Contents

5

Advanced Data Types

We discuss some of the advanced data types of LEDA: dictionary arrays, hashing arrays, maps, priority queues, partitions, and sorted sequences. For each type we give its functionality, discuss its performance and implementation, and describe applications.

5.1 Sparse Arrays: Dictionary Arrays, Hashing Arrays, and Maps

Sparse arrays are arrays with an infinite or at least very large index set of which only a "sparse" subset is in actual use. We discuss the sparse array types of LEDA and the many implementations available for them. We start with the functionality and then discuss the performance guarantees given by the different types and implementations. We also give an experimental comparison. We advise on how to choose an implementation satisfying the needs of a particular application and discuss the implementation of *maps* in detail.

5.1.1 *Functionality*

Dictionary arrays (type *d_array*$<I, E>$), hashing arrays (type *h_array*$<I, E>$), and maps (type *map*$<I, E>$) realize arrays with large or even unbounded index set I and arbitrary entry type E. Examples are arrays indexed by points, strings, or arbitrary integers. We refer to d_arrays, h_arrays, and maps as *sparse array types*; another common name is *associative arrays*. The sparse array types have different requirements for the index type: dictionary arrays work only for linearly ordered types (see Section 2.10), hashing arrays work only for hashed types (see Section 2.8), and maps work only for pointer and item types and the type int. They also differ in their performance guarantees and functionality. Figure 5.1 shows the manual page of maps and Table 5.1 summarizes the properties of our sparse array types. Before we discuss them we illustrate the sparse array types by small examples.

	d_arrays	h_arrays	Maps
index type	linearly ordered	hashed	int or pointer or item type
access time	$O(\log n)$ worst case	$O(1)$ expected	$O(1)$ expected
forall_defined loop	sorted	unsorted	unsorted
persistence of variables	yes	no	no
undefine operation	available	available	not available

Table 5.1 Properties of d_arrays, h_arrays, and maps. The meaning of the various rows is explained in the text.

In the first example we use a d_array to build a small English–German dictionary and to print all word pairs in the dictionary.

```
d_array<string,string> dic;
dic["hello"] = "hallo";
dic["world"] = "Welt";
dic["book"]  = "Buch";

string s;
forall_defined(s,dic) cout << s << " " << dic[s] << "\n";
```

The *forall_defined* loop iterates over all indices of the array that were used as a subscript prior to the loop. The iteration is according to the order defined by the *compare* function of the index type; recall that dictionary arrays work only for linearly ordered types. In the case of strings the default *compare* function defines the lexicographic ordering and hence the program outputs:

```
book Buch
hello hallo
world Welt
```

In the second example we use a h_array to read a sequence of strings from standard input, to count the multiplicity of each string in the input, and to output the strings together with their multiplicities. H_arrays work only for hashed types and hence we need to define a hash function for strings. We define a very primitive hash function that maps the empty string to zero and any non-empty string to its leading character (for a string x, $x[0]$ returns the leading character of x).

```
int Hash(const string& x) { return (x.length() > 0) ? x[0] : 0; }
h_array<string,int> N(0); // default value 0
while (cin >> s) N[s]++;
forall_defined(s,N) cout << s << " " << N[s] << "\n";
```

1. Definition

An instance M of the parameterized data type *map*$<I, E>$ is an injective mapping from the data type I, called the index type of M, to the set of variables of data type E, called the element type of M. I must be a pointer, item, or handle type or the type int. We use $M(i)$ to denote the variable indexed by i. All variables are initialized to *xdef*, an element of E that is specified in the definition of M. A subset of I is designated as the domain of M. Elements are added to $dom(M)$ by the subscript operator.

Related data types are *d_arrays*, *h_arrays*, and *dictionaries*.

2. Creation

map$<I, E>$ M;	creates an injective function m from I to the set of unused variables of type E, sets *xdef* to the default value of type E (if E has no default value then *xdef* is set to an unspecified element of E), and initializes M with m.
map$<I, E>$ $M(E\ x)$;	creates an injective function m from I to the set of unused variables of type E, sets *xdef* to x, and initializes M with m.

3. Operations

$E\&$	$M[I\ i]$	returns the variable $M(i)$ and adds i to $dom(M)$. If M is a const-object then $M(i)$ is read-only and i is not added to $dom(M)$.
bool	M.defined($I\ i$)	returns true if $i \in dom(M)$.
void	M.clear( )	makes M empty.
void	M.clear($E\ x$)	makes M empty and sets *xdef* to x.

Iteration

forall_defined(i, M) { "the indices i with $i \in dom(M)$ are successively assigned to i" }

forall(x, M) { "the entries $M[i]$ with $i \in dom(M)$ are successively assigned to x" }

4. Implementation

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

Figure 5.1 The manual page of data type *map*.

There are two further remarks required about this code fragment. First, in the definition of N we defined a default value for all entries of N: all entries of N are initialized to this default value. Second, hashed types have no particular order defined on their elements and hence the *forall_defined* loop for h_arrays steps through the defined indices of the array in no particular order.

In the third example we assume that we are given a list of segments in *seglist* and that we want to associate a random bit with each segment. A *map<segment, bool>* serves well for this purpose.

```
map<segment,bool> color;

segment seg;
forall(seg,seglist) color[seg] = rand_int(0,1);
```

After these introductory examples we turn to the detailed discussion of our sparse array types. An object A of a sparse array type is characterized by three quantities:

- An injective mapping from the index type into the variables of type E. For an index i we use $A(i)$ to denote the variable selected by i.
- An element *xdef* of type E, the default value of all variables in the array. It is determined in one of three ways. If the definition of the array has an argument, as, for example, in

  ```
  h_array<int,int> N(0);
  ```

  then this argument is *xdef*. If the definition of the array has no argument but the entry type of the array has a default value[1], as, for example, in

  ```
  d_array<string,string> D;
  ```

  then this default value is *xdef*. If the definition of the array has no argument and the entry type of the array has no default value, as, for example, in

  ```
  map<point,int> color;
  ```

  then *xdef* is some arbitrary value of E. This value may depend on the execution history.
- A subset $dom(A)$ of the index set, the so-called *domain* of A. All variables outside the domain have value *xdef*. Indices are added to the domain by the subscript operation and are deleted from the domain by the *undefine* operation. Maps have no *undefine* operation and put some indices in the domain even if they were not accessed[2]. D_arrays and h_arrays start with an empty domain and indices are added to the domain only by the subscript operation.

We come to the operations defined on sparse arrays. We assume that A belongs to one of our sparse array types and that I is a legal index type for this sparse array type as defined in the first row of Table 5.1. The subscript operator *operator*[] comes in two kinds:

```
const E& operator[](const I& i) const
E&       operator[](const I& i)
```

[1] This is the case for all but the built-in types of C++.

[2] These indices are used as sentinels in the implementation and allow us to make maps faster than the other sparse array types. We refer the reader to Section 5.2 for details.

The first version applies to const-objects and the second version applies to non-const-objects. Both versions return the variable $A(i)$. The first version allows only read access to the variable and the second version also allows us to modify the value of the variable. The second version adds i to the domain of A and the first version does not. How is the selection between the two versions made? Recall that in C++ every member function of a class X has an implicit argument referring to an instance of the object. This implicit argument has type `const X<I,E>*` for the first version of the subscript operator and has type `X<I,E>*` for the second version of the access operator; here X stands for one of the sparse array types. Thus depending on whether the subscript operator is applied to a constant sparse array or a modifiable sparse array either the first or the second version of the subscript operator is selected. Consider the following examples.

```
const map<int,int> M1;
      map<int,int> M2;
int x;
x = M1[5];                         // first  version
x = M2[5];                         // second version
x = ((const map<int,int>) M2)[7];  // first  version
```

Observe that the first version of the subscript operator is used in the first and the last call since *M1* is a constant map and since *M2* is cast to a constant map in the last line. The second version of the subscript operator is used in the second access. It is tempting but wrong to say (Kurt has made this error many times) that the use of the variable $A(i)$ dictates the selection: an access on the left-hand side of an assignment uses the second version (since the type `E&` is needed) and an access on the right-hand side of an assignment uses the second version (since the type `const E&` suffices). We emphasize, *the rule just stated is wrong*. In C++ the return type of a function plays no role in the selection of a version of an overloaded function; the selection is made solely on the basis of the argument types. We continue the example above.

```
    x = M2[5];                     // second version
M2[5] = x;                         // second version
    x = M1[5];                     // first version
M1[5] = x;                         // first version, illegal
```

The last assignment is illegal, since the first version of the access operator is selected for the constant map *M1*. It returns a constant reference to the variable *M1*(5), to which no assignment is possible.

```
bool A.defined(I i)
```

returns true if $i \in dom(A)$ and returns false otherwise. Finally, the operation

```
void A.undefine(I i)
```

removes i from $dom(A)$ and sets $A(i)$ to *xdef*. This operation is not available for maps.

Sparse arrays offer an iteration statement

```
forall_defined(i,A)
{ the elements of dom(A) are successively assigned to i }
```

which iterates over the indices in *dom*(A). In the case of d_arrays the indices are scanned in increasing order (recall that the index type of a d_array must be linearly ordered), in the case of h_arrays and maps the order is unspecified. The iteration statement

```
forall(x,A)
{ A[i] for i in dom(A) is successively assigned to x }
```

iterates over the values of the entries in *dom*(A).

5.1.2 *Performance Guarantees and Implementation Parameters*

Sparse arrays are one of the most widely studied data type and many different realizations with different performance guarantees have been proposed for them. We have included several into the LEDA system and give the user the possibility to choose an implementation through the implementation parameter mechanism.

```
_d_array<string,int,rs_tree> D1(0);
_d_array<string,int,rb_tree> D2(0);
_d_array<int,   int,dp_hashing> H;
```

defines three sparse arrays realized by randomized search trees, red-black trees, and dynamic perfect hashing, respectively. We now survey the available implementations; see also Tables 5.2 and 5.3. The implementations fall into two classes, those requiring a linearly ordered index type and those requiring a hashed index type. We use n to denote the size of the domain of the sparse array.

Implementations requiring a Linearly Ordered Index Type: This class of implementations contains deterministic and randomized implementations. The deterministic implementations are (a,b)-trees [Meh84a], *AVL*-trees [AVL62], *BB*$[\alpha]$-trees [NR73, BM80, Meh84a], red-black-trees [GS78, Meh84a], and unbalanced trees. The corresponding implementation parameters are *ab_tree*, *avl_tree*, *bb_tree*, *rb_tree*, and *bin_tree*, respectively. Except for unbalanced trees, all deterministic implementations guarantee $O(\log n)$ insertion, lookup, and deletion time. The actual running times of all deterministic implementations (except for unbalanced trees) are within a factor of two to three of one another. The unbalanced tree implementation can deteriorate to linear search and guarantees only linear insertion, lookup, and deletion time, as is clearly visible from the right part of Table 5.2. It should not be used.

The randomized implementations are skiplists [Pug90b] (*skiplist*) and randomized search trees [AS89] (*rs_tree*). Both implementations guarantee an expected insertion, deletion, and lookup time of $O(\log n)$. The expectations are taken with respect to the internal coin flips of the data structures.

Among the implementations requiring a linearly ordered index type ab-trees and skiplists

	Random integers				Sorted integers			
	insert	lookup	delete	total	insert	lookup	delete	total
ch_hash	0.23	0.09	0.18	0.5	0.2	0.05	0.12	0.37
dp_hash	1.48	0.21	1.08	2.77	1.37	0.21	1.02	2.6
map	0.15	0.04	—	0.19	0.15	0.05	—	0.2
skiplist	0.78	0.54	0.54	1.86	0.43	0.16	0.14	0.73
rs_tree	1.04	0.71	0.76	2.51	0.42	0.19	0.2	0.81
bin_tree	0.83	0.59	0.62	2.04	2704	1354	0.1501	4058
rb_tree	0.9199	0.54	0.74	2.2	0.6499	0.1802	0.3	1.13
avl_tree	0.8599	0.55	0.7	2.11	0.45	0.2	0.2402	0.8901
bb_tree	1.23	0.52	1	2.75	0.6399	0.2	0.3301	1.17
ab_tree	0.5898	0.25	0.4502	1.29	0.22	0.1399	0.2	0.5598
array					0.01001	0.01001	—	0.02002

Table 5.2 The performance of various implementations of sparse arrays. Hashing with chaining (*ch_hash*) and dynamic perfect hashing (*dp_hash*) are implementations of h_arrays, *map* is the implementation of map, and skiplists (*skiplist*), randomized search trees (*rs_tree*), unbalanced binary trees (*bin_tree*), red-black-trees (*rb_tree*), AVL-trees (*avl_tree*), BB[α]-trees (*bb_tree*), and 2-4-trees (*ab_trees*) are implementations of d_arrays. Running times are in seconds. We performed 10^5 insertions followed by 10^5 lookups followed by 10^5 deletions. We used random keys of type *int* in $[0 .. 10^7]$ for the left half of the table and we used the keys $0, 1, 2, \ldots$ for the right half of the table. Maps are the fastest implementation followed by hashing with chaining. Among the implementations of *d_arrays* ab-trees and skiplists are currently the most efficient. Observe the miserable performance of the *bin_tree* implementation for the sorted insertion order. For comparison we also included arrays for the second test.

are currently the most efficient. We give the details of the skiplist implementation in Section 5.7.

All implementations use linear space, e.g., the skiplist implementation requires $76n/3 + O(1) = 25.333n + O(1)$ bytes.

Implementations requiring a Hashed Index Type: There are two implementations: Hashing with chaining and dynamic perfect hashing.

Hashing with chaining is a deterministic data structure. Figure 5.2 illustrates it. It consists of a table and a singly linked list for each table entry. The table size T is a power of two such that $T = 1024$ if $n < 1024$ and $T/2 \leq n \leq 2T$ if $n \geq 1024$. The i-th list contains all x in the domain of the sparse array such that $i = \mathit{Hash}(x) \bmod T$. Let l_i be the number of

Random doubles				
	insert	lookup	delete	total
skiplist	3.09	2.36	1.95	7.4
rs_tree	3.81	2.69	2.48	8.98
bin_tree	2.85	1.94	2.15	6.94
rb_tree	2.75	1.82	2.28	6.85
avl_tree	2.82	1.89	2.24	6.95
bb_tree	4.06	1.88	3.81	9.75
ab_tree	2.09	1.51	1.61	5.21

Table 5.3 The performance of various implementations of sparse arrays. Running times are in seconds. We performed 10^5 insertions followed by 10^5 lookups followed by 10^5 deletions. We used random keys of type *double* in $[0 \,..\, 2^{31}]$.

elements in the i-th list and let k be the number of empty lists. The space requirement for hashing with chaining is $12(n+k)$ bytes.

We justify this formula. An item in a singly linked list requires twelve bytes; four bytes for the pointer to the successor and four bytes each for the key and the information (if a key or information does not fit into four bytes the space for the key or information needs to be added, see Section 13.4). There are T list items in the table and $l_i - 1$ extra items in the i-th list, if $l_i \geq 1$. Next observe that

$$\sum_{i;l_i \geq 1} (l_i - 1) = \sum_i (l_i - 1) + k = n - T + k.$$

The space required is therefore $12(T + n - T + k) = 12(n+k)$ bytes.

If the hash function behaves like a random function, i.e., its value is a random number in $[0 \,..\, T-1]$, the probability that the i-th list is empty is equal to $(1 - 1/T)^n$ and hence the expected value of k is equal to $T(1-1/T)^n = T(1-1/T)^{T(n/T)} \approx Te^{-n/T}$; here, we used the approximation $(1 - 1/T)^T \approx e^{-1}$. The expected space requirement of hashing with chaining is therefore equal to $12(n + Te^{-n/T})$ bytes. The time to search for an element x, to insert it, or to delete it is $O(1)$ plus the time to search in the linear list to which x is hashed. The latter time is linear in the worst case. For random indices the expected length of each list is n/T and hence all operations take constant expected time for random indices.

After an insertion or deletion it is possible that the invariant relating T and n is violated. In this situation a so-called *rehash* is performed, i.e., the table size is doubled or halved and all elements are moved to the new table.

Dynamic perfect hashing [FKS84, DKM$^+$94] uses randomization. It is the implementation with the theoretically best performance. The operation *defined* takes constant worst

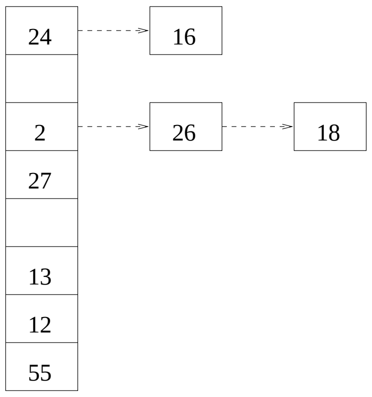


Figure 5.2 Hashing with chaining: The table size is 8 and the domain of the sparse array is $\{2, 12, 13, 16, 18, 24, 26, 27, 55\}$. The hash function $H(x)$ is the identity function $H(x) = x$ and hence any number x is stored in the list with index $x \bmod 8$.

case time and the operation $A[i]$ takes constant expected amortized time or constant worst case time depending on whether it is the first access with index i or not. This requires some explanation. Dynamic perfect hashing uses a two-level hashing scheme. A first-level hash function hashes the domain to some number T of buckets. T is chosen as in the case of hashing with chaining. As above, let l_i be the number of elements in the domain that are hashed to the i-th bucket. In the second level a separate table of size l_i^2 is allocated to the i-th bucket and a perfect hash function is used to map the elements in the i-th bucket to their private table, see Figure 5.3. In [FKS84, DKM$^+$94] it is shown that suitable hash functions exist and can be found by random selection from a suitable class of hash functions. It is also shown in these papers that the space requirement of the scheme is linear, although with a considerably larger constant factor than for hashing with chaining. An access operation requires the evaluation of two hash functions and hence takes constant time in the worst case. An insertion (= first access to $A[i]$ for some index i) may require a rehash on either the second level or the first level of the data structure. Rehashes are costly but rare and hence the expected amortized time for an insert or delete is constant.

