + 1$ lists of elements, initially empty; appending each element x of L to bucket number $k = \mathrm{ord}(x) - i + 1$; and then collecting the elements in sorted order by concatenating all the buckets back into a single list.

The following algorithm sorts the elements of list L using bucket sort, implementing the previous procedure. The elements of L are distributed into buckets, and the buckets are concatenated back into the sorted list L. The identity function is provided as default value for the ord parameter. Space efficiency of the implementation follows from both *pop* and *conc* operations being destructive.

435 ⟨subroutines 40⟩+≡

```
template<class E>
void straight_bucket_sort(
  list<E>& L,
  int i,
  int j,
  int (*ord)(const E&) = id)
{
  int n = j - i + 1;     // number of buckets needed
  array<list<E> > bucket(1,n);
  while ( ¬L.empty() ) {
    E x = L.pop();
    int k = ord(x) - i + 1;     // element x belongs in bucket k
    if ( k ⩾ 1 ∧ k ⩽ n ) {
      bucket[k].append(x);
    } else {
      error_handler(1,"bucket sort: value out of range");
  } }
  for ( i = 1; i ⩽ n; i++ )
    L.conc(bucket[i]);     // destructive
  ⟨double-check bucket sort 436b⟩
}
```

Remark A.2. Notice that bucket sort is a *stable* sorting method, that is, if $\mathrm{ord}(x) = \mathrm{ord}(y)$ and x precedes y in L, then x also precedes y in the bucket sorted list L, for all elements x and y of L. As a matter of fact, in the previous bucket sorting algorithm, the elements of L are appended to appropriate buckets in order, and this order is preserved when concatenating the buckets.

Bucket sort is often called just on a list of elements. A procedure call of the form *straight_bucket_sort*(*L*,*ord*) is the same as the procedure call *straight_bucket_sort*(*L*,*i*,*j*,*ord*), where *i* and *j* are respectively

the minimum and maximum values of $\mathrm{ord}(x)$ as x ranges over the elements of list L. Again, the identity function is provided as default value for the ord parameter.

436a ⟨subroutines 40⟩+≡

```
template<class E>
void straight_bucket_sort(
  list<E>& L,
  int (*ord)(const E&) = id)
{
  if ( L.empty() ) return;
  int i = ord(L.head());
  int j = i;
  E x;
  forall(x,L) {
    int k = ord(x);
    if ( k < i ) i = k;
    if ( k > j ) j = k;
  }
  straight_bucket_sort(L,i,j,ord);
}
```

The following double-check of bucket sort, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that L is sorted, that is, that $\mathrm{ord}(x) \leqslant \mathrm{ord}(\mathrm{succ}(x))$ for all but the last element x of list L.

436b ⟨double-check bucket sort 436b⟩≡

```
{ list_item it;
  forall_items(it,L) {
    if ( it ≠ L.last() ∧ ord(L[it]) > ord(L[L.succ(it)]) ) {
      error_handler(1,"Wrong implementation of bucket sort");
} } }
```

Lemma A.3. *The algorithm for bucket sorting runs in $O(n+j-i)$ time using $O(j-i)$ additional space, where n is the length of list L and i, j are respectively the minimum and maximum values of* $\mathrm{ord}(x)$ *as x ranges over the elements of L.*

Proof. The first loop ranges over the elements of L, and thus takes $O(n)$ time, and the second loop ranges over the buckets, thus taking $O(j-i)$ time. Therefore, the algorithm runs in $O(n+j-i)$ time. Further, the algorithm uses $O(j-i)$ additional space, because an array of $j-i+1$ buckets is first allocated, the elements of L are popped from L

and appended to their respective bucket, and the buckets are concatenated back into L, where both *pop* and *conc* operations are destructive.

The double-check of bucket sorting runs in $O(n)$ time using $O(1)$ additional space. □

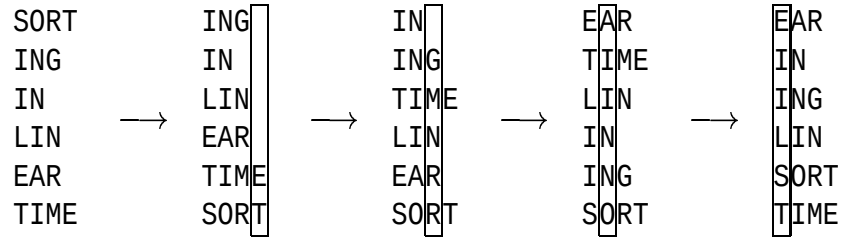
It follows that $j - i \leqslant n$ is a sufficient condition for bucket sort to run in $O(n)$ time.