Experiments show that hashing with chaining is usually superior to dynamic perfect hashing and hence we have chosen hashing with chaining as the default implementation of *h_array*<I, E>.

Maps: Maps are implemented by hashing with chaining. Since the index type of a map must be an item or pointer type or the type int and since maps do not support the *undefine* operation, three optimizations are possible with respect to hashing with chaining as described above. First, items and pointers are interpreted as integers and the identity function is used as the hash function, i.e., an integer x is hashed to $x \bmod T$ where T is the table size. Since T is chosen as a power of two, evaluation of this hash function is very fast. Second, the list elements are not allocated in free store but are all stored in an array. This allows

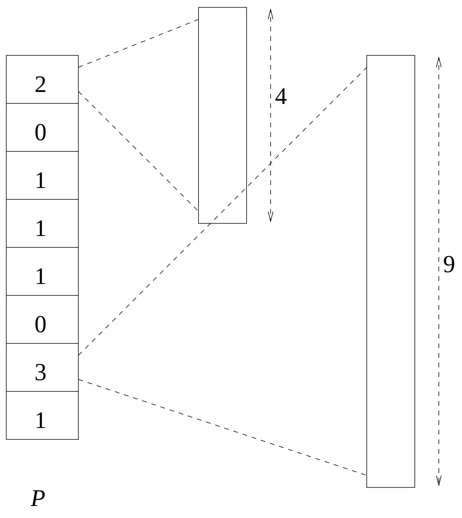


Figure 5.3 Dynamic perfect hashing: The first-level table P has size 8. For each entry of this table the number of elements hashed to this entry are indicated. If l, $l > 1$, elements are hashed to an entry then a second-level table of size l^2 is used to resolve the collisions. The sizes of the two second-level tables that are required in our example are also indicated.

for a faster realization of the rehash operation. Third, since the keys are integers a particularly efficient implementation of the access operation is possible. Section 5.2 contains the complete implementation of maps.

An Experimental Comparison: We give an experimental comparison of all sparse array types. We perform three kinds of experiments. In the first one, we use random integer keys in the range $[0 .. 10^7]$, in the second one, we use the keys 0, 1, ..., and in the third one, we use random double keys. In each case we perform 10^5 insertions, followed by 10^5 lookups, followed by 10^5 deletions. Tables 5.2 and 5.3 summarize the results.

The following program performs the first two experiments and generates Table 5.2. In the main program we first define sparse arrays, one for each implementation, and two arrays A and B of size 10^5. We fill A with random integers and we fill B with the integers 0, 1, ... Then we call the function *dic_test* for each sparse array; *dic_test* first inserts $A[0]$, $A[1]$, ..., then looks up $A[0]$, $A[1]$, ..., and finally deletes $A[0]$, $A[1]$, ... It then performs the same sequence of operations with B instead of A. For each sparse array type it produces a row of Table 5.2. The chunks ⟨*map test*⟩ and ⟨*array test*⟩ perform the same tests for maps[3] and arrays, respectively. We leave their details to the reader.

⟨*dic_performance.c*⟩≡

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

[3] Since maps do not support delete operations, we need two maps *M1* and *M2*, one for the experiment with A and one for the exeriment with B.

```
#include <LEDA/IO_interface.h>
#include <LEDA/impl/ch_hash.h>
#include <LEDA/impl/dp_hash.h>
#include <LEDA/impl/avl_tree.h>
#include <LEDA/impl/bin_tree.h>
#include <LEDA/impl/rs_tree.h>
#include <LEDA/impl/rb_tree.h>
#include <LEDA/impl/skiplist.h>
#include <LEDA/impl/ab_tree.h>
#include <LEDA/impl/bb_tree.h>
int N;
int* A; int* B;
IO_interface I;
void dic_test(d_array<int,int>& D, string name)
{
  I.write_table("\n " + name);
  float T; float T0 = T = used_time();
  int i;
  for(i = 0; i < N; i++)  D[A[i]] = 0;
  I.write_table(" & ",used_time(T));
  for(i = 0; i < N; i++)  int* ptr = &D[A[i]];
  I.write_table(" & ",used_time(T));
  for(i = 0; i < N; i++)  D.undefine(A[i]);
  I.write_table(" & ",used_time(T));
  I.write_table(" & ",used_time(T0));
  ⟨same for B⟩
}
⟨map test⟩
int main()
{
  _d_array<int,int,ch_hash> CHH_DIC;
  _d_array<int,int,dp_hash> DPH_DIC;
  map<int,int> M1, M2;
  _d_array<int,int,avl_tree> AVL_DIC;
  _d_array<int,int,bin_tree> BIN_DIC;
  _d_array<int,int,rb_tree>  RB_DIC;
  _d_array<int,int,rs_tree>  RS_DIC;
  _d_array<int,int,skiplist> SK_DIC;
  _d_array<int,int,bb_tree>  BB_DIC;
  _d_array<int,int,ab_tree>  AB_DIC;
  N  = 100000;
  A = new int[N]; B = new int[N];
  int i;
  for(i = 0; i < N; i++) { A[i] = rand_int(0,10000000); B[i] = i; }
  dic_test(CHH_DIC,"ch\\_hash");
  dic_test(DPH_DIC,"dp\\_hash"); I.write_table(" \\hline");
  map_test(M1,M2,  "map");       I.write_table(" \\hline");
  dic_test(SK_DIC, "skiplist");
```

```
    dic_test(RS_DIC ,"rs\\_tree");
    dic_test(BIN_DIC,"bin\\_tree");
    dic_test(RB_DIC ,"rb\\_tree");
    dic_test(AVL_DIC,"avl\\_tree");
    dic_test(BB_DIC ,"bb\\_tree");
    dic_test(AB_DIC ,"ab\\_tree"); I.write_table(" \\hline");
    ⟨array test⟩
  }
```

5.1.3 *Persistence of Variables*

We stated above that an access operation

```
E& A[I i]
```

returns the variable $A(i)$. Thus, one can write

```
E& x = A[5];
<some statements not touching A[5]>;
A[5] = y;
if ( x == y ) { .... }
```

and expect that the test $x == y$ returns true. This is not necessarily the case for h_arrays and maps as these types do not guarantee that different accesses to $A[5]$ return the same variable and *we therefore recommend never to establish a pointer or a reference to a variable contained in a map or h_array.* Given the efficiency of h_arrays and maps there is really no need to do so. The fact that the identity of variables is not preserved is best explained by recalling the implementation of h_arrays and maps. They use an array of linked lists where the size of the array is about the size of the domain of the sparse array. Whenever the invariant linking the size of the table and the size of the domain is violated the content of the sparse array is rehashed. In the process of rehashing new variables are allocated for some of the entries of the sparse array. Of course, the values of the entries are moved to the new variables. Thus, the content of $A(i)$ is preserved but not the variable $A(i)$.

D_arrays behave differently. Variables in d_arrays are persistent, i.e, the equality test in the code sequence above is guaranteed to return true.

5.1.4 *Choosing an Implementation*

LEDA gives you the choice between many implementations of sparse arrays. Which is best in a particular situation?

Tables 5.2 and 5.3 show that in certain situations maps are faster than h_arrays which in turn are faster than d_arrays. On the other hand the slower data types offer an increased functionality. This suggests using the type whose functionality just suffices in a particular application.

There are, however, other considerations to be taken into account. Maps and h_arrays perform well only for random inputs, they can perform miserably for non-random inputs. For maps a bad example is easily constructed. Use the indices $1024i$ for $i = 0, 1, \ldots$.

Since maps use the hash function $x \longrightarrow x \bmod T$ where T is the table size, and T is always a power of two these keys will not be distributed evenly by the hash function and hence the performance of maps will be much worse than for random inputs. In the case of h_arrays the situation is not quite as bad since you may overwrite the default hash function. For example, you may want to use

```
int Hash(int x){ return x/1024; }
```

if you know that the indices are multiples of 1024.

Which implementations are we using ourselves? We usually use maps to associate information with item types such as points and segments, we use d_arrays or dictionaries when the order on the indices is important for the application, and we use h_arrays when we know a hash function suitable for the application.

If you are not happy with any of the implementations provided in LEDA you may provide your own. Section 13.6 explains how this is done.

5.2 The Implementation of the Data Type Map

We give the complete implementation of the data type *map*. This section is for readers who want to understand the internals of LEDA. Readers that "only" want to use LEDA may skip this section without any harm.

We follow the usual trichotomy in the definition of LEDA's parameterized data types as explained in Section 13.4. Familiarity with this section is required for some of the fine points of this section. We define two classes, namely the abstract data type class *map*$<I, E>$ and the implementation class *ch_map*, in three files, namely map.h, ch_map.h, and _ch_map.c. The abstract data type class has template parameters I and E and the implementation class stores *GenPtrs* (= *void*$*$). In map.h we define the abstract data type class and implement it in terms of the implementation class. This implementation is fairly direct; its main purpose is to translate between the untyped view of the implementation class and the typed view of the abstract data type class. In ch_map.h and _ch_map.c, respectively, we define and implement the implementation class.

We first give the global structure of LEDAROOT/incl/LEDA/map.h.

$\langle$*map.h*$\rangle+\equiv$

```
template<class I, class E>
class map : private ch_map {

 E xdef;

 void copy_inf(GenPtr& x)   const { LEDA_COPY(E,x);  }
 void clear_inf(GenPtr& x)  const { LEDA_CLEAR(E,x); }
 void init_inf(GenPtr& x)   const { x = leda_copy((E&)xdef); }

 public:
   typedef ch_map::item item;
   ⟨member functions of map⟩
  };
```

We give some explanations. We derive the abstract data type class *map* from the implementation class *ch_map* and give it an additional data member *xdef*, which stores the default value of the variables of the map. Therefore, an instance of *map* consists of an instance of *ch_map* and a variable *xdef* of type E. The private function members *copy_inf*, *clear_inf*, and *init_inf* correspond to virtual functions of the implementation class and redefine them. The first two are required by the LEDA method for the implementation of parameterized data types and are discussed in Section 13.4. The third function is used to initialize an entry to a copy of *xdef*.

The public member functions will be discussed below. They define the user interface of maps as given in Table 5.1.

We come to our implementation class *ch_map*. It is based on the data structure hashing with chaining. Hashing with chaining uses an array of singly linked lists and therefore we introduce a container for list elements, which we call *ch_map_elem*. A *ch_map_elem* stores an unsigned long k, a generic pointer i, and a pointer to the successor container. We refer to k as the key-field and to i as the inf-field of the container. This nomenclature is inspired by dictionaries. Keys correspond to indices (type I) in the abstract data type class and infs correspond to elements (type E) in the abstract data type class.

A pointer to a *ch_map_elem* is called a *ch_map_item*.

The flag `__exportC` is used during a precompilation step. On UNIX-systems it is simply deleted and on Windows-systems it is replaced by appropriate key words that are needed for the generation of dynamic libraries.

⟨*ch_map_elem*⟩≡

```
class __exportC ch_map_elem
{
  friend class __exportC ch_map;
  unsigned long    k;
  GenPtr           i;
  ch_map_elem*  succ;
};
typedef ch_map_elem*  ch_map_item;
```

Next we discuss the data members of the implementation class.

⟨*data members of ch_map*⟩≡

```
ch_map_elem STOP;
ch_map_elem* table;
ch_map_elem* table_end;
ch_map_elem* free;
int table_size;
int table_size_1;
```

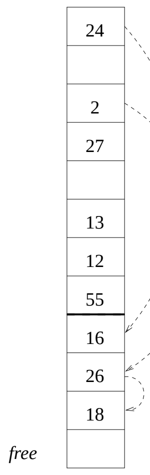


Figure 5.4 A hash table of size 12. The last four locations are used as an overflow area and the first eight locations correspond to eight linear lists. The set stored is $\{2, 12, 13, 16, 18, 24, 26, 27, 55\}$ and any number x is stored in the list with index $x \bmod 8$. If the i-th list contains more than one element then the first element is stored in the i-th table position and all other elements are stored in the overflow area. In the example, three elements are hashed to the second list and hence two of them are stored in the overflow area. The variable *free* points to the first free position in the overflow area.

We use a *table* of map elements of size $f \cdot T$ where T is a power of two and f is a number larger than one, see Figure 5.4. We use $f = 1.5$ in our implementation. The first T elements of the table correspond to the headers of T linear lists and the remaining $(f - 1)T$ elements of the table are used as an overflow area to store further list elements. The variable *free* always points to the first unused map element in the overflow area. When the overflow area is full we move to a table twice the size. We use *table_size* to store T and *table_size_1* to store $T - 1$.

The main use of maps is to associate information with objects. Thus the most important operation for maps is the access operation with keys that are already in the table (the data structure literature calls such accesses *successful searches*) and we designed maps so that successful searches are particularly fast. An access for a key x involves the evaluation of a hash function plus the search through a linear list. Our hash function simply extracts the last log *table_size* bits from the binary representation of x.

$\langle$*HASH function*$\rangle\equiv$

```
ch_map_elem*  HASH(unsigned long x)  const
{ return table + (x & table_size_1);  }
```

Why do we dare to take such a simple hash function? Let U be the set of unsigned longs. We assume, as is customary in the analysis of hashing, that a random subset $S \subseteq U$ of size n is stored in the hash table. Let $m = table\_size$ denote the size of the hash table and for all

i, $0 \le i < m$, let s_i be the number of elements in S that are hashed to position i. Then

$$s_0 + s_1 + \ldots + s_{m-1} = n$$

and hence

$$\mathrm{E}[s_0] + \mathrm{E}[s_1] + \ldots + \mathrm{E}[s_{m-1}] = n$$

by linearity of expectations. A hash function is called *fair* if the same number of elements of U are hashed to every table position. Our hash function is fair. For a fair hash function symmetry implies that the expectations of all the s_i's are the same. Hence

$$\mathrm{E}[s_i] = n/m$$

for all i. No hash function can do better since $\sum_i \mathrm{E}[s_i] = n$. We conclude that any fair hash function yields the optimal expectations for the $\mathrm{E}[s_i]$. For the sake of speed the simplest fair hash function should be used. This is exactly what we do.

We mentioned already that our main goal was to make access operations as fast as possible. We will argue in the next three paragraphs that most successful accesses are accesses to elements which are stored in the first position of the list containing them. Let k denote the number of empty lists. Then $T - k$ lists are non-empty and hence there are $T - k$ elements which are first in their list. If n denotes the number of elements stored in the table the fraction of elements that are first in their list is $(T - k)/n$. We want to estimate this fraction for random keys and immediately before and after a rehash. We move to a new table when the overflow area is full. At this time, there are $(f - 1)T$ elements stored in the overflow area and $T - k$ elements in the first T positions of the table. Thus $n = fT - k$ at the time of a rehash.

For random keys the expected number of empty lists is $k = T \cdot (1 - 1/T)^n \approx Te^{-n/T}$. For random keys we will therefore move to a new table when $n \approx T \cdot (f - e^{-n/T})$ or $n/T + e^{-n/T} \approx f$. For $f = 1.5$ we get $n \approx 1.2T$, i.e., when about $1.2T$ elements are stored in the table we expect to move to a new table.

When $n \approx 1.2T$ about $0.7T$ elements are stored in the first T slots of the table and about $0.5T$ elements are stored in the overflow area of the table. Thus about $0.7/1.2 \approx 58\%$ of the successful searches go to the first element in a list. Immediately after a rehash we have $n \approx 0.6T$ (since $n \approx 1.2T$ before the rehash and a rehash doubles the table size) and the expected number of empty lists is $Te^{-0.6} \approx 0.55T$. Thus $0.45/0.6 \approx 75\%$ of the successful searches go to the first element in a list. In either case a significant fraction of the successful searches goes to the first element in a list.

How can we make accesses to first elements fast? A key problem is the encoding of empty lists. We explored two possibilities. In both solutions we use a special list element *STOP* as a sentinel. In the first solution we maintain the invariant that the i-th list is empty if the successor field of *table*$[i]$ is nil and that the last entry of a non-empty list points to *STOP*. This leads to the following code for an access operation:

```
inline GenPtr& ch_map::access(unsigned long x)
{ ch_map_item p = HASH(x);
  if ( p->succ == nil)
    { p->k = x;
      init_inf(p->i);   // initializes p->i to xdef
      p->succ = &STOP;
      return p->i;
    }
   else
    { if ( p->k == x ) return p->i;
    }
    return access(p,x);
 }
```

In this code, *access*(p, x) handles the case that the list for x is non-empty and that the first element does not contain x. This code has two weaknesses. First, it tests each list for emptiness although successful searches always go to non-empty lists and, second, it needs to change the successor pointer of *table*$[i]$ to &*STOP* after the first insert into the i-th list.

In the second solution we encode the fact that the i-th list is empty in the key field of *table*$[i]$. Let NULLKEY and NONNULLKEY be keys that are hashed to zero and some non-zero value, respectively. In our implementation we use 0 for NULLKEY and 1 for NONNULLKEY. We use the special keys NULLKEY and NONNULLKEY to encode empty lists. More specifically, we maintain:

- *table*$[0].k =$ NONNULLKEY, i.e., the first entry of the zero-th list is unused. The information field of this entry is arbitrary.
- *table*$[i].k =$ NULLKEY iff the i-th list is empty for all i, $i > 0$, and
- the last entry of a non-empty list points to *STOP* and if the i-th list is empty then *table*$[i]$ points to *STOP*.

Observe that the zero-th list is treated somewhat unfairly. We leave its first position unused and thus make it artificially non-empty. Figure 5.5 illustrates the items above.

Consider a search for x and let p be the hash-value of x. If x is stored in the first element of the p-th list we have a successful search, and the p-th list is empty iff the key of the first element of the p-th list is equal to NULLKEY. Observe that this is true even for p equal to zero, because the first item guarantees that NULLKEY is not stored in the first element of list 0. We obtain the following code for the access operation:

⟨*inline functions*⟩≡

```
inline GenPtr& ch_map::access(unsigned long x)
{ ch_map_item p = HASH(x);
  if ( p->k == x ) return p->i;
  else
  { if ( p->k == NULLKEY )
    { p->k = x;
      init_inf(p->i);  // initializes p->i to xdef
```

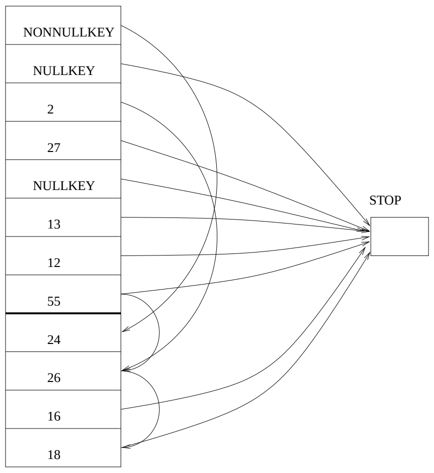


Figure 5.5 The realization of the hash table of Figure 5.4 in *ch_map*. The first entry of the zero-th list containts `NONNULLKEY` (whether the zero-th list is empty or not), empty lists other than the zero-th list contain `NULLKEY` in their first element, and each list points to `STOP`.

```
      return p->i;
    }
    else
      return access(p,x);
  }
}
```

Note that a successful search for a key x that is stored in the first position of its list is very fast. It evaluates the hash function, makes one equality test between keys, and returns the information associated with the key. If x is not stored in the first position of its table, we need to distinguish cases: if the list is empty we store $(x, xdef)$ in the first element of the list (note that the call $init\_inf(p \to i)$ sets the inf-field of p to *xdef*), and if the list is non-empty we call $access(p, x)$ to search for x in the remainder of the list. We will discuss this function below.

Our experiments show that the second design is about 10% faster than the first and we therefore adopted it for maps. In the implementation of h_arrays by hashing with chaining we use the first solution. Since h_arrays use non-trivial hash functions that may require substantial time for their evaluation, the second solution looses its edge over the first in the case of h_arrays.

We can now give an overview over LEDAROOT/incl/LEDA/impl/ch_map.h.

⟨*ch_map.h*⟩≡
```
#ifndef LEDA_CH_MAP_H
#define LEDA_CH_MAP_H
#include <LEDA/basic.h>
```
⟨*ch_map_elem*⟩

```
class __exportC ch_map
{
   const unsigned long NULLKEY;
   const unsigned long NONNULLKEY;
   ⟨data members of ch_map⟩
   virtual void clear_inf(GenPtr&)  const { }
   virtual void copy_inf(GenPtr&)   const { }
   virtual void init_inf(GenPtr&)   const { }
   ⟨HASH function⟩
   ⟨private member functions of ch_map⟩
   protected:
   typedef ch_map_item item;
   ⟨protected member functions of ch_map⟩
};
⟨inline functions⟩
#endif
```

We have already explained the data members. The virtual function members *clear_inf*, *copy_inf*, and *init_inf* are required by the LEDA method for the implementation of parameterized data types. We saw already how they are redefined in the definition of *map*.

The protected and private member functions will be discussed below. The protected member functions are basically in one-to-one correspondence to the public member functions of the abstract data type class and the private member functions define some basic functionality that is needed for the protected member functions, e.g., rehashing to move to a larger table.

We come to the file LEDAROOT/src/dic/_ch_map.c. There is little to say about it at this point except that is contains the implementation of class *ch_map*.

⟨*_ch_map.c*⟩≡
```
#include <LEDA/impl/ch_map.h>
⟨implementation of ch_map⟩
```

Having defined all data members and the global structure of all files we can start to implement functions. We start with the private members of *ch_map*.

⟨*private member functions of ch_map*⟩≡
```
void init_table(int T);
```

initializes a table of size T (T is assumed to be a power of two) and makes all lists (including list zero) empty. This is trivial to achieve. We allocate a new table of size fT and set all data members accordingly. We also initialize $table[0].k$ to `NONNULLKEY`, $table[i].k$ to `NULLKEY` for all i, $1 \le i < table\_size$, and let $table[i].succ$ point to `STOP` for all i, $0 \le i < table\_size$. This initializes all lists to empty lists.

⟨*implementation of ch_map*⟩≡

```
void ch_map::init_table(int T)
{
  table_size = T;
  table_size_1 = T-1;
  table = new ch_map_elem[T + T/2];
  free = table + T;
  table_end = table + T + T/2;
  for (ch_map_item p = table; p < free; p++)
  { p->succ = &STOP;
    p->k = NULLKEY;
  }
  table->k = NONNULLKEY;
}
```

⟨*private member functions of ch_map*⟩+≡

```
void rehash();
```

moves to a table twice the current size. We do so by first moving all elements stored in the first T elements of the table and then all elements in the overflow area. Note that this strategy has two advantages over moving the elements list after list: First, we do not have to care about collisions when moving the elements in the first T table positions (because the element in position i is moved to either position i or $T + i$ in the new table depending on the additional bit that the new hash function takes into account), and second, locality of reference is better (since we move all elements by scanning the old table once).

When moving the elements from the overflow area we make use of the member function *insert*. We define it inline. It takes a pair (x, y) and moves it to the list for key x. If the first element of the list is empty, we move (x, y) there, and if the first element is non-empty, we move (x, y) to position *free*, insert it after the first element of the list, and increment *free*.

⟨*private member functions of ch_map*⟩+≡

```
inline void insert(unsigned long x, GenPtr y);
```

⟨*implementation of ch_map*⟩+≡

```
inline void ch_map::insert(unsigned long x, GenPtr y)
{ ch_map_item q = HASH(x);
  if ( q->k == NULLKEY )
    { q->k = x;
      q->i = y;
    }
  else
   { free->k = x;
     free->i = y;
     free->succ = q->succ;
     q->succ = free++;
   }
}
```

In *rehash* we first initialize the new table (this puts NONNULLKEY into the first entry of the zero-th list) and then move elements. We first move the elements in the main part of the table (*table*[0] is unused and hence the loop for moving elements starts at *table* + 1) and then the elements in the overflow area.

⟨*implementation of ch_map*⟩+≡

```
void ch_map::rehash()
{
  ch_map_item old_table = table;
  ch_map_item old_table_mid = table + table_size;
  ch_map_item old_table_end = table_end;
  init_table(2*table_size);
  ch_map_item p;
  for(p = old_table + 1; p < old_table_mid; p++)
  { unsigned long x = p->k;
    if ( x != NULLKEY ) // list p is non-empty
    { ch_map_item q = HASH(x);
      q->k = x;
      q->i = p->i;
    }
  }
  while (p < old_table_end)
  { unsigned long x = p->k;
    insert(x,p->i);
    p++;
  }
  delete[] old_table;
}
```

⟨*private member functions of ch_map*⟩+≡

```
GenPtr& access(ch_map_item p, unsigned long x);
```

searches for x in the list starting at p. The function operates under the precondition that the list is non-empty and x is not stored in p. The function is called by the inline function *access*(x).

We search down the list starting at p. If the search reaches STOP, we have to insert x. If the table is non-full, we insert x at position *free*, and if the table is full, we rehash and recompute the hash value of x. If x now hashes to an empty list, we put it into the first entry of the list, and otherwise, we put it at *free*.

⟨*implementation of ch_map*⟩+≡

```
GenPtr& ch_map::access(ch_map_item p, unsigned long x)
{
  STOP.k = x;
  ch_map_item q = p->succ;
  while (q->k != x) q = q->succ;
  if (q != &STOP) return q->i;
  // index x not present, insert it
```

```
    if (free == table_end)   // table full: rehash
    { rehash();
      p = HASH(x);
    }
    if (p->k == NULLKEY)
    { p->k = x;
      init_inf(p->i);  // initializes p->i to xdef
      return p->i;
    }
    q = free++;
    q->k = x;
    init_inf(q->i);    // initializes q->i to xdef
    q->succ = p->succ;
    p->succ = q;
    return q->i;
  }
```

We come to the protected member functions of *ch_map*. We start with some trivial stuff.

⟨*protected member functions of ch_map*⟩≡

```
  unsigned long key(ch_map_item it) const { return it->k; }
  GenPtr&       inf(ch_map_item it) const { return it->i; }
```

Constructors and Assignment: We start with the implementation class.

⟨*protected member functions of ch_map*⟩+≡

```
  ch_map(int n = 1);
  ch_map(const ch_map& D);
  ch_map& operator=(const ch_map& D);
```

The default constructor initializes a data structure of size min(512, $2^{\lceil \log n \rceil}$). The copy constructor initializes a table of the same size as D and then copies all elements from D to the new table. Elements from the first part of the table are moved if their key is different from `NULLKEY` and elements from the second part of the table are always moved. The assignment operator works in the same way but clears and destroys the old table first.

⟨*implementation of ch_map*⟩+≡

```
  ch_map::ch_map(int n) : NULLKEY(0), NONNULLKEY(1)
  {
    if (n < 512)
       init_table(512);
    else
     { int ts = 1;
       while (ts < n) ts <<= 1;
       init_table(ts);
      }
  }

  ch_map::ch_map(const ch_map& D) : NULLKEY(0), NONNULLKEY(1)
```

```
{
  init_table(D.table_size);
  for(ch_map_item p = D.table + 1; p < D.free; p++)
  { if (p->k != NULLKEY || p >= D.table + D.table_size)
    { insert(p->k,p->i);
      D.copy_inf(p->i);  // see chapter Implementation
     }
   }
}
ch_map& ch_map::operator=(const ch_map& D)
{
  clear_entries();
  delete[] table;
  init_table(D.table_size);
  for(ch_map_item p = D.table + 1; p < D.free; p++)
  { if (p->k != NULLKEY || p >= D.table + D.table_size)
    { insert(p->k,p->i);
      copy_inf(p->i);    // see chapter Implementation
     }
   }
  return *this;
}
```

The constructors of the abstract data type class simply call the appropriate constructor of the implementation class.

⟨*member functions of map*⟩≡

```
map()  { }
map(E x,int table_sz) : ch_map(table_sz), xdef(x) { }
map(E x) : xdef(x) { }
map<I,E>& operator=(const map<I,E>& M)
{ ch_map::operator=((ch_map&)M);
  xdef = M.xdef;
  return *this;
}
map(const map<I,E>& M): ch_map((ch_map&)M), xdef(M.xdef) { }
```

Destruction: We follow our canonical design for constructors, see Section 13.4.3. On the level of the implementation class, we define a function *clear_entries* that clears the information field of all used entries, a function *clear* that first clears the entries of the table and destroys the table and then reinitializes the table to its default size (*clear* is not used but we define it for the sake of uniformity), and the destructor that simply deletes *table*. Note that our canonical design ensures that *clear_entries* is called before any call of the destructor and hence only *table* must be destroyed by the destructor. Following standard practice (see [ES90, page278]) we declare the destructor virtual.

⟨*protected member functions of ch_map*⟩+≡

```
void clear_entries();
void clear();
virtual ~ch_map() { delete[] table; }
```

⟨*implementation of ch_map*⟩+≡

```
void ch_map::clear_entries()
{ for(ch_map_item p = table + 1; p < free; p++)
    if (p->k != NULLKEY || p >= table + table_size)
    clear_inf(p->i);  // see chapter Implementation
 }

void ch_map::clear()
{ clear_entries();
  delete[] table;
  init_table(512);
}
```

The destructor of the abstract data type class first calls *clear_entries* and then the destructor of the implementation class.

⟨*member functions of map*⟩+≡

```
~map() { clear_entries(); }
```

Access Operations: We have already defined the operation *access*(x) that searches for x and, if unsuccessful, inserts x into the table. *Lookup* only searches; it returns the item corresponding to a key x, if there is one, and *nil* otherwise.

⟨*protected member functions of ch_map*⟩+≡

```
GenPtr& access(unsigned long x);
ch_map_item lookup(unsigned long x) const;
```

⟨*implementation of ch_map*⟩+≡

```
ch_map_item ch_map::lookup(unsigned long x) const
{ ch_map_item p = HASH(x);
  ((unsigned long &)STOP.k) = x;  // cast away const
  while (p->k != x) p = p->succ;
  return (p == &STOP) ? nil : p;
}
```

The abstract data type class uses these functions in the obvious way.

⟨*member functions of map*⟩+≡

```
const E& operator[](const I& i) const
{ ch_map_item p = lookup(ID_Number(i));
  return (p) ? LEDA_CONST_ACCESS(E,ch_map::inf(p)) : xdef;
}
```

```
E& operator[](const I& i)
{ return LEDA_ACCESS(E,access(ID_Number(i))); }

bool defined(const I& i) const { return lookup(ID_Number(i)) != nil; }
```

In the above, *LEDA_ACCESS*(E, i) returns the value of i converted to type E, see Section 13.4.5, and *ID_number*(i) returns the ID-number of i.

⟨*member functions of map*⟩+≡

```
void clear()    { ch_map::clear(); }
void clear(E x) { ch_map::clear(); xdef = x; }
```

Iteration: The implementation of the iteration statements follows the general strategy described in Section 13.9. The implementation class provides two functions that return the first used item and the used item following a used item, respectively. Both functions are simple. The first item in the hash table is always unused and hence *first_item* returns *next_item*(*table*). We come to *next_item*(*it*). Let *it* be any item. If *it* is *nil*, we return *nil*. So assume otherwise. To find the next used item we advance *it* one or more times until we are either in the overflow area or have reached an item whose key is not equal to `NULLKEY`. If the resulting value of *it* is less than *free* we return it and otherwise we return *nil*.

⟨*protected member functions of ch_map*⟩+≡

```
ch_map_item first_item() const;
ch_map_item next_item(ch_map_item it) const;
```

⟨*implementation of ch_map*⟩+≡

```
ch_map_item ch_map::first_item() const
{ return next_item(table); }

ch_map_item ch_map::next_item(ch_map_item it) const
{ if ( it == nil ) return nil;
  do { it++; }
  while ( it < table + table_size && it->k == NULLKEY);
  return ( it < free ? it : nil);
}
```

The abstract data type class must provide the functions *first_item*, *next_item*, *inf*, *key*. All four functions reduce to the corresponding function in the implementation class.

⟨*member functions of map*⟩+≡

```
item first_item() const       { return ch_map::first_item(); }
item next_item(item it) const { return ch_map::next_item(it); }

E inf(item it) const
{ return LEDA_CONST_ACCESS(E,ch_map::inf(it)); }

I key(item it) const
{ return LEDA_CONST_ACCESS(I,(GenPtr)ch_map::key(it)); }
```

Exercises for 5.2

1 The unbalanced tree implementation of sparse arrays deteriorates to linear lists in the case of a sorted insertion order. In particular, if the keys 1, 2, ..., n are inserted in this order then each insertion appends the key to be inserted at the end of the list. Try to explain the row for *bin_trees* in the lower half of Table 5.2 in view of this sentence.

2 Use maps and the indices $1024i$ for $i = 0, 1, \ldots$.

3 Use h_arrays and the indices $1024i$ for $i = 0, 1, \ldots$. Define your own hash function.

4 Design a hash function for strings. The function should depend on all characters of a string.

5 Extend the implementation of h_arrays such that variables become persistent. (Hint: do not store the array variables directly in the hash table but access them indirectly through a pointer). What price do you pay in terms of access and insert time?

6 Provide a new implementation of d_arrays or h_arrays and perform the experiments of Table 5.2.

5.3 Dictionaries and Sets

Dictionaries and sets are essentially another interface to d_arrays and therefore we can keep this section short.

A *dictionary* is a collection of items (type *dic_item*) each holding a key of some linearly ordered type K and an information from some type I. Note that we now use I for the information type and no longer for the index type. We illustrate dictionaries by a program that reads a sequence of strings from standard input, counts the number of occurrences of each string, and prints all strings together with their multiplicities.

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

In the while-loop we first search for s in the dictionary. The lookup returns *nil* if s is not part of the dictionary and returns the unique item with key s otherwise. In the first case we insert the item $\langle s, 1\rangle$ into the dictionary. In the second case we increment the information associated with s.

Dictionaries are frequently used to realize sets. In this situation the information associated with an element in the dictionary is irrelevant, the only thing that counts is whether a key belongs to the dictionary or not. The data type *set* is appropriate in this situation. A set S of integers is declared by *set<int>* S. The number 5 is added by *S.insert*(5), the number 8 is tested for membership by *S.member*(8), and the number 3 is deleted by *S.delete*(3). The operation *S.choose*( ) returns some element of the set. Of course, *choose* requires the set to be non-empty.

We will discuss an extension of dictionaries in a later section: *Sorted sequences*. Sorted sequences extend dictionaries by more fully exploiting the linear order defined on the key type. They offer queries to find the next larger element in a sequence and also operations to merge and split sequences.

LEDA also contains extensions of dictionaries to geometric objects such as points and parallel line segments. We discuss a dictionary type for points in Section 10.6. For more dictionary types for geometric objects we refer the reader to the manual.

Exercises for 5.3

1 Implement dictionaries in terms of d_arrays. Are you encountering any difficulties?

2 Implement d_arrays in terms of dictionaries. Are you encountering any difficulties?

5.4 **Priority Queues**

Priority queues are an indispensable ingredient for many network and geometric algorithms. Examples are Dijkstra's algorithm for the single-source shortest-path problem (cf. Section 6.6), and the plane sweep algorithm for line segment intersection (cf. Section 10.7.2). We start with the basic properties of priority queues, and then discuss the many implementations of priority queues in LEDA. We give recommendations about which priority queue to choose in a particular situation.