Corollary A.4. *The algorithm for bucket sorting runs in $O(n)$ time upon a list L of n elements if $j - i \leqslant n$, where* $\mathrm{ord}(x) \in [i..j]$ *for all elements x of L.*

Radix Sort

Radix sort is a stable sorting method consisting of k passes of bucket sort, on the last element, the previous-to-last element, and so on, until bucket sorting on the first element of each list, where k is the maximum among the lengths of the lists of integers to be sorted. Since such a procedure requires direct access to individual elements in each list, though, the list of lists of integers to be sorted is represented by a list of arrays of integers instead.

Example A.5. Radix sorting the list of arrays of integers $[[83,79,82,84], [73,78,71], [73,78], [76,73,78], [69,65,82], [84,73,77,69]]$, which are the ASCII codes of the characters in the list of character strings [SORT,ING, IN, LIN, EAR, TIME], proceeds as follows.



The leftmost column shows the list of character strings corresponding to the list of arrays of integers to be sorted, and the remaining columns show the result of bucket sorting on increasingly significant character positions. The character position bucket sorted on to produce each list of character strings from the previous one is highlighted.

Remark A.6. The previous algorithm is known as LSD (least significant digit) radix sort, because the least significant element in the arrays of integers is bucket sorted first. MSD (most significant digit) radix sort, on the contrary, consists of bucket sorting on the most significant element first, and then radix sorting each group of array suffixes sharing a common first element. While with LSD radix sort all elements of the arrays are inspected, only the distinguishing prefixes of the arrays are inspected with MSD radix sort, although at the expense of decomposing the sorting problem into a large number of subproblems. With LSD radix sort, bucket sorting is applied several times to the same list of arrays of integers instead.

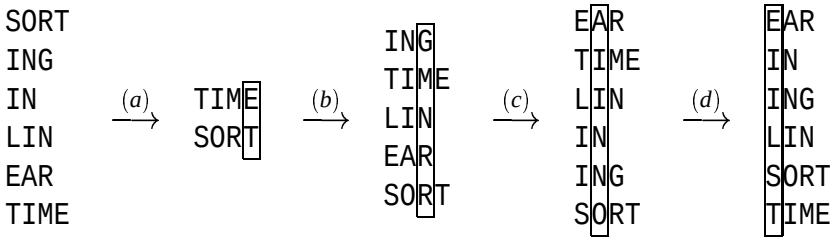
Now, given a list L of n arrays of at most k integers each, where $\mathrm{ord}(x) \in [i..j]$ for all integers x in each array in list L and $j - i \leqslant n$, the previous radix sorting algorithm still makes k passes of bucket sort, each taking $O(n)$ time, and therefore runs in $O(kn)$ time. This is efficient when all arrays have approximately the same length, because the total length of the arrays of integers to be sorted is $\sum_{i=1}^{n} O(k) = O(\sum_{i=1}^{n} k) = O(kn)$. However, radix sorting a list L of n arrays of integers, one of them of length n and the other ones of only a few integers, would take $O(n^2)$ time, although the total length of the arrays of integers to be sorted is, in this case, $O(n) + \sum_{i=2}^{n} O(1) = O(n)$.

The reason for the lack of efficiency is that in the previous radix sorting algorithm, a large number of empty buckets might be inspected at each pass of bucket sort. Nevertheless, the previous algorithm can be extended to a simple radix sorting algorithm running in time linear in the total length of the arrays of integers to be sorted, as follows.

Let maxlen be the maximum length among all arrays of integers to be sorted. Inspecting empty buckets can be avoided by building for each position i in the arrays, with $1 \leqslant i \leqslant \mathrm{maxlen}$, a list $LEN[i]$ of the arrays of length i and also a sorted list $FULL[i]$ of the integers appearing at the ith position in the arrays.

Then, bucket sorting the list L of arrays on the ith component proceeds by concatenating at the front of L the list $LEN[i]$ of arrays of length i, distributing the resulting list L into buckets according to the ith component and, finally, concatenating back into L all nonempty buckets, that is, those buckets corresponding to $FULL[i]$.