5.4.1 ***Functionality***

A priority queue Q over a priority type P and an information type I is a collection of items (type *pq_item*), each containing a priority from type P and an information from type I. The type P must be linearly ordered. A priority queue organizes its items such that an item with minimum priority can be accessed efficiently.

```
p_queue<P,I> Q;
```

defines a priority queue Q with priority type P and information type I and initializes Q to the empty queue. A new item $\langle p, i\rangle$ is added by

```
Q.insert(p,i);
```

and

```
pq_item it = Q.find_min();
```

returns an item of minimal priority and assigns it to *it* (*find_min* returns *nil* if Q is empty). Frequently, we do not only want to access an item with minimal information but also want to delete it.

```
P p = Q.del_min();
```

deletes an item with minimum priority from Q and assigns its priority to p (Q must be non-empty, of course). An arbitrary item *it* can be deleted by

```
Q.del_item(it);
```

The fields of an item are accessed by *Q.prio*(*it*) and *Q.inf*(*it*), respectively. The operation *Q.insert*(p, i) adds a new item $\langle p, i\rangle$ and returns the item; so we may store it for later use:

```
pq_item it = Q.insert(p,i);
```

There are two ways to change the content of an item. The information can be changed arbitrarily:

```
Q.change_inf(it,i1);
```

makes *i1* the new information of item *it*. The priority of an item can only be decreased:

```
Q.decrease_p(it,p1);
```

makes *p1* the new priority of item *it*. The operation raises an error if *p1* is larger than the current priority of *it*. There is no way to increase the priority of an item[4]. Finally, there are the operations

```
Q.empty();
Q.size();
Q.clear();
```

that test for emptiness, return the number of items, and clear a queue, respectively.

Let us see priority queues at work. We read a sequence of doubles from standard input and store them in a priority queue. We then repeatedly extract the minimum element from the queue until the queue is empty. The net effect is to sort the input sequence into increasing order.

```
p_queue<double,int> Q; //the information type is irrelevant
double x;
while (cin >> x) Q.insert(x,0);
while (! Q.empty()) cout << Q.del_min << "\n";
```

A more sophisticated use of priority queues is *discrete event simulation*. We have a set of events associated with points in time. An event associated with time t is to be executed at time t. The execution of an event may create new events that are to be executed at later moments of time. Priority queues support discrete event simulation in a very natural way; one only has to store all still to be executed events together with their scheduled time in a priority queue (with time playing the role of the priority) and to always extract and execute the event with the minimal scheduled time.

[4] The fact that priorities can be decreased but not increased is dictated by the implementations. There are implementations that support very efficient decrease of priorities but there are no implementations that support efficient decrease and increase.

			Running times			
Name	Prio	Args	*insert*	*delete_min*	*decrease_p*	*create, destruct*
f_heap	general	—	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
p_heap	general	—	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$
k_heap	general	$N, k = 2$	$O(\log_k n)$	$O(k \log_k n)$	$O(\log_k n)$	$O(N)$
bin_heap	general	—	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)$
list_pq	general	—	$O(1)$	$O(n)$	$O(1)$	
b_heap	int, $[l .. h]$	l, h	$O(1)$	$O(h - l)$	$O(h - l)$	$O(h - l)$
r_heap	int	C	$O(\log C)$	$O(\log C)$	$O(1)$	$O(\log C)$
m_heap	int	C	$O(1)$	$O(min - p\_min)$	$O(1)$	$O(C)$

Table 5.4 Properties of different priority queue implementations: the second column indicates whether the priorities can come from an arbitrary linearly ordered type (general) or must be integers, the third column indicates the arguments of the constructor, and the remaining columns indicate the running times of the various priority queue operations. *B_heaps* can only handle integer priorities from a fixed range $[l .. h]$ and *r_heaps* and *m_heapa* maintain a variable *p_min* and priorities must be integers in the range $[p\_min .. p\_min + C - 1]$. *B_heaps* also support a *delete_max* operation. More detailed explanations are given in the text.

5.4.2 *Performance Guarantees and Implementation Parameters*

LEDA provides many implementations of priority queues. The implementations include Fibonacci heaps [FT87], pairing heaps [SV87], k-ary heaps and binary heaps [Meh84a, III.5.3.1], lists[5], buckets[6], redistributive heaps [AMOT90], and monotone heaps [Meh84b, IV.7.2]. Fibonacci heaps are the default implementation and other implementations can be selected using the implementation parameter mechanism. The implementation parameters are *f_heap*, *p_heap*, *k_heap*, *bin_heap*, *list_pq*, *b_heap*, *r_heap*, and *m_heap*, respectively. Fibonacci heaps support *insert*, *del_item* and *del_min* in time $O(\log n)$, *find_min*, *decrease_p*, *change_inf*, *inf*, *size*, and *empty* in time $O(1)$, and *clear* in time $O(n)$, where n denotes the current size of the queue. The time bounds are amortized. The space requirement of Fibonacci heaps is linear in the size of the queue. We give their implementation in Section 13.10.

Table 5.4 surveys the properties of the other implementations. Some implementations allow any linearly ordered type for the priority type (this is indicated by the word general) and some work only for a prespecified range of integer priorities. The constructors take zero or more arguments. For all priority queues that work only for a subset of the integers the set of admissible priorities is defined by constructor arguments. k-ary heaps require that

[5] In the list implementation the items of the queue are stored as an unordered list. This makes *delete_min* and *find_min* linear time processes (linear search through the entire list) and trivializes all other operations.

[6] In the bucket implementation we have an array of linear lists; the list with index i contains all items whose priority is equal to i. This scheme requires the priorities to be integers from a prespecified range.

an upper bound N for the maximal size of the queue and the parameter k is specified in the constructor; the default value of k is 2.

Redistributive heaps and monotone heaps do only support monotone use of the priority queue. The use of a priority queue is *monotone* if the priority argument in any *insert* or *decrease_p* operation is at least as large as the priority returned by the last *delete_min* or *find_min* operation. Dijkstra's shortest-path algorithm uses its priority queue in a monotone way. *R_heaps* and *m_heaps* maintain a variable *p_min* that is initialized to the priority of the first insertion and that is updated to the priority returned by any *delete_min* or *find_min* operation. Only priorities in the range $[p\_min .. p\_min + C - 1]$ can be inserted into the queue, where C is specified in the constructor. In *m_heaps* the cost of a *delete_min* is the difference between the result of this *delete_min* operation and the preceding one[7].

The *b_heap* implementation allows one to ask for the maximum priority and not only for the minimum priority. This is sometimes called a *double-sided* priority queue. For integer priorities there are realizations known that have an even better performance than *r_heaps*. The papers [AMOT90] and [CGS97] describe realizations where *insert* and *delete_min* take time $O(\sqrt{\log C})$ and $O((\log C)^{1/3+\varepsilon})$ for arbitrary $\varepsilon > 0$, respectively.

In order to select an implementation different from the default implementation, a declaration

```
_p_queue<K,int,prio_impl> Q(parameters);
```

has to be used, where *parameters* denotes the list of parameters required by the implementation, e.g.,

```
_p_queue<int,int,r_heap> Q(100000);
```

selects the *r_heap* implementation and sets C to 100000.

A priority queue with a particular implementation is, of course, still a priority queue and can hence be used wherever a priority queue can be used. We give an example. We write a procedure *dijkstra* that takes a graph G, a node s, an *edge_array<int> cost* of edge weights, and a *p_queue<int, node> PQ*, and solves the single-source shortest-path problem for the specified source node. The distances are returned in a *node_array<int> dist*. The edge costs must be non-negative.

⟨*dijkstra*⟩≡

```
void dijkstra(graph& G, node s, const edge_array<int>& cost,
              node_array<int>& dist, p_queue<int,node>& PQ)
{ node_array<pq_item> I(G);
  node v;

  forall_nodes(v,G)
    dist[v] = MAXINT;

  dist[s] = 0;
```

[7] The *m_heap* implementation uses an array of size C of linear lists and a variable *p_min* which is initialized to the priority of the first insertion. An item with priority i is stored in the list with index $i \bmod C$. Since priorities are allowed only from the range $[p\_min .. p\_min + C - 1]$ this implies that each list contains only items with the same priority. A *delete_min* or *find_min* operation advances *p_min* cyclically until a non-empty list is found.

```
    I[s] = PQ.insert(0,s);

    while (! PQ.empty())
    { pq_item it = PQ.find_min();
      node u = PQ.inf(it);
      int du = dist[u];
      edge e;
      forall_adj_edges(e,u)
      { v = G.target(e);
        int c = du + cost[e];
        if (c < dist[v])
        { if (dist[v] == MAXINT)
            I[v] = PQ.insert(c,v);
          else
            PQ.decrease_p(I[v],c);
          dist[v] = c;
        }
      }
      PQ.del_item(it);
    }
  }
```

We give some explanations; the correctness of the algorithm is shown in Section 6.6. Dijkstra's algorithm keeps a tentative distance value for each node and a set of active nodes. For a node v its tentative distance value is stored in *dist*$[v]$ and the set of pairs $(\mathit{dist}[v], v)$, where v is an active node, is stored in the priority queue *PQ*. Each active node v knows the *pq_item* containing the pair $(\mathit{dist}[v], v)$; it is stored in entry $I[v]$ of the *node_array<pq_item>* I. Initially, only the source node s is active and its distance from s is zero. In each iteration of the loop the pair with minimum distance value is deleted from *PQ*, say the pair (du, u) and all edges e leaving u are scanned. An edge $e = (u, v)$ allows us to reach node v through a path of cost $c = du + cost[e]$. If c is smaller than the cost of the best path known to v so far, this change is recorded in *dist*$[v]$ and the priority queue is informed about the change. More precisely, if no path to v was known so far, i.e., *dist*$[v]$ is still equal to *MAXINT*, a new pair (c, v) is inserted into the priority queue and the item returned is stored in $I[v]$ and if some path was already known then the priority of node v in the queue is updated. Note that in the latter case $I[v]$ contains the item for v in *PQ*.

We turn to the analysis of the running time. It can be shown (see Section 7.5.3) that each node is inserted and deleted from the priority queue at most once; of course, nodes that cannot be reached from s are never inserted into the queue. The algorithm therefore performs at most n *insert*, *empty*, *find_min*, and *delete_min* operations and at most m *decrease_p* operations. Here n and m denote the number of nodes and edges of G, respectively. The time spent outside the calls to the priority queue is $O(n + m)$ since array accesses take constant time and since the time to scan through all edges leaving a node u is proportional to the outdegree of the node. It is fair also to include the time for the construction and the destruction of the queue (although this happens outside procedure *dijkstra*). The total running time is therefore bounded by $O(n+m+n\cdot(T_{insert}+T_{empty}+T_{find\_min}+T_{delete\_min})+m\cdot T_{decrease\_p}+$

	Worst case running time	Expected running time
f_heap	$O(m + n \log n)$	$O(m + n \log n)$
p_heap	$O(m + n \log n)$	$O(m + n \log n)$
k_heap	$O(m \log_k n + nk \log_k n)$	$O(m + n(\log(2m/n) + k) \log_k n)$
bin_heap	$O(m \log n + n \log n)$	$O(m + n \log(m/n) \log n)$
list_pq	$O(m + n^2)$	$O(m + n^2)$
b_heap	$O((m + n)nM)$	$O((m + n)nM)$
r_heap	$O(m + n \log M)$	$O(m + n \log M)$
m_heap	$O(m + max\_dist + M)$	$O(m + max\_dist + M)$

Table 5.5 Asymptotic running times of Dijkstra's algorithm with different priority queue implementations. In order to keep the formulae simple we assumed $n \le m$. For the last three rows the edge weights must be integral and from the range $[0 \mathinner{..} M - 1]$. The rows for *b_heaps* and *m_heaps* require some explanation. Note that the maximal priority ever removed from the queue is bounded by $(n - 1)M$ since a shortest path consists of at most $n - 1$ edges. Thus one can use *b_heaps* with $l = 0$ and $h = nM$. For *r_heaps* and *m_heaps* we observe that the fact that edge costs are bounded by M guarantees that all priorities in the queue come from the range $[p\_min \mathinner{..} p\_min + M - 1]$ and hence we can use these implementations with $C = M$. In *m_heaps* the cost of a *delete_min* is $O(min - p\_min)$, where *min* and *p_min* are the results of the current and the previous *delete_min* operations. The sum of the differences $min - p\_min$ over all *delete_min* operations is bounded by the maximal distance of any node from the source.

$T_{create} + T_{destruct}$) where T_X is the time bound for operation X. Note that the expression above is an upper bound on the running time. The actual number of *decrease_p* operations may be smaller than m. In fact, it can be shown that for random graphs and random edge weights the expected number of *decrease_p* operations is $O(\min(m, n \log(2m/n)))$, see [Nos85]. We can now use Table 5.4 to estimate the asymptotic running time of *dijkstra* with different implementations of the priority queue.

The result is shown in Table 5.5. The first five lines contain the implementations that work for arbitrary non-negative real edge weights. The best worst case and average case time is $O(m + n \log n)$; they are achieved by *f_heaps* and *p_heaps*. For dense graphs with $m = n^{1+\varepsilon}$ for some positive ε, *k_heaps* with $k = n^{1/\varepsilon}$ achieve a worst case time[8] of $O((1/\varepsilon)m)$ which is competitive with the above for ε bounded away from zero. The expected running time[9] of *bin_heaps* is competitive for $m = \Omega(n \log(m/n) \log n)$. The last three lines of the table contain implementations that work only for integral edge weights. In these lines we use M to denote 1 plus the maximal weight of any edge. The best worst case and

[8] The worst case running time of *k_heaps* is $O(nk \log_k n + m \log_k n)$. For $k = n^{1/\varepsilon}$ we have $\log_k n = \log n / \log k = 1/\varepsilon$ and $nk = n^{1+1/\varepsilon} = m$.

[9] In *bin_heaps* the cost of a *decrease_key* is $O(\log n)$. The expected number of *decrease_key* operations is $n \log(2m/n)$. Thus, if $m \ge n \log(2m/n) \log n$ the running time is $O(m)$.

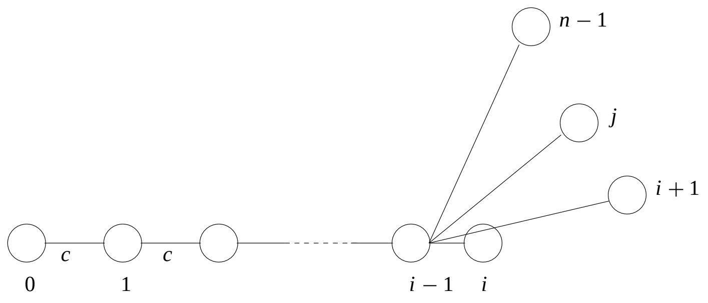


Figure 5.6 A worst case graph for Dijkstra's algorithm. All edges $(i, i+1)$ have cost c and an edge (i, j) with $i+1 < j$ has cost $c_{i,j}$. The $c_{i,j}$ are chosen such that the shortest path tree with root 0 is the path $0, 1, \ldots, n-1$ and such that the shortest path tree that is known after removing node $i-1$ from the queue is as shown. Among the edges out of node $i-1$ the edge $(i-1, i)$ is the shortest, the edge $(i-1, n-1)$ is the second shortest, and the edge $(i-1, i+1)$ is the longest.

average case time is $O(m + n \log M)$ achieved by *r_heaps*. For $M = O(1)$ the *m_heap* implementation is competitive. The heap implementations described in [AMOT90] and [CGS97] yield a running time of $O(m + n\sqrt{\log M})$ and $O(m + n(\log M)^{1/3+\varepsilon})$ for arbitrary $\varepsilon > 0$, respectively.

How do the different implementations compare experimentally? We will perform experiments with random graphs and with worst case graphs. Before reporting running times we construct a graph with n nodes and m edges that forces Dijkstra's algorithm into $m - n + 1$ *decrease_p* operations; observe that this number is the maximal possible since the distance of s is never decreased and since for any node v different from s the first edge into v that is scanned leads to an *insert* but not to a *decrease_p* operation. The construction works for all m and n with $m \leq n(n-1)/2$. Let c be any non-negative integer. The graph consists of:

- the nodes $0, 1, \ldots, n-1$,
- the $n-1$ edges $(i, i+1)$, $0 \leq i < n-1$, each having cost c, and
- the first $m' = m - (n-1)$ edges in the sequence $(0, 2)$, $(0, 3)$, …, $(0, n-1)$, $(1, 3)$, $(1, 4)$, …, $(1, n-1)$, $(2, 4)$, … . The edge (i, j) in this sequence is given cost $c_{i,j}$ to be defined below.

We will define the $c_{i,j}$ such that the shortest path tree with respect to node 0 is the path $[0, 1, \ldots, n-1]$, such that the nodes are removed from the queue in the order of their node number, and such that the shortest path tree that is known after removing node i from the queue is as shown in Figure 5.6. The shortest path from 0 to i has cost ic and the path $[0, 1, \ldots, i-1, i, j]$ has cost $ic + c_{i,j}$, see Figure 5.6.

When node 0 is removed from the queue all other nodes are put into the queue. The priority of node 1 is equal to c and the priority of node j, $j > 1$, is equal to $c_{0,j}$. Generally, just prior to the removal of node i the queue contains nodes i to $n-1$: Node i has priority

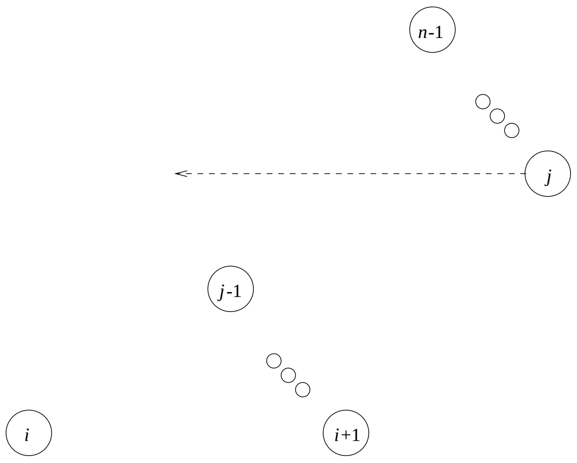


Figure 5.7 The effect of scanning the edges out of node i. When the scanning starts the nodes $i+1$ to $n-1$ are in the queue and we have $dist[n-1] < \ldots < dist[i+1]$. Just prior to the scanning of edge $(i,\ j)$ we have the situation shown; in this figure distance values are indicated as x-coordinates. Scanning $(i,\ j)$ will make $dist[j]$ the smallest priority in the queue. The edges out of i are scanned in the order $(i, i+2), \ldots, (i, n-1), (i, i+1)$.

ic and node j, $j > i$, has priority $(i-1)c + c_{i-1,j}$. We now remove node i from the queue and scan through the edges out of i. We postulate that we look at the edges in the order $(i, i+2), (i, i+3)$, $(i, n-1), (i, i+1)$.

Under what conditions will each edge $(i,\ j)$ cause a *decrease_p* operation and, moreover, will the new priority given to node j by this edge be the smallest priority in the queue? This will be the case if the $c_{i,j}$ are chosen such that

$$\begin{array}{rrcll} & ic + c_{i,i+2} & < & (i-1)c + c_{i-1,n-1}, & \\ & c_{i,j} & < & c_{i,j-1} & \text{for all } j, i+2 < j \le n-1, \\ \text{and} & c = c_{i,i+1} & < & c_{i,n-1}. & \end{array}$$

Note that the first inequality implies that the edge $(i, i+2)$ causes a *decrease_p* operation, that the second inequality implies that the edge $(i,\ j)$ causes a *decrease_p* operation for all $j, i+2 < j \le n-1$, and that the third inequality implies that the edge $(i, i+1)$ causes a *decrease_p* operation. Also note that this choice of edge costs implies that before the scan of the edges out of i we have $dist[n-1] < \ldots < dist[i+1]$ and that consideration of edge $(i,\ j)$ will make $dist[j]$ the smallest value in the queue, i.e., before $(i,\ j)$ is considered we have $dist[j-1] < \ldots < dist[i+2] < dist[n-1] < \ldots < dist[j] < dist[i+1]$ and after $(i,\ j)$ is considered we have $dist[j] < dist[j-1] < \ldots < dist[i+2] < dist[n-1] < \ldots < dist[j+1] < dist[i+1]$, see Figure 5.7. In this way each edge scan causes a major change in the priority queue.

How can we choose $c_{i,j}$'s satisfying these inequalities? We suggest the following strategy. We first determine the m' additional edges to be used and then assign the edge costs to the additional edges in reverse order. Note that the last edge can be given cost $c+1$, and that $c_{i,j}$ can be put to $c_{i,j+1}+1$ if $j < n-1$ and can be put to $c_{i+1,i+3}+c+1$ if $j = n-1$. The following program realizes this strategy and returns the largest cost assigned to any edge.

⟨*worst case generator*⟩≡

```
int DIJKSTRA_GEN(GRAPH<int,int>& G, int n, int m, int c = 0)
{ G.clear();
  array<node> V(n);
  int i;
  for (i = 0; i < n; i++) V[i] = G.new_node(i);
  stack<edge> S;
  int m1 = m - (n - 1);
  i = 0;
  int j = i + 2;
  while (m1 > 0)
  { if (j < n )
    { S.push(G.new_edge(V[i],V[j])); m1--; j++; }
    else
    { i++; j = i + 2;
      if (j == n)
      error_handler(1,"DIJKSTRA_GEN: m can be at most n*(n-1)/2");
    }
  }
  edge e = S.pop();
  int last_c = G[e] = c + 1;
  while (!S.empty())
  { e = S.pop();
    int j = G[G.target(e)];
    if (j == n-1)
      last_c = G[e] = last_c + c + 1;
    else
      last_c = G[e] = last_c + 1;
  }
 for (i = 0; i < n-1; i++) G.new_edge(V[i], V[i+1], c);
 return last_c;
}
```

A further remark about this program is required. The *new_edge* operation appends the new edge to the adjacency list of the source node and hence the adjacency list of any node i will be ordered $(i, i+2), \ldots, (i, n-1), (i, i+1)$, as desired.

We come to the experimental comparison of our different priority queue implementations. We refer the reader to [CGS97] for more experimental results. It is easy to time *dijkstra* with a particular implementation, e.g.,

⟨*generate a section of table: Dijkstra timings*⟩≡

```
{ p_queue<int,node> fheap;                       K = "fheap";
  dijkstra(G,s,cost,dist,fheap);
}
```
⟨*report time for heap of kind K*⟩

```
{ _p_queue<int,node,p_heap> pheap;              K = "pheap";
  dijkstra(G,s,cost,dist,pheap);
}
⟨report time for heap of kind K⟩
{ int d = m/n;                    // degree for k_heap
  if ( d < 2 ) d = 2;
  _p_queue<int,node,k_heap> kheap(n,d);         K = "kpeap";
  dijkstra(G,s,cost,dist,kheap);
}
⟨report time for heap of kind K⟩
{ _p_queue<int,node,bin_heap> binheap(n);       K = "binheap";
  dijkstra(G,s,cost,dist,binheap);
}
⟨report time for heap of kind K⟩
if (i != 2)  // listheaps are too slow for section 2 of table
{
  { _p_queue<int,node,list_pq> listheap;        K = "listheap";
    dijkstra(G,s,cost,dist,listheap);
  }
  ⟨report time for heap of kind K⟩
}
else cout << "& - " ; cout.flush();
{ _p_queue<int,node,r_heap> rheap(C);           K = "rheap";
  dijkstra(G,s,cost,dist,rheap);
}
⟨report time for heap of kind K⟩
{ _p_queue<int,node,m_heap> mheap(C);           K = "mheap";
  dijkstra(G,s,cost,dist,mheap);
}
⟨report time for heap of kind K⟩
```

generates one section of Table 5.6. We have enclosed the experiment in a block such that the time for the destruction of the queue is also measured. Table 5.6 shows the results of our experiments. You can perform your own experiments with the priority queue demo.

We see that *p_heaps* are consistently better than *f_heaps* and that *r_heaps* are in many situations even better. The exception is when the ratio m/n is very small, the maximal edge weight is large, and we use the worst case graph. In the latter situation, the $n \log M$ term in the running time dominates. For random graphs *bin_heaps* are competitive. *K_heaps* are worse than *bin_heaps* on random graphs (because our choice of k is bad for random graphs) and are competitive for worst case graphs. *List_pq* cannot be run for large values of n because of the n^2-term in the running time. *M_heaps* do surprisingly well even for large edge weights. This is due to the fact that the M-term in the running time does not really harm *m_heaps* in our experiments because of the large value of m.

5.4.3 *Choosing an Implementation*

LEDA gives you the choice between many implementations of priority queues. Which is best in a particular situation?

Instance	*f_heap*	*p_heap*	*k_heap*	*bin_heap*	*list_pq*	*r_heap*	*m_heap*
s,r,S	0.36	0.34	0.35	0.34	0.51	0.33	0.35
s,r,L	0.38	0.36	0.37	0.34	0.54	0.35	0.54
s,w,S	1.86	1.09	3.77	1.38	1	0.76	2.68
s,w,L	1.87	1.1	3.68	1.34	1	0.77	8.49
m,r,S	1.24	0.94	1.14	0.94	31.6	0.83	0.94
m,r,L	1.39	1.13	1.28	1.02	23	0.93	1.22
m,w,S	2.36	1.44	4.94	1.77	22.7	0.99	2.78
m,w,L	2.36	1.45	4.84	1.74	21.7	1.03	3.29
l,r,S	4.96	3.19	5.2	3.36	-	2.52	2.52
l,r,L	6.61	4.81	6.4	4.49	-	3.76	3.38
l,w,S	3.32	2.56	9.17	3.79	-	1.63	3.11
l,w,L	2.91	1.92	7.65	3.22	-	2.57	2.55

Table 5.6 Running times of Dijkstra's algorithm with different priority queue implementations. We used graphs with $m = 500000$ edges and either $n = 2000$, $n = 20000$, or $n = 200000$ nodes. The three cases are distinguished by the labels s, m, and l, respectively. For each combination of n and m we generated four graphs. Two random graphs (r) with random edge weights in $[0 .. M - 1]$, where $M = 100$ or $M = 100000$, and two worst case graphs (w) with $c = 0$ or $c = 10000$. The two cases for M and c are distinguished by the labels S and L, respectively. So s,r,L indicates that we used 2000 nodes, a random graph, and M equal to 100000. In the *k_heap* implementation we set $k = \max(2, m/n)$, as this minimizes the worst case running time.

Tables 5.4 and 5.6 suggest to use either *p_heaps*, *bin_heaps*, or *r_heaps*. *R_heaps* are the data structure of choice if the use of the queue is monotone and the parameter C is such that $\log C$ is not much larger that $\log n$. If the keys are not integers or $\log C$ is much larger than $\log n$, one should use either *bin_heaps* or *p_heaps*. The former are to be preferred when the number of *decrease_p* operations is not too large and the latter is to be preferred otherwise.

If you are not happy with any of the implementations provided in LEDA, you may provide your own. Section 13.6 explains how this is done.

Exercises for 5.4

1 Consider a graph with two nodes v and w and one edge (v, w) of cost M. What is the running time of the different versions of *dijkstra* on this graph as a function of M. Verify your result experimentally.

2 Implement hot queues as described in [CGS97].

3 Time Dijkstra's algorithm with *k_heaps* for different values of k. Do so for random graphs and also for worst case graphs. Which value of k works best?

4 Use priority queues to sort a set of n random integers or random doubles. Compare the different queue implementations. In the case of *k_heaps* try different values of k. Compare your findings for *k_heaps* with the experiments in [LL97].

5.5 Partition

We discuss the data type partition: its functionality, its implementation, and a non-trivial application in the realm of program checking.

5.5.1 *Functionality*

A partition P consists of a finite set of items of type *partition_item* and a decomposition of this set into disjoint sets called blocks. Figure 5.8 visualizes a partition. The declaration

```
partition P;
```

declares a partition P and initializes it to the empty partition, i.e., there are no items in P yet.

```
P.make_block();
```

adds a new item to P, makes this item a block by itself, and returns the item; see Figure 5.9. We may store the returned item for later use.

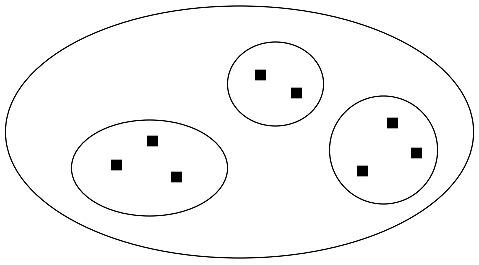


Figure 5.8 A partition P of eight items into three blocks. Partition items are indicated as solid squares and blocks are indicated as ellipses enclosing the items constituting the block.

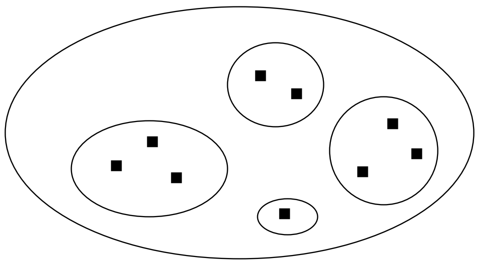


Figure 5.9 The partition of Figure 5.8 after a *make_block* operation.

```
partition_item it = P.make_block();
```

There are several ways to query a partition and to modify it.

```
P.same_block(it1,it2);
```

returns *true* if the partition items *it1* and *it2* belong to the same block of P and *false* otherwise.

```
P.union_blocks(it1,it2);
```

combines the blocks containing items *it1* and *it2*, respectively.

For each block one of its elements is designated as the "canonical" item of the block.

```
P.find(it);
```

returns the "canonical" element of the block containing *it*. Note that *it* and P.*find*(*it*) belong to the same block of P and that if *it1* and *it2* belong to the same block then P.*find*(*it1*) and P.*find*(*it2*) return the same item. Thus

```
P.same_block(it1,it2) == (P.find(it1) == P.find(it2))
```

is a fancy way to write the constant *true*.

If L is a list of partition items then

```
P.split(L);
```

splits all blocks consisting of items in L into singelton blocks. L must be a union of blocks of P.

We give a small example program to see partitions at work. We maintain a partition P of n items. We start with the partition into singleton blocks and then repeat the following step until the largest block has reached size $9n/10$. We choose two items at random and merge the blocks containing them (this has no effect if the two items belong already to the same block). During the experiment we keep track of the block sizes. Whenever the size of the maximal block reaches $in/100$ for some i, $i \geq 1$, we report the number of steps and the size of the two largest components.

In order to facilitate the selection of two random items we store all items of the partition in an *array*<*partition_item*> *Item*. This reduces the selection of a random partition item to the selection of a random integer.

We keep track of the block sizes in a *sortseq*<*int*, *int*> *freq*; see Section 5.6. We store for each block size s the number k of blocks having size s in *freq*. Initially, all blocks have size 1 and there are n blocks of size 1.

⟨*giant_component_demo*⟩≡
```
main(){
  ⟨giant component demo: read n⟩
  partition P;
  array<partition_item> Item(n);
  sortseq<int,int> freq;
```

```
    for (int i = 0; i < n; i++) Item[i] = P.make_block();
    int iteration = 0; int step = 1; int max_size = 1;
    freq.insert(1,n);
    while ( max_size < n/2 )
    { int v = rand_int(0,n-1);
      int w = rand_int(0,n-1);
      iteration++;
      if ( P.same_block(Item[v],Item[w]) ) continue;
      seq_item it = freq.lookup(P.size(Item[v]));
      freq[it]--;
      if ( freq[it] == 0 ) freq.del_item(it);
      it = freq.lookup(P.size(Item[w]));
      freq[it]--;
      if ( freq[it] == 0 ) freq.del_item(it);
      P.union_blocks(Item[v],Item[w]);
      int size = P.size(Item[v]);
      it = freq.lookup(size);
      if (it) freq[it]++; else freq.insert(size,1);
      it = freq.max();
      max_size = freq.key(it);
      int second_size = freq.key(freq.pred(it));
      while (max_size >= step*n/100 )
      { ⟨giant component demo: report step⟩
        step++;
      }
    }
  }
```

Part of the output of a sample run of the program above with $n = 10^6$ is as follows:

The maximal block size jumped above 0.16n after 542386 iterations. The maximal size of a block is 160055 and the second largest size of a block is 715.

The maximal block size jumped above 0.17n after 545700 iterations. The maximal size of a block is 170030 and the second largest size of a block is 722.

The maximal block size jumped above 0.18n after 548573 iterations. The maximal size of a block is 180081 and the second largest size of a block is 330.

The maximal block size jumped above 0.19n after 552784 iterations. The maximal size of a block is 190008 and the second largest size of a block is 336.

The maximal block size jumped above 0.20n after 556436 iterations. The maximal size of a block is 200003 and the second largest size of a block is 380.

Observe that it took more than 500 000 iterations until the largest block reached size $0.16n$, and only 4 000 additional iterations until the largest block reached size $0.17n$, ... Moreover, the size of the largest block is much larger than the size of the second largest block. In fact, the second largest block is tiny compared to the largest block. This phenomenon is called *the evolution of the giant component* in the literature on random graphs, see [ASE92] for an analytical treatment of the phenomenon. You may perform your own experiments with the giant component demo. Qualitatively, the phenomenon of the giant component is easy to explain. At any time during the execution of the algorithm the probability to merge two blocks of size k_1 and k_2, respectively, is proportional to k_1k_2 since k_1k_2 is the number of pairs that can be formed by choosing one item in each block. Thus the two blocks most likely to be merged are the largest and the second largest block. Merging them makes the largest block larger and the second largest block smaller (as the third largest block becomes the second largest). Although we knew about the phenomenon before we wrote the demo we were surprised to see how dominating the largest block is.

There are two variants of the partition data type: *Partition* and *node_partition*. A node partition is a partition of the nodes of a particular graph. It is very useful for graph algorithms and we will discuss it in Section 6.6. A *Partition*$<I>$ is a partition where one can associate an information of type I with every item of the partition. The operation

```
partition_item it = P.make_block(i);
```

creates an item with associated information i and makes the item a new block of P, the operation

```
P.inf(it);
```

returns the information of item *it* and

```
P.change_inf(it, i1);
```

changes the information of *it* to *i1*. The type *Partition* is appropriate whenever one wants to associate information with either the items or the blocks of a partition. In the latter case one simply associates the information with the canonical item of the block. We give one such application in Section 5.5.3.

5.5.2 ***The Implementation***

Partitions are implemented by the so-called *union-find data structure with weighted union and path compression*. This data structure is a collection of *partition_nodes* which are arranged into a set of trees, see Figure 5.10 for an example. Each block of the partition corresponds to a tree. A *partition_item* is a pointer to a *partition_node*. Each partition node contains a pointer to its parent and each root node knows the size of the tree rooted at it. This is called the *size* of the root. A partition node also contains a field *next* that is used to link all nodes of a partition into a singly linked list. The definition of class *partition_node* is as follows:

⟨*partition_node*⟩≡

```
class partition_node {
  friend class partition;
  partition_node* parent;
  partition_node* next;
  int size;
```

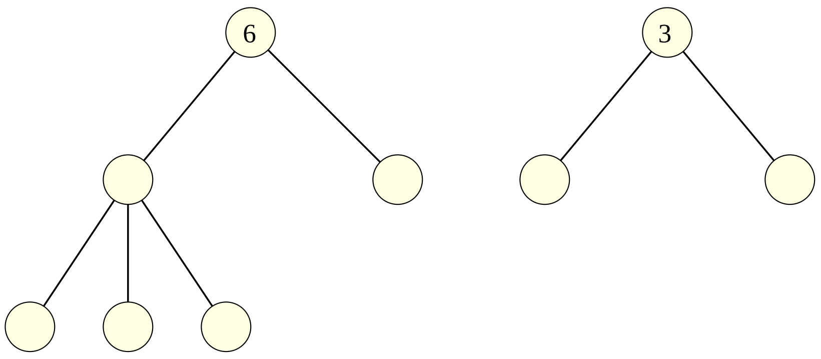


Figure 5.10 The representation of a partition with two blocks of six and three items, respectively. All edges are directed upwards. The size of root nodes is indicated inside the node. All nodes are also linked into a singly linked list. This list is not shown.

```
public:
  partition_node(partition_node* n)  { parent = 0; size = 1; next = n; }
  LEDA_MEMORY(partition_node)
};
typedef partition_node* partition_item;
```

The constructor constructs a node with no parent and size one. We will see its use below, where the use of the field *next* and the argument *n* will also become clear.