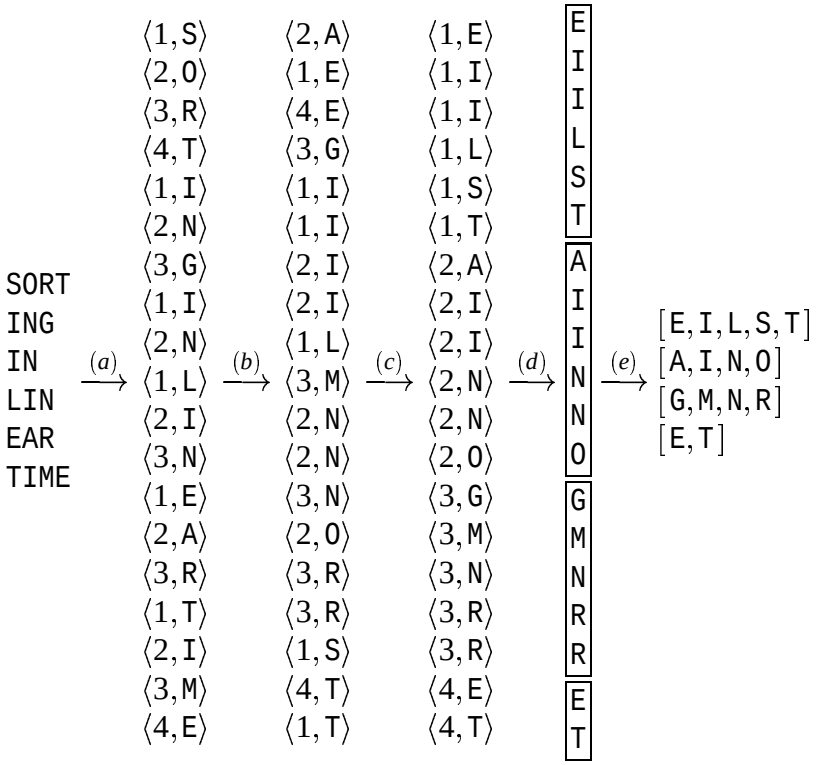
Example A.7. More efficient radix sorting of the list of arrays of integers of Example A.5, which are the ASCII codes of the characters in the list of character strings [SORT, ING, IN, LIN, EAR, TIME].



(a) List $\mathtt{LEN}[4]; L = [\,\mathtt{SORT}, \mathtt{TIME}\,]; [\,]$ is distributed into buckets according to the fourth component, but only those buckets corresponding to $\mathtt{FULL}[4] = [\,\mathtt{E}, \mathtt{T}\,]$ are concatenated back into L. (b) List $\mathtt{LEN}[3]; L = [\,\mathtt{ING}, \mathtt{LIN}, \mathtt{EAR}\,]; [\,\mathtt{TIME}, \mathtt{SORT}\,]$ is distributed into buckets according to the third component, but only the buckets corresponding to $\mathtt{FULL}[3] = [\,\mathtt{G}, \mathtt{M}, \mathtt{N}, \mathtt{R}\,]$ are concatenated back into L. (c) List $\mathtt{LEN}[2]; L = [\,\mathtt{IN}\,]; [\,\mathtt{ING}, \mathtt{TIME}, \mathtt{LIN}, \mathtt{EAR}, \mathtt{SORT}\,]$ is distributed into buckets according to the second component, but only those buckets corresponding to $\mathtt{FULL}[2] = [\,\mathtt{A}, \mathtt{I}, \mathtt{N}, \mathtt{O}\,]$ are concatenated back into L. (d) List $\mathtt{LEN}[1]; L = [\,]; [\,\mathtt{EAR}, \mathtt{TIME}, \mathtt{LIN}, \mathtt{IN}, \mathtt{ING}, \mathtt{SORT}\,]$ is distributed into buckets according to the first component, but only those buckets corresponding to $\mathtt{FULL}[1] = [\,\mathtt{E}, \mathtt{I}, \mathtt{L}, \mathtt{S}, \mathtt{T}\,]$ are concatenated back into L.

Nonempty buckets can be found, as a matter of fact, by bucket sorting. Let P be a list of pairs $\langle i, A[i] \rangle$ for all arrays A in list L, with $1 \leqslant i \leqslant \mathrm{maxlen}$. After bucket sorting P on the second position and then on the first position, a sorted list (without duplicates) $FULL[i]$ of the integers in the ith position of the arrays can be obtained, indicating which buckets will be nonempty during the radix sort, when bucket sorting on the ith position of the arrays.

Example A.8. Finding nonempty buckets for a more efficient radix sorting of the list of arrays of integers of Example A.7, which are the ASCII codes of the characters in the list of character strings [SORT, ING, IN, LIN, EAR, TIME].



(a) Extracting a list P of pairs $\langle i, A[i]\rangle$ for all arrays A in list L, with $1 \leqslant i \leqslant 4$. (b) Bucket sorting list P by the second position. (c) Bucket sorting the resulting list P by the first position. (d) Extracting a sorted list $FULL[i]$ of the integers in the ith position of the arrays. (e) Removing duplicates from each sorted list $FULL[i]$.

The following algorithm sorts the arrays of integers of list L using radix sort, implementing the previous procedure. In order to avoid using a new bucket array for each pass of bucket sort, a single array *bucket* of lists of arrays of integers is used throughout the whole procedure, and the list P of pairs $\langle i, A[i]\rangle$ is also represented by a list of arrays of integers.

```
440  ⟨subroutines 40⟩+≡
       void radix_sort(
         list<array<int> >& L,
         int min,
         int max)
       {
         if ( L.empty() ) return;
         ⟨compute number n of buckets needed 441a⟩
```

```
  array<list<array<int> > > bucket(1,n);
  ⟨find nonempty buckets 441b⟩
  ⟨distribute arrays by length 442c⟩
  for ( int i = maxlen; i ⩾ 1; i− ) {
    ⟨bucket sort list L of arrays on ith component 443a⟩
  }
  ⟨double-check radix sort 443c⟩
}
```

The number of buckets needed is the largest of $max - min + 1$ and *maxlen*, the maximum length among all arrays A in list L.

441a ⟨compute number n of buckets needed 441a⟩≡

```
int maxlen = 0;
array<int> A,T;
forall(A,L)
  maxlen = leda_max(maxlen,A.size());
int n = leda_max(max − min + 1,maxlen);
```

Nonempty buckets are found by (a) building a list P of pairs $\langle i, A[i] \rangle$ for all arrays A in list L, with $1 \leqslant i \leqslant$ maxlen; (b) bucket sorting list P on the second component and then on the first component; (c) collecting the second component of each pair of the sorted list P in a sorted list $FULL[i]$ of integers; and (d) making each lists $FULL[i]$ unique by removing duplicates.

Notice that duplicates can be removed in time linear in the length of the list using the LEDA procedure *unique*, because lists $FULL[i]$ are sorted.

441b ⟨find nonempty buckets 441b⟩≡

```
⟨make list P of pairs (i,A[i]) 441c⟩
⟨bucket sort list P of pairs by A[i] 442a⟩
⟨bucket sort list P of pairs by i 442b⟩
array<list<int> > FULL(1,maxlen);
forall(A,P)
  FULL[A[1]].append(A[2]);
for ( int i = 1; i ⩽ maxlen; i++ )
  FULL[i].unique();
```

Recall that the list of pairs $\langle i, A[i] \rangle$ is represented by a list P of arrays $T = [i, A[i]]$ of integers, in order to use the same array *bucket* of lists of arrays of integers for bucket sorting throughout the radix sorting algorithm.

441c ⟨make list P of pairs $(i,A[i])$ 441c⟩≡

```
list<array<int> > P;
```

```
forall(A,L) {
  for ( int i = 1; i ⩽ A.size(); i++ ) {
    array<int> T(1,i,A[i]);
    P.append(T);
} }
```

When bucket sorting the list P of arrays T of integers by the second component, array $T = [i, A[i]]$ belongs in bucket $A[i] - min + 1$.

442a ⟨bucket sort list P of pairs by $A[i]$ 442a⟩≡

```
while ( ¬P.empty() ) {
  T = P.pop();
  int k = T[2] − min + 1;    // T belongs in bucket k
  if ( k ⩾ 1 ∧ k ⩽ n ) {
    bucket[k].append(T);
  } else {
    error_handler(1,"radix sort: value out of range");
} }
for ( int i = 1; i ⩽ n; i++ )
  P.conc(bucket[i]);    // destructive
```

When bucket sorting the list P of arrays T of integers by the first component, on the other hand, array $T = [i, A[i]]$ belongs in bucket i.

442b ⟨bucket sort list P of pairs by i 442b⟩≡

```
while ( ¬P.empty() ) {
  T = P.pop();
  int k = T[1];    // T belongs in bucket k
  if ( k ⩾ 1 ∧ k ⩽ n ) {
    bucket[k].append(T);
  } else {
    error_handler(1,"radix sort: value out of range");
} }
for ( int i = 1; i ⩽ n; i++ )
  P.conc(bucket[i]);    // destructive
```

Distributing the list L of arrays A of integers into lists $LEN[i]$ of arrays of length i is straightforward. An array A of length i belongs in $LEN[i]$.

442c ⟨distribute arrays by length 442c⟩≡

```
array<list<array<int> > > LEN(1,maxlen);
while ( ¬L.empty() ) {
  A = L.pop();
  LEN[A.size()].append(A);
}
```

Now, bucket sorting the list L of arrays at component i can be done by first concatenating all arrays of length i at the front of L, then distributing L into buckets according to the integers at the ith component and, finally, concatenating all nonempty buckets back into L.