We come to class partition. It has only one data member *used_items* that points to the first item in the linear list of all items comprising the partition.

⟨*partition.h*⟩≡
```
#include <LEDA/basic.h>
⟨partition_node⟩
class partition {
  partition_item used_items;   // list of used partition items
public:
  ⟨member functions of partition⟩
};
```

In order to create an empty partition we set *used_items* to *nil* and in order to destroy a partition we go through the list of items comprising the partition and delete all of them.

⟨*member functions of partition*⟩≡
```
partition() { used_items = nil; }
~partition()
{ while (used_items)
  { partition_item p = used_items;
    used_items = used_items->next;
```

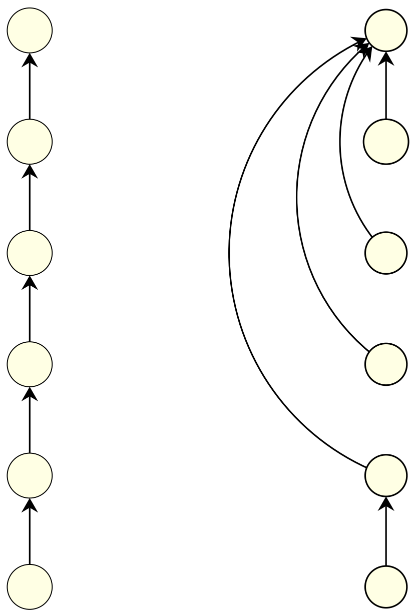


Figure 5.11 Path compression: All edges are directed upwards and the path compression was initiated by an operation *find*(p). After the path compression all ancestors of p including p point directly to the root of the tree containing p.

```
    delete p;
  }
}
```

In order to make a new block we allocate a new *partition_node*, append it to the front of the list of items comprising the partition, and return a pointer to the new node. Observe that we defined the constructor of class *partition_node* such that this works nicely.

⟨*member functions of partition*⟩+≡

```
partition_item make_block()
{ used_items = new partition_node(used_items);
  return used_items;
}
```

We come to function *find*(*partition_item* p). It returns the root of the tree representing the block containing p. This root is easy to find, we only have to follow the chain of parent pointers starting at p. We do slightly more. Once we have determined the *root* of the tree containing p we traverse the path starting at p a second time and change the parent pointer of all nodes on the path to *root*, see Figure 5.11. This is called path compression; it makes the current find operation a bit more expensive but saves all later find operations from traversing the path from p to *root*.

⟨*member functions of partition*⟩+≡

```
partition_item find(partition_item p)
{ // find with path compression
  partition_item x = p->parent;
  if (x == 0) return p;
  partition_item root = p;
  while (root->parent) root = root->parent;
  while (x != root)  // x is equal to p->parent
  { p->parent = root;
    p = x;
    x = p->parent;
  }
  return root;
}
```

The function *same_block*(p, q) returns *find*$(p) ==$ *find*(q).

⟨*member functions of partition*⟩+≡

```
bool  same_block(partition_item p, partition_item q)
{ return find(p) == find(q); }
```

In order to unite the blocks containing items p and q we first determine the roots of the trees containing these items. If the roots are the same then there is nothing to do. If the roots are different, we make one of them the child of the other. We follow the so-called weighted union rule and make the lighter root the child of the heavier root. This rule tends to keep trees shallow[10].

⟨*member functions of partition*⟩+≡

```
void  union_blocks(partition_item p, partition_item q)
{ // weighted union
  p = find(p);
  q = find(q);
  if ( p == q ) return;
  if (p->size > q->size)
       { q->parent = p;
         p->size += q->size; }
  else { p->parent = q;
         q->size += p->size; }
}
```

Despite its simplicity the implementation of *partition* given above is highly effective. A sequence of n *make_block* and m other operations takes time $O((m+n)\alpha(m+n,n))$

[10] We show that the depth of all trees is logarithmically bounded in their size. For any non-negative integer d let s_d be the minimal size of a root whose tree has depth d. Then $s_0 = 1$. A tree of depth d arises by making the root of a tree of depth $d-1$ the child of another root. The former root has size at least s_{d-1} and the latter root has at least this size by the weighted union rule. Thus $s_d \geq 2s_{d-1}$ and hence $s_d \geq 2^d$.

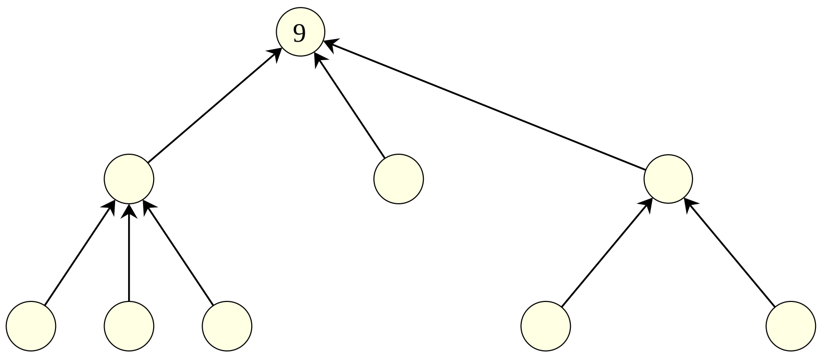


Figure 5.12 The weighted union rule: When the trees of Figure 5.10 are united the root of size 3 is made a child of the root of size 6.

[Tar75]. Here α is the so-called inverse Ackermann function; this function is extremely slowly growing and has value less than 5 even for $n = m = 10^{100}$, see [CLR90, Chapter 22] or [Meh84a, III.8.3].

5.5.3 *An Application of Partitions: Checking Priority Queues*

This section is joint work with Uli Finkler.

We will describe a checker for priority queues; this section assumes knowledge of the data type *p_queue*, see Section 5.4. We define a class *checked_p_queue*$<P, I>$ that can be wrapped around any priority queue *PQ* to check its behavior, see Figure 5.13. The resulting object behaves like *PQ*, albeit a bit slower, if *PQ* operates correctly. However, if *PQ* works incorrectly then this fact will be revealed ultimately. In other words the layer of software that we are going to design behaves like a watch-dog. It monitors the behavior of *PQ* and is silent if *PQ* works correctly. However, if *PQ* behaves incorrectly, the watch-dog barks.

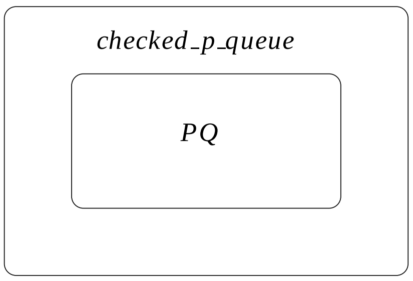


Figure 5.13 The class *checked_p_queue* wraps around a priority queue *PQ* and monitors its behavior. It offers the functionality of a priority queue.

How can the class *checked_p_queue* be used? Suppose we have designed a class *new_impl* which is a new implementation of priority queues. Using the implementation parameter mechanism we can write

```
_p_queue<P,I,new_impl> PQ;
```

to declare a *p_queue*$<P, I>$ which is implemented by *new_impl*. We may use *PQ* in any application using a *p_queue*$<P, I>$.

Assume now that *new_impl* is faulty. Then an application using *PQ* may go astray and we will have to locate the bug. Is it in *PQ* or is it in the application program? The use of *checked_p_queues* facilitates the debugging process greatly. We write

```
_p_queue<P,I,list_item,new_impl> PQ;
checked_p_queue<P,I> CPQ(PQ);
```

and use *CPQ* in the application program. If *PQ* works incorrectly, *CPQ* will tell us. There is *no* change required in the application program since *checked_p_queue* is publicly derived from *p_queue* and hence can be used wherever a *p_queue* can be used, for example,

```
void f(p_queue<P,I>&) { ...}
p_queue<P,I> PQ;                     f(PQ);
_p_queue<P,I,new_impl> PQI;         f(PQI);
_p_queue<P,list_item,new_impl>PQI1;
checked_p_queue<P,I> CPQ(PQI1);      f(CPQ);
```

Observe that the information type of *PQI1* is *list_item* instead of I, i.e., we are checking a *p_queue*$<P$, *list_item*$>$ instead of a *p_queue*$<P, I>$. This is a slight weakness of our solution. We believe that it is only a slight weakness because the information type I plays a minor role in the implementation of priority queues. Moreover, it can be overcome, see the exercises.

In the remainder of this section we give the implementation of the class *checked_p_queue*. The implementation is involved and reading this section certainly requires some stamina. We decided to put this section into the book because we strongly believe that the work on checkers is highly important for software libraries. Section 2.14 contains a general discussion on program checking.

The Idea: How can one monitor the behavior of a priority queue? Without concern for efficiency a solution is easy to come up with. Whenever a *delete_min* or *find_min* operation is performed all items of *PQ* are inspected and it is confirmed that the reported priority is indeed the minimum of all priorities in the queue. This solution does the job but defeats the purpose as it makes *delete_min* and *find_min* linear time operations. Our goal is a solution that adds only a small overhead to each priority queue operation. Our solution performs the checking of the items in the queue in a lazy way, i.e., when a *delete_min* or *find_min* operation is performed it is only recorded that all items currently in the queue must have a priority at least as large as the priority reported. The actual checking is done later. Note that this design implies that an error will not be detected immediately anymore but only ultimately.

Consider Figure 5.14. The top part of this figure shows the items in a priority queue from left to right in the order of their time. The *time* of a *pq_item it* is the time of the last *decrease_p* operation on *it* or, if there was none, the time of the addition of *it* to *PQ*. The

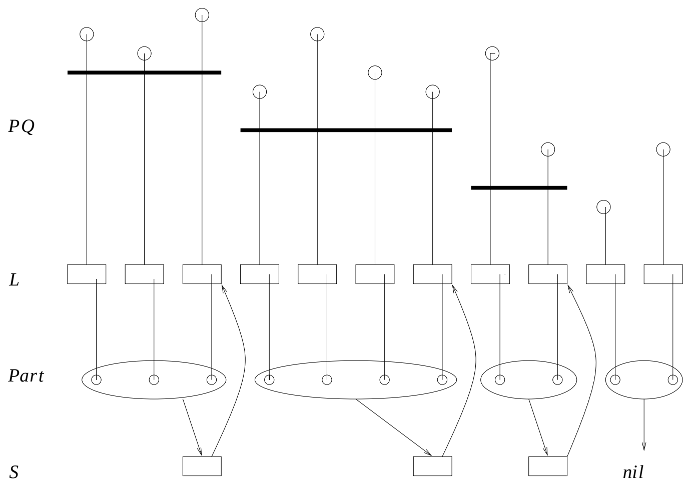


Figure 5.14 In the top part of the figure the items in a priority queue are shown as circles in the xy-plane. The x-coordinate corresponds to the time of an item and the y-coordinate corresponds to the priority of an item. The lower bounds for the priorities are indicated as heavy horizontal lines. The lower bound for the last two items is $-\infty$. The lower part of the figure illustrates our design of class *checked_p_queue*. The list L has one item for each item in *PQ*, the list S has one item for each step of L except for the step with lower bound $-\infty$ and the partition *Part* has one item for each item in L and one block for each step. The blocks of *Part* are indicated as ellipses. The information of the canonical item of a block of *Part* is the *S_item* associated with the block (*nil* for the block with lower bound $-\infty$). Each *S_item* knows the last *L_item* in its step.

vertical coordinate indicates the priority. With each item of the priority queue we have an associated lower bound. The *lower bound* for an item *it* is the maximal priority reported by any *delete_min* or *find_min* operation that took place after the time of *it*. We observe that *PQ* operates correctly if the priority of all *pq_items* is at least as large as their lower bound. We can therefore check *PQ* by comparing the priority of an item with its lower bound whenever an item is deleted from *PQ* or the time of an item is changed through a *decrease_p* operation.

How can we efficiently maintain the lower bounds of the items in the queue? We observe that lower bounds are monotonically decreasing from left to right, i.e., if the time of *it* is smaller than the time of *it*$'$ then the lower bound for *it* is at least as large as the lower bound for *it*$'$. This observation follows immediately from the definition of the lower bounds and leads to the staircaselike form of the lower bounds shown in Figure 5.14. We call a maximal segment of items with the same lower bound a *step*.

How does the system of lower bounds evolve over time? When a new item is added to the queue its associated lower bound is $-\infty$ and when a *find_min* or *delete_min* operation reports a priority of value p all lower bounds smaller than p are increased to p, i.e., all steps of value at most p are removed and replaced by a single step of value p. Since the staircase of lower bounds is falling from left to right this amounts to replacing a certain number of steps at the end of the staircase by a single step, see Figure 5.15.

How can we represent a staircase of lower bounds such that it can be updated efficiently

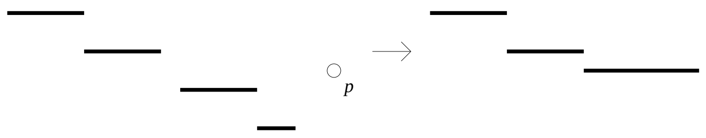


Figure 5.15 Updating the staircase of lower bounds after reporting a priority of p. All steps whose associated lower bound is at most p are replaced by a single step whose associated lower bound is p.

and such that lower bounds can be looked up efficiently? We keep a list L of *check_objects* and a list S of *step_objects*. We have one *check_object* in L for each item in *PQ* and order L according to the times of the corresponding item in *PQ*. We have one *step_object* in S for each step of our staircase of lower bounds except for the step whose associated lower bound is $-\infty$, see Figure 5.14.

A *check_object* is a quadruple consisting of a priority p, an information i, a *pq_item* and a *partition_item*. We explain the use of the partition item below. We mentioned already that check objects are in one-to-one correspondence to the items in *PQ* (if *PQ* operates correctly). The check object o corresponding to a *pq_item p_it* with associated priority p and associated information i contains p, i, and *p_it* as its first three components. We store p and i in the check object to guarantee *data integrity*, i.e., the checking layer stores its own copies of the pairs stored in the priority queue and hence can check whether *PQ* tampers with this data. In fact, we will not store the information i in *PQ* at all. We will rather use the information field of the item *p_it* of *PQ* to store the item of L containing o. In other words the queue to be checked will be of type *p_queue*<P, *L_item*> where *L_item* is a synonym for *list_item* that we reserve for the items in L. We use *l_it* as the canonical name of an *L_item*.

A *step_object* is a pair consisting of a priority and an *L_item*. The priority is the lower bound associated with the step and the *L_item* is the last item in L that belongs to the step. The list S will play a crucial role when we update our set of lower bounds after a *del_min* or *find_min* operation. When a priority p is reported by a *del_min* or *find_min* all steps whose *step_object* has a priority of at most p are merged into a single step. These steps constitute a final segment of S. We use *S_item* as the name of the items in S and use *s_it* as the canonical name of an *S_item*.

For the efficient lookup of lower bounds we use a *Partition*<*S_item*> *Part* with one item for each item in L and one block for each step of L. The information associated with the canonical element of a step is *nil*, if the step's lower bound is $-\infty$, and is the *S_item* corresponding to the step otherwise. The fourth component of each check object is the partition item corresponding to the check object.

Let us summarize. A checked priority queue consists of a priority queue, the lists L and S, a partition *Part*, and two integer counters *phase_length* and *op_count* (their use will be explained below). The items of L are in one-to-one correspondance to the items of *PQ* (if *PQ* operates correctly). All operations on *PQ* go through the checking layer, e.g., an operation *insert*(p, i) causes the checking layer to update its internal data structures, in particular, to add an item to L, and to forward the insert request to *PQ*. The new *L_item* will be returned by the insert operation.

The Class checked_p_queue: We fix the definitions of the data structures of the checking layer in the following layout for the class *checked_p_queue*$<P, I>$.

⟨*checked_p_queue.h*⟩≡

```
#ifndef LEDA_CHECKED_P_QUEUE_H
#define LEDA_CHECKED_P_QUEUE_H
#include <LEDA/p_queue.h>
#include <LEDA/list.h>
#include <LEDA/partition.h>
#include <assert.h>
#include <LEDA/tuple.h>
template <class P, class I>
class checked_p_queue : public p_queue<P,I>
{
  typedef  four_tuple<P,I,pq_item,partition_item> check_object;
  list<check_object> L;
  typedef list_item L_item;
  typedef two_tuple<P,L_item> step_object;
  list<step_object> S;
  typedef list_item S_item;
  Partition<S_item> Part;
  int phase_length, op_count;
  p_queue<P,L_item>* PQ;
  ⟨private member functions of class checked_p_queue⟩
  /* the default copy constructor and assignment operator work
     incorrectly, we make them unaccessible by
     declaring them private                                  */
  checked_p_queue(const checked_p_queue<P,I>& Q);
  checked_p_queue<P,I>& operator=(const checked_p_queue<P,I>& Q);
public:
  checked_p_queue(p_queue<P,L_item>& PQ_ext) // constructor
  { PQ = &PQ_ext;
    assert(PQ->empty());
    phase_length = 4; op_count = 0;
  }
  ⟨member functions of class checked_p_queue⟩
};
#endif
```

Observe that *checked_p_queue*$<P, I>$ is publicly derived from *p_queue*$<P, I>$ and hence will offer the same functions as *p_queues*. The private data members are a pointer to the *p_queue*$<P,$ *L_item*$>$ to be checked, the lists L and *S*, the partition *Part*, and two integers *phase_length* and *op_count*; we will explain the latter two data members below.

The constructor of *checked_p_queue* gets a reference to the queue to be checked and stores it in *PQ*. It also initializes *phase_length* to four and *op_count* to zero. The queue to be checked must be empty (but, of course, there is no guarantee that the emptiness test does not lie). All other data members are initialized by their default constructor.

The member functions of *checked_p_queue* split into private and public member functions. The public member functions are exactly the public member functions of the base class *p_queue* except for the copy constructor and the assignment operator. We were too lazy to implement them. Since C++ provides default implementations of both functions and since the default implementations are incorrect we declared both functions private to make them unaccessible.

The private member functions are used in the implementation of the public member functions. In order to motivate their definitions we give an overview of the implementations of the public member functions *insert* and *delete_min*. In this overview we concentrate on the interplay between the checking layer and *PQ* and do not give any details on how the staircase of lower bounds is manipulated.

An $insert(p, i)$ is realized as follows. The checking layer creates a check object o containing the pair (p, i) and appends o to L. Let *Lit* be new *L_item*. It then inserts the pair (p, Lit) into *PQ*. *PQ* returns an item *p_it* which the checking layer records in o. The checking layer also creates a new partition item corresponding to o. The new item either forms a block of its own (if the step with lower bound $-\infty$ is empty) or is joined into the step with lower bound $-\infty$. The checking layer then returns *Lit* as the result of the *insert*.

A *del_min* is realized as follows. The checking layer forwards the request to *PQ* and *PQ* returns a pair (p, Lit). Let $o = (p', i, p\_it, part\_it)$ be the checking object stored in *Lit*. The checker verifies that $p = p'$ and that p satisfies the lower bound associated with o, it updates the staircase of lower bounds, and it finally returns p.

We want to stress that the checking layer is responsible for the communication with the environment and that the checking layer stores all the pairs (p, i) that are in the priority queue. It forwards all requests from the environment to *PQ*. In a *del_min* operation it uses *PQ* as an *oracle*. The checking layer has no own means to answer minimum queries. It therefore asks *PQ* to point out the correct item. It maintains the system of lower bounds in order to find out whether *PQ* ever lied to it. The checker discovers lies by checking the lower bounds of items whenever an item is deleted or the priority of an item is changed.

We want to bound the delay between a lie and its discovery. For this purpose the checker has a private member function *periodic_check*. This operation goes through all elements of L and checks the lower bound of every element. *Periodic_check* is called after the 2^l-th operation performed on the priority queue for all $l \geq 2$. It is also called after the last operation performed on the priority queue. The integers *phase_length* and *op_count* are used to control the periodic checks. We divide the execution into phases. We use *phase_length* for the length of the current phase and use *op_count* to count the number of operations in the current phase. When *op_count* reaches *phase_length* we check all lower bounds, double *phase_length*, and reset *op_count* to zero.

The discussion above implies that we need to make two assumptions about the behavior of *PQ*:

- All calls to member functions of *PQ* must terminate. They may give wrong answers but they must terminate. It is beyond our current implementation to guarantee termination. A solution would require non-trivial but standard modification of the implementation of *PQ*. One can guard against run-time errors (e.g., invalid addresses) by compiling *PQ* with the debugging option and one can guard against infinite loops by specifying an upper bound on the execution time of each member function of *PQ*. The latter requires a worst case analysis of the running time of *PQ*'s member functions.
- All calls of $PQ \rightarrow inf$ must return valid *L_items*. One may guard against invalid *L_items* by compiling *checked_p_queues* with the debugging option. An alternative solution is described at the end of this section.

Private Member Functions: We are now ready for the definition of the private member functions. The first group provides natural access to the components of *check_objects* and *step_objects*.

⟨*private member functions of class checked_p_queue*⟩≡
```
P& prio(L_item l_it)                         { return L[l_it].first(); }
const P&  prio(L_item l_it) const            { return L[l_it].first(); }
I& inf(L_item l_it)                          { return L[l_it].second(); }
const I&  inf(L_item l_it) const             { return L[l_it].second(); }
pq_item& pq_it(L_item l_it)                  { return L[l_it].third(); }
pq_item  pq_it(L_item l_it) const            { return L[l_it].third(); }
partition_item& part_it(L_item l_it)         { return L[l_it].fourth(); }
partition_item  part_it(L_item l_it) const { return L[l_it].fourth(); }
P& prio_of_S_item(S_item s_it)               { return S[s_it].first(); }
P  prio_of_S_item(S_item s_it) const         { return S[s_it].first(); }
L_item& L_it(S_item s_it)                    { return S[s_it].second(); }
L_item  L_it(S_item s_it)  const             { return S[s_it].second(); }
```

The second group supports the navigation in the data structures of the checker.

The canonical partition item corresponding to an *L_item Lit* is obtained by performing *Part.find* on the associated partition item.

The information associated with the canonical item is obtained by applying *Part.inf* to the canonical item.

The item *Lit* belongs to the step with lower bound $-\infty$ if the canonical information is equal to *nil* and belongs to a step with a defined lower bound otherwise.

The last item in the step containing *Lit* is either the last item of L (if *Lit* is unrestricted) or is the *L_item* stored in the *S_item* given by *canonical_inf*(*Lit*).

An item is the only item in its step if it is the last item in its step and is either the first item of L or its predecessor item in L is also the last item in its step.

All functions above are *const*-functions. They use operations *find* and *inf* of class *Partition* which are not *const*-functions. We therefore write ((*Partition*<*S_item*>∗) &*Part*) → instead of *Part.* to cast away *const* when calling one of these functions[11].

⟨*private member functions of class checked_p_queue*⟩+≡

```
partition_item canonical_part_it(L_item l_it) const
{ return ((Partition<S_item>*) &Part)->find(part_it(l_it)); }
S_item canonical_inf(L_item l_it) const
{ return ((Partition<S_item>*) &Part)->inf(canonical_part_it(l_it)); }
bool is_unrestricted(L_item l_it) const
{ return canonical_inf(l_it) == nil; }
bool is_restricted(L_item l_it) const
{ return ! is_unrestricted(l_it); }
L_item  last_item_in_step(L_item l_it) const
{ if ( is_restricted(l_it) )
      return L_it(canonical_inf(l_it));
  return L.last();
}
bool is_last_item_in_step(L_item l_it) const
{ return ( last_item_in_step(l_it) == l_it) ; }
bool is_only_item_in_step(L_item l_it) const
{ return (is_last_item_in_step(l_it) &&
   ( L.pred(l_it) == nil || is_last_item_in_step(L.pred(l_it))));
}
```

We put the functions above to their first use by writing a function that tests the validity of the data structures of the checking layer. This function is for debugging purposes only. The data structures must satisfy the following conditions:

- The sizes of L and PQ must be equal.
- Each item *Lit* in L points to an item in PQ which points back to *Lit*.
- The items in L can be partitioned into segments such that in each segment the value of *canonicaLinf* is constant. Except for maybe the last segment, the *canonicaLinf* is equal to an item in S and this item points back to the last *L_item* in the segment. In the last segment the *canonicaLinf* is *nil*. The last segment may be empty and all other segments are non-empty.

⟨*private member functions of class checked_p_queue*⟩+≡

```
void validate_data_structure() const
{
#ifdef VALIDATE_DATA_STRUCTURE
  assert( PQ->size() == L.size() );
  L_item l_it;
  forall_items(l_it,L)
    { assert( pq_it(l_it) != nil ) ;
```

11 It is tempting to write the cast as ((*Partition*<*S_item*>) *Part*). but this would amount to a call of the copy constructor of *Partition* and hence be a disaster.

```
        assert( PQ->inf(pq_it(l_it)) == l_it );
      }
    l_it = L.first();
    S_item s_it = S.first();
    while (s_it)
      { assert(canonical_inf(l_it) == s_it);
        while (l_it != L_it(s_it) )
          { l_it = L.succ(l_it);
            assert(l_it != nil);
            assert(canonical_inf(l_it) == s_it);
          }
        s_it = S.succ(s_it);
        l_it = L.succ(l_it);
      }
    while (l_it)
      { assert(canonical_inf(l_it) == nil);
        l_it = L.succ(l_it);
      }
#endif
}
```

The final group of private member functions checks lower bounds and update the staircase of lower bounds.

An *Litem Lit* satisfies its lower bound if either *Lit* is unrestricted or the priority of the step containing *Lit* is no larger than the priority of *p_it*.

⟨*private member functions of class checked_p_queue*⟩+≡

```
void check_lower_bound(L_item l_it) const
{ assert(is_unrestricted(l_it) ||
         compare(prio_of_S_item(canonical_inf(l_it)), prio(l_it)) <= 0 );
}
```

The function *periodic_check* is called at the end of every public member function. It increases *op_count* and when *op_count* has reached *phase_length* checks all lower bounds, doubles *phase_length*, and resets *op_count* to zero.

⟨*private member functions of class checked_p_queue*⟩+≡

```
void periodic_check()
{ if ( ++op_count == phase_length )
  { L_item l_it;
    forall_items(l_it,L) check_lower_bound(l_it);
    phase_length = 2*phase_length;
    op_count = 0;
  }
}
```

Finally, we show how to update lower bounds, see Figure 5.14. Let p be a priority. We move all lower bounds that are smaller than p up to p. This amounts to removing all items in S whose associated lower bound is less than or equal to p and adding a new item with priority p to S. We give more details.

If L is empty or the last step in our staircase of lower bounds extends to the end of the list and has a priority at least as large as p then there is nothing to do.

So assume otherwise. We scan S form its right end (= rear end) and remove items as long as their priority is at most p. Whenever we remove an item *s_it* from S we join the step corresponding to *s_it* with the step after it (it it exists). Finally, we add an item to S representing a step with priority p and ending at *L.last*( ) and make the item the canonical information of all items in the last step.

$\langle$*private member functions of class checked_p_queue*$\rangle{+}{\equiv}$

```
void update_lower_bounds(P p)
{ if ( L.empty() ||
       ( !S.empty() && compare(prio_of_S_item(S.last()),p) >= 0
         && L_it(S.last()) == L.last()))  return;
  S_item s_it;
  while ( !S.empty()  &&
          compare(prio_of_S_item(s_it = S.last()),p) <= 0 )
    { L_item l_it = L_it(s_it);
      if ( L.succ(l_it) )
        Part.union_blocks(part_it(l_it),part_it(L.succ(l_it)));
      S.pop_back();
    }
  Part.change_inf(canonical_part_it(L.last()),
        S.append(step_object(p,L.last())));
}
```

After all this preparatory work we come to the public member functions.

The Insert Operation: To insert a new item $\langle p, i\rangle$ we append to L a new check object $(p, i, p\_it, part\_it)$; *p_it* is a new item in *PQ* created by the insertion of $(p, -)$ and *part_it* is a new item in *Part*. The lower bound of the new item is $-\infty$ and hence the information associated with *part_it* is *nil*. Let *l_it* be the new item in L. We store *l_it* as the information of *p_it*.

If there was already a step with lower bound $-\infty$, we add the new item to this block.

Finally, we call *periodic_check* and return *l_it* (after casting it to *pq_item*)[12].

$\langle$*member functions of class checked_p_queue*$\rangle{\equiv}$

```
pq_item insert(const P& p, const I& i)
{ pq_item p_it = PQ->insert(p,(L_item) 0);
  L_item last_l_it = L.last(); // last item in old list
  partition_item pa_it = Part.make_block((S_item) 0);
  list_item l_it = L.append(check_object(p,i,p_it,pa_it));
  PQ->change_inf(p_it,l_it);
  if (last_l_it && is_unrestricted(last_l_it) )
```

[12] The cast from *L_item* to *pq_item* is necessary since early in the design of LEDA we made the decision that the global type *pq_item* is the return type of *insert*. It would be more elegant to have *pq_item* as a type local to *p_queue*.

```
      Part.union_blocks(part_it(l_it),part_it(last_l_it));
  periodic_check();
  validate_data_structure();
  return (pq_item) l_it;
}
```

The Find_min Operation: In order to perform a *find_min* operation we perform a *find_min* operation on *PQ* and extract an item *Lit* in L from the answer. Having received this advice from *PQ* we check the lower bound for *Lit* and update the system of lower bounds using the priority of *Lit*.

Since *checked_p_queue* is derived from *p_queue*, since *find_min* is a *const*-function of *p_queue*, and since *update_lower_bounds* and *perodic_check* are not, we need to cast away the *const*.

⟨*member functions of class checked_p_queue*⟩+≡

```
pq_item find_min() const
{ L_item l_it = PQ->inf(PQ->find_min());
  check_lower_bound(l_it);
  ((checked_p_queue<P,I>*)this)->update_lower_bounds(prio(l_it));
  ((checked_p_queue<P,I>*)this)->periodic_check();
  validate_data_structure();
  return (pq_item) l_it;
}
```

The Delete Operation: To delete an item *p_it* we check its lower bound, we delete it from *PQ*, and we delete the corresponding *L_item Lit* from L. If *Lit* is restricted and is the only item in its step, we delete the item in S representing the step and if *Lit* is the last item in its step but not the only item in its step, we change the *L_it*-field of *canonical_inf*(*Lit*) to the predecessor of *Lit* in L. We should also delete the item corresponding to *p_it* from *Part*. Unfortunately, *partition* does not offer a delete operation. We comment on this point at the end of the section.

⟨*member functions of class checked_p_queue*⟩+≡

```
void del_item(pq_item p_it)
{ L_item l_it = (L_item) p_it;
  check_lower_bound(l_it);
  if ( is_restricted(l_it) )
  { if ( is_only_item_in_step(l_it) )
      S.del_item(canonical_inf(l_it));
    else if (is_last_item_in_step(l_it) )
            L_it(canonical_inf(l_it)) = L.pred(l_it);
  }
  PQ->del_item(pq_it(l_it));
  L.del_item(l_it);
```

```
    periodic_check();
    validate_data_structure();
  }
```

To perform a *del_min* operation we perform a *find_min* on *PQ* and then a *del_min* on the item returned. Finally, we update the lower bound according to the priority of the item returned.

⟨*member functions of class checked_p_queue*⟩+≡
```
  P del_min()
  { L_item l_it = PQ->inf(PQ->find_min());
    P p = prio(l_it);
    del_item((pq_item)l_it);
    update_lower_bounds(p);
    periodic_check();
    validate_data_structure();
    return p;
  }
```

Miscellaneous Functions: The functions *prio, inf, change_inf, size* and *empty* reduce to appropriate functions of the checking layer.

⟨*member functions of class checked_p_queue*⟩+≡
```
  const P& prio(pq_item it) const
  { ((checked_p_queue<P,I>*)this) -> periodic_check();
    return prio((L_item) it);
  }
  const I& inf(pq_item it) const
  { ((checked_p_queue<P,I>*)this) -> periodic_check();
    return inf((L_item) it);
  }
  void change_inf(pq_item it, const I& i)
  { periodic_check();
    inf((L_item) it) = i ;
  }
  int  size() const
  { ((checked_p_queue<P,I>*)this) -> periodic_check();
    return L.size();
  }
  bool empty() const
  { ((checked_p_queue<P,I>*)this) -> periodic_check();
    return L.empty();
  }
```

The Decrease_p Operation: In order to perform a *decrease_p* on item *Lit* we check whether the current priority satisfies its lower bound and we check whether the *decrease_p* operation actually decreases the priority of *Lit*. If so, we change the priority of *Lit* and forward the change to *PQ*.

The new lower bound for the item *Lit* is $-\infty$. If the old lower bound was also $-\infty$ then no action is required. Otherwise we must move *Lit* from its current position in L to the last position in L. This affects the step that contained *Lit*. If *Lit* was the only item in the step, we remove the step altogether and if *Lit* was the last, but not the only, item in its step, we record that *Lit*'s predecessor is the new last element in the step.

In order to move *Lit* to the last position of L we split L into three pieces (the items before *Lit*, *Lit*, and the items after *Lit*) and then reassemble the pieces. We allocate a new partition item for *Lit* and set its information to *nil* (since the new lower bound for *Lit* is $-\infty$). If the step with lower bound $-\infty$ was non-empty, we add *Lit* to this step.

⟨*member functions of class checked_p_queue*⟩$+\equiv$

```
void decrease_p(pq_item p_it, const P& p)
{ L_item l_it = (L_item) p_it;
  check_lower_bound(l_it);
  assert( compare(p,prio(l_it)) <= 0 );
  prio(l_it) = p;
  PQ->decrease_p(pq_it(l_it),p);
  if ( is_restricted(l_it) )
  { if ( is_only_item_in_step(l_it) ) S.del_item(canonical_inf(l_it));
    else if (is_last_item_in_step(l_it) )
            L_it(canonical_inf(l_it)) = L.pred(l_it);
    list<check_object> L1, L_it;
    L.split(l_it,L,L1,LEDA::before);
    L1.split(l_it,L_it,L1,LEDA::after);
    L.conc(L1);
    list_item last_it = L.last();
    L.conc(L_it);
    part_it(l_it) = Part.make_block((S_item) 0);
    if (last_it && is_unrestricted(last_it) )
      Part.union_blocks(part_it(l_it),part_it(last_it));
  }
  periodic_check();
  validate_data_structure();
}
```

The Clear Operation and the Destructor: Finally, to clear our data structure we check the lower bounds of all items and then clear for *PQ*, L, S, and *Part*. The destructor calls *clear*.

⟨*member functions of class checked_p_queue*⟩$+\equiv$

```
void clear()
{ L_item l_it;
  forall_items(l_it,L) check_lower_bound(l_it);
```

```
    PQ->clear(); L.clear(); S.clear(); Part.clear();
  }
  ~checked_p_queue() { clear(); }
```

Efficiency: We have now completed the definition of our checker for priority queues. How much overhead does it add? The body of any function of class *checked_p_queue* consists of a call of the same function of *PQ* plus a constant number of calls to functions of L, S, and *Part*, a call to *periodic_check* plus (maybe) a call of *update_lower_bounds*.