443a ⟨bucket sort list L of arrays on ith component 443a⟩≡

```
L.conc(LEN[i],LEDA::before);    // destructive
while ( ¬L.empty() ) {
  A = L.pop();
  int k = A[i] − min + 1;    // array A belongs in bucket k
  if ( k ⩾ 1 ∧ k ⩽ n ) {
    bucket[k].append(A);
  } else {
    error_handler(1,"radix sort: value out of range");
} }
int x;
forall(x,FULL[i])
  L.conc(bucket[x − min + 1]);    // destructive
```

Radix sort is often called just on a list of arrays of elements. A procedure call of the form *radix_sort*(L) is the same as the procedure call *radix_sort*(L,i,j), where i and j are respectively the minimum and maximum elements among all arrays in list L.

443b ⟨subroutines 40⟩+≡

```
void radix_sort(
  list<array<int> >& L)
{
  if ( L.length() ⩽ 1 ) return;
  if ( L.head().size() ≡ 0 ) return;
  int i = L.head()[1];
  int j = i;
  array<int> A;
  int x;
  forall(A,L) {
    forall(x,A) {
      i = leda_min(i,x);
      j = leda_max(j,x);
  } }
  radix_sort(L,i,j);
}
```

The following double-check of radix sort, although being redundant, gives some reassurance of the correctness of the implementation. It verifies that L is sorted in lexicographic order, that is, that $A \leqslant \mathrm{succ}(A)$ for all but the last array of integers A of the list of arrays of integers L.

443c ⟨double-check radix sort 443c⟩≡

```
{ list_item it;
  forall_items(it,L) {
    if ( it ≠ L.last() ∧ L[it] > L[L.succ(it)] ) {
      error_handler(1,"Wrong implementation of radix sort");
} } }
```

Remark A.9. Correctness of radix sorting follows from bucket sort being a stable sorting method.

Lemma A.10. *The algorithm for radix sorting runs in $O(n+j-i)$ time using $O(\max(n, j-i))$ additional space, where n is the total length of the arrays in list L and i and j are respectively the minimum and maximum elements among all arrays in list L.*

Proof. Let L be the list of m arrays of integers to be sorted, let n be the total length of the arrays in list L, and let i and j be respectively the minimum and maximum elements among all arrays in list L. Let also P be a list of pairs $\langle k, A[k]\rangle$ for all arrays A in list L.

List P is built in $O(n)$ time, and nonempty buckets are found in $O(n+j-i)$ time by two passes of bucket sort upon list P. Then, for each component k in the arrays, bucket sorting list L on the kth component takes time linear in the number of arrays A with length$[A] \geqslant k$ and therefore, bucket sorting list L on all array components takes a total of $O(n+j-i)$ time. Further, $O(\max(n, j-i))$ additional space is used for storing the array of buckets, which is shared by all passes of bucket sort.

The double-check of radix sorting makes m array comparisons, where each comparison takes time linear in the length of the arrays, and therefore runs in $O(n)$ time, using $O(1)$ additional space. □

A few implementation details still need to be filled in, though. The double-check of radix sort requires the lexicographic comparison of arrays of integers, and the following procedure defines, then, a default linear order for LEDA arrays of integers. Given two arrays of integers A_1 and A_2, it returns -1 if A_1 is smaller in lexicographic order than A_2, 0 if the arrays A_1 and A_2 are identical, and 1 if A_1 is larger in lexicographic order than A_2.

444 ⟨default procedures 444⟩≡

```
int compare(
  const array<int>& A1,
  const array<int>& A2)
{
  int n = leda_min(A1.size(),A2.size());
  for ( int i = 1; i ⩽ n; i++ ) {
    if ( A1[i] < A2[i] ) return −1;
    if ( A1[i] > A2[i] ) return 1;
  }
  if ( A1.size() < A2.size() ) return −1;
  if ( A1.size() > A2.size() ) return 1;
  return 0;
}
```

Now, the following Boolean comparison operator for arrays of integers makes it possible to test an array for being larger than another one.

445 ⟨default procedures 444⟩+≡

```
bool operator>(
  const array<int>& A1,
  const array<int>& A2)
{
  return compare(A1,A2) ≡ 1;
}
```

Bibliographic Notes

See [234, 236] for a comprehensive description of LEDA. The simple implementation of radix sort is based on [233, Sect. 2.2]. See also [8, 9, 36, 87, 195, 230, 373, 374].