Update_lower_bounds adds at most one element to S (and no other function does) and removes zero or more entries from S. We conclude that the total number of elements added to S and hence removed from S is bounded by the number of operations on *PQ*. A call of *update_lower_bounds* that removes k elements from S has cost $O(1 + k)$ plus the cost for $O(1 + k)$ operations on a partition. We conclude that all calls of *update_lower_bounds* contribute a linear number of operations on *Part*. Therefore each call to *update_lower_bounds* contributes a constant number of operations on *Part* in the amortized sense.

The cost of a call to *periodic_check* is also amortized constant. This follows from the fact that the number of elements in the queue is at most twice *phase_length*, that the cost of a call is either $O(1)$ or $O(\textit{phase\_length})$, and that the latter alternative occurs only in every *phase_length*-th call to *perodic_check*.

We conclude that the *amortized overhead for each operation on PQ is a constant number of operations on lists and partitions*. Operations on lists require constant time and operations on partitions requires $\alpha(n)$ time.

An Experiment: The following program compares unchecked and checked priority queues experimentally. We generate an array of n random doubles and then use a binary heap to sort them. We first use the binary heap directly and then wrap it into a *checked_p_queue*. The running time of the checked version is about two times the running time of the unchecked version, e.g., it takes about 6.1 seconds to sort 100000 doubles with the unchecked version and slightly more than 12 seconds with the checked version.

⟨*checked_p_queue_demo.c*⟩≡
  ⟨*checked_p_queue demo: includes*⟩
```
  main(){
```
  ⟨*checked_p_queue demo: read n*⟩
```
  array<double> A(n);
  random_source S;
  for (int i = 0; i < n; i++) S >> A[i];
  float T = used_time();
  { _p_queue<double,int,bin_heap> PQ(n);
    for (int i = 0; i < n; i++) PQ.insert(A[i],0);
    while ( !PQ.empty() ) PQ.del_min();
  }
  float T1 = used_time(T);
```

```
{ _p_queue<double,list_item,bin_heap> PQ(n);
  checked_p_queue<double,int> CPQ(PQ);
  for (int i = 0; i < n; i++) CPQ.insert(A[i],0);
  while ( !CPQ.empty() ) CPQ.del_min();
}
float T2 = used_time(T);
⟨checked_p_queue demo: report times⟩
}
```

We made a similar test with the priority queue in Dijkstra's algorithm and observed a slowdown by a factor of about 2.5.

Final Remarks: We close this section with a discussion of some alternatives and improvements to our design.

The overhead introduced by our design is a constant number of operations on lists and partitions for each priority queue operation. Since operations on partitions take slightly super-linear time this invalidates the $O(1)$ upper bound for the *decrease_p* operation in the *f_heap* and *p_heap* implementation of priority queues. This can be remedied as follows. The class *checked_p_queue* uses the type *Partition* in a very special way. The blocks of L partition L into contiguous segments and all unions are between adjacent segments. For this special situation there is a realization of partitions that supports all operations in constant time, see [GT85].

Partitions do not offer an operation that deletes items and hence the *del_item* operation of *checked_p_queue* can only delete the items in *PQ*, L, and S, but cannot delete the item in *Part*. This shortcoming can be remedied by giving partitions a *del_item* operation. We briefly sketch the implementation. We perform deletions in a lazy way. When an item is to be deleted it is marked for deletion. We also keep track of the total number of items in the partition and the number of items that are marked for deletion. When more than three-quarters of the items are marked for deletion the partition data structure is cleaned. We go through all items (recall that they are linked into a singly linked list) and perform a find operation for each item. This makes all trees depth one. Then we delete all marked items except those that are the root of a non-trivial tree.

In our realization the checker puts some trust into *PQ*, namely that $PQ \rightarrow inf$ always returns a valid *L_item*. This shortcoming can be overcome by introducing a level of indirection into the data structure. We add an *array<L_item> A*. When the queue has size n precisely the first n entries of this array are used. When an item *p_it* of *PQ* stores a list item *Lit* in the current design it stores some integer $i \in [0 .. n-1]$ in the new design and $A[i]$ contains *Lit*. In this way the index-out-of-bounds check for arrays allows us to check for an invalid pointer. When an item is deleted from the queue and this item corresponds to position i of A, this position is first swapped with position $n-1$ and then the last entry is removed. We leave the details to the reader. This solution is inspired by [AHU74, exercise 2.12].

The class *checked_p_queue* catches errors of the underlying priority queue eventually (at the latest at the next call of *periodic_check*) but not immediately. Is there a solution which guarantees immediate error detection? Yes and No. **Yes**, because we could simply put a correct priority queue implementation into the checker, and **no**, because it can be shown that no data structure whose running time has a smaller order of magnitude than the running time of priority queues can guarantee immediate error detection.

Exercises for 5.5

1 Modify the program checked_p_queue_demo so that you can experiment with different implementations of priority queues and not only with the binary heap implementation.

2 Implement the copy constructor and the assignment operator of our class *checked_p_queue*.

3 Modify the implementation of class *checked_p_queue*, so as to remove the assumption that $PQ \rightarrow inf$ always returns a valid *L_item*.

4 Modify the implementation of class *checked_p_queue* so that the queue to be checked has type *p_queue*<P, I>. Hint: Use a map to make the correspondence between *pq_items* and the items of L.

5 Use checked priority queues instead of priority queues in Dijkstra's algorithm as discussed in Section 5.4.

6 Add an operation *del_item* to the types *partition* and *Partition*<E>. Follow the sketch at the end of Section 5.5.3.

7 In the extract minimum problem we are given a permutation of the integers 1 to n interspersed with the letter E, e.g., 6,E,1,4,3,E,E,5,2,E,E,E is a possible input sequence. The E's are processed from left to right. Each E extracts the smallest number to its left which has not been extracted by a previous E. The output in our example would therefore be 6,1,3,2,4,5. Solve the problem using a priority queue. In the off-line version of this problem the input sequence is completely known before the first output needs to be produced. Solve the problem with the partition data type (Hint: Determine first which E outputs the number 1, then which E outputs 2, ...).

8 Implement the data structure of [GT85]. Make it available as a LEDA extension package.

5.6 Sorted Sequences

Sorted sequences are a versatile data type. We discuss their functionality in this section, give their implementation by means of skiplists in the next section, and apply them to Jordan sorting in the last section of this chapter.

A *sorted sequence* is a sequence of items in which each item has an associated key from a linearly ordered type K and an associated information from an arbitrary type I. We call K the key type and I the information type of the sorted sequence and use $\langle k, i \rangle$ to denote an item with associated key k and information i. The keys of the items of a sorted sequence must be in strictly increasing order, i.e., if $\langle k, i \rangle$ is before $\langle k', i' \rangle$ in the sequence then k is before k' in the linear order on K. Here comes a sorted sequence of type *sortseq*<*string*, *int*>:

$\langle Ena, 7\rangle \; \langle Kurt, 4\rangle \; \langle Stefan, 2\rangle \; \langle Ulli, 8\rangle$

Sorted sequences offer a wide range of operations. They can do almost everything lists, dictionaries, and priority queues can do and they can do many other things. They even do all these things with the same asymptotic efficiency. Of course, there is a price to pay: Sorted sequences require more space (about $23.33n$ bytes for a sequence of n items) and the constant factors in the time bounds are larger. So please use sorted sequences only if you need their power.

We discuss the functionality of sorted sequences in several steps. In each step we introduce some operations and then give a small program using these operations. We start with the operations that we know already from dictionaries and priority queues, then turn to so-called finger searches, and finally discuss operations for splitting and merging sorted sequences.

Basic Functionality: Sorted sequences come in two kinds. The definitions

```
 sortseq<K,I> S;
_sortseq<K,I,ab_tree> T;
```

define S and T as sorted sequences with key type K and information type I. For T the ab_tree implementation of sorted sequences is chosen and for S the default implementations of types *sortseq* is chosen. The type *_sortseq*<K, I, *IMPL*> offers only a subset of the operations of *sortseq*<K, I>; in particular it does not offer any of the finger search operations. The items in a sorted sequence have type *seq_item*. The following implementations of *sortseqs* are currently available: skiplists [Pug90b], randomized search trees [AS89], $BB(\alpha)$-trees [NR73], ab-trees [AHU74, HM82], and red-black-trees [GS78]. They are selected by the implementation parameters skiplist, rs_tree, bb_tree, ab_tree, and rb_tree, respectively. Skiplists are the default implementation. We have mentioned already that sorted sequences extend dictionaries, lists, and priority queues, in particular we have the following operations:

K	S.key(*seq_item it*)	returns the key of item *it*. *Precondition*: *it* is an item in S.
I	S.inf(*seq_item it*)	returns the information of item *it*. *Precondition*: *it* is an item in S.
seq_item	S.lookup(K k)	returns the item with key k (*nil* if no such item exists in S).
seq_item	S.locate(K k)	returns the item $\langle k', i\rangle$ in S such that k' is minimal with $k' \geq k$ (*nil* if no such item exists).
seq_item	S.locate_succ(K k)	equivalent to *S.locate*(k).
seq_item	S.succ(K k)	equivalent to *S.locate*(k).

seq_item	*S*.locate_pred($K\ k$)	returns the item $\langle k', i\rangle$ in *S* such that k' is maximal with $k' \le k$ (*nil* if no such item exists).
seq_item	*S*.pred($K\ k$)	equivalent to *S.locate_pred*(k).
seq_item	*S*.min_item( )	returns the item with minimal key (*nil* if *S* is empty).
seq_item	*S*.max_item( )	returns the item with maximal key (*nil* if *S* is empty).
seq_item	*S*.succ(*seq_item it*)	returns the successor item of *it* in the sequence containing *it* (*nil* if there is no such item).
seq_item	*S*.pred(*seq_item* x)	returns the predecessor item of *it* in the sequence containing *it* (*nil* if there is no such item).
seq_item	*S*.insert($K\ k,\ I\ i$)	associates information i with key k: If there is an item $\langle k, j\rangle$ in *S* then j is replaced by i, otherwise a new item $\langle k, i\rangle$ is added to S. In both cases the item is returned.
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 sorted sequence.
void	*S*.del($K\ k$)	removes the item with key k from *S* (null operation if no such item exists).
void	*S*.del_item(*seq_item it*)	removes the item *it* from the sequence containing *it*.
void	*S*.change_inf(*seq_item it*, $I\ i$)	makes i the information of item *it*.

The operations *key*, *inf*, *succ*, *pred*, *max*, *min*, *del_item*, *change_inf*, *size*, and *empty* take constant time, *lookup*, *locate*, *locate_pred*, and *del* take logarithmic time, and *clear* takes linear time.

We come to our first program. We read a sequence of strings (terminated by "stop") and build a sorted sequence of type *sortseq<string, int>* for them[13]. Then we read a pair (*s1*, *s2*) of strings and output all input strings larger than or equal to *s1* and smaller than or equal to *s2*. This is done as follows. If *s2* is smaller than *s1* then there are no such strings. Assume otherwise and let item *last* contain the largest string less than or equal to *s2* and let *first* contain the smallest string larger or equal to *s1*. If either *first* or *last* does not exist or *last* is

[13] Observe that a sorted sequence needs an information type; we do not need informations in this application and have chosen the information type *int*; any other type would work equally well.

the predecessor of *first* then the answer is empty. Otherwise it consists of all strings that are stored in the items starting at *first* and ending at *last*.

⟨*sortseq_demo1.c*⟩≡

```
#include <LEDA/sortseq.h>
main()
{ sortseq<string,int> S;
  string s1,s2;
  cout << "Input a sequence of strings terminated by stop.\n";
  while (cin >> s1 && s1 != "stop") S.insert(s1, 0);
  while ( true )
  { cout << "\nInput a pair of strings.\n\n";
    cin >> s1 >> s2;
    cout << "All strings s with " <<
                              s1 <<" <= s <= " << s2 <<":\n";
    if ( s2 < s1 ) continue;
    seq_item last  = S.locate_pred(s2);
    seq_item first = S.locate(s1);
    if ( !first || !last || first == S.succ(last) ) continue;
    seq_item it = first;
    while ( true )
    { cout << "\n" << S.key(it);
      if ( it == last ) break;
      it = S.succ(it);
    }
  }
}
```

The running time of this program is $O(n \log n + m \log n + L)$, where n denotes the number of strings put into the sorted sequence, m denotes the number of queries, and L is the total number of strings in all answers. In this time bound we have assumed for simplicity that a comparison between strings takes constant time and that a string can be printed in constant time. Both assumptions require that the strings have bounded length.

Finger Search: All search operations discussed so far take logarithmic time. *Finger search* opens the possibility for sub-logarithmic search time. It requires that the position of the key k to be searched for is approximately known. Let *it* be an item of the sorted sequence S; in the context of finger search we call *it* a *finger* into S. The operations

```
S.finger_locate(k);
S.finger_locate_from_front(k);
S.finger_locate_from_rear(k);
S.finger_locate(it, k);
```

have exactly the same functionality as the operation *locate*, i.e., all of them return the leftmost item *it*$'$ in S having a key at least at large as k. They differ in their running time. If *it*$'$ is the d-th item in a list of n items then the first three operations run in time $O(\log \min(d, n-d))$, $O(\log d)$, and $O(\log(n-d))$, respectively[14]. In other words,

[14] For the remainder of this section we assume $\log x$ to mean $\max(0, \log x)$.

finger_locate_from_front is particularly efficient for searches near the beginning of the sequence, *finger_locate_from_end* is particularly efficient for searches near the end of the sequence, and *finger_locate* is particularly efficient for searches near either end of the sequence (however, with a larger constant of proportionality); it runs the two former functions in parallel and stops as soon as one of them stops. The operation *S.finger_locate*(it, k) runs in time $O(\log \min(d, n-d))$ where d is the number of items in S between *it* and *it*$'$. For example, if *it* is the 5th item of S and *it*$'$ is the 17th item then $d = 17 - 5 = 12$.

After a fast search we also want to insert fast. That's the purpose of the operation *insert_at*. Assume that *it* is an item of S and k is a key and that *it* is either the rightmost item in S with $key(it) < k$ or the leftmost item with $key(it) > k$. Then

```
S.insert_at(it, k, i)
```

adds $\langle k, i\rangle$ to S in time $O(1)$. If k's relation to the key of *it* is known then it is more efficient to use

```
S.insert_at(it, k, i, dir)
```

with *dir* equal to *LEDA*::*before* or *LEDA*::*after*.

We give an application of finger searching to sorting. More precisely, we give a sorting algorithm which runs fast on inputs that are nearly sorted. Let n and f be integers with $0 \le f \ll n$ and consider the sequence

$$n-1, n-2, \ldots, n-f, 0, 1, 2, \ldots, n-f-1.$$

We store this sequence in a list L and sort it in five different ways: four versions of insertion sort and, for comparison, the built-in sorting routine for lists. The easiest way to build a sorted sequence S from L is to call *S.insert* for each element of L. As before, we must give our sorted sequence an information type; we use the type *int* and hence insert the pair $(k, 0)$ for each element k of L.

⟨*repeated insertion sort*⟩≡

```
forall(k,L) S.insert(k, 0);
```

The running time of repeated insertion sort is $O(n \log n)$.

Let us take a closer look where the insertions are taking place for our input sequence. In the first f insertions the new element is always inserted at the beginning of the sequence and in the remaining $n - f$ insertions the new element is always inserted before the f-th element from the end of the sequence. Since $f \ll n$ it should be more efficient to search for the place of insertion from the rear end of the sequence.

⟨*finger search from rear end*⟩≡

```
forall(k, L)
{ if (S.empty()) it = S.insert(k, 0);
  else
  { seq_item it = S.finger_locate_from_rear(k);
    if (it) S.insert_at(it,k,0,LEDA::before);
```

```
      else S.insert_at(S.max_item(),k,0,LEDA::after);
    }
  }
```

With finger search from the rear end each search takes time $O(\log f)$ and hence the total running time becomes $O(n \log f)$. The same running time results if we use the version of finger search that does not need to be told from which end of the sequence it should search.

⟨*finger search from both ends*⟩≡

```
  forall(k, L)
  { if (S.empty()) it = S.insert(k, 0);
    else
    { seq_item it = S.finger_locate(k);
      if (it) S.insert_at(it,k,0,LEDA::before);
      else S.insert_at(S.max_item(),k,0,LEDA::after);
    }
  }
```

We can do even better by observing that each insertion takes place next to the previous insertion. Hence it is wise to remember the position of the last insertion and to start the finger search from there.

⟨*finger search from last insertion*⟩≡

```
  forall(k, L)
  { if (S.empty()) it = S.insert(k, 0);
    else
    { it = S.finger_locate(it,k);
      it = ( it ? S.insert_at(it,k,0,LEDA::before) :
                  S.insert_at(S.max_item(),k,0,LEDA::after) );
    }
  }
```

With this version of finger search each search takes constant time and hence a total running time of $O(n)$ results.

Table 5.7 shows the running times of our four versions of insertion sort in comparison to the built-in sorting routine for lists (*L.sort*( )) for $n = 500000$ and $f = 50$. We made the comparison for the key types *int*, *double*, and *four_tuple<int*, *int*, *int*, *int>* to study the influence of the cost of comparing two keys. The table shows that insertion sort with finger search is superior to repeated insertion sort for nearly sorted input sequences and that the advantage becomes larger (as is to be expected from the asymptotic analysis) as comparisons become more expensive. The table also shows that in the case of very expensive comparisons insertion sort with finger search can even compete with quicksort (which is the algorithm used in the sorting routine for lists).

It is worthwhile to take a more abstract view of the programs above. The less mathematically inclined reader may skip the next two paragraphs. Let $k_1, \ldots, k_n$ be a sequence of distinct keys from a linearly ordered type K. An *inversion* is a pair of keys that is not

	Repeated insertion	Finger search from rear	Finger search from both ends	Finger search from last insertion	List sort
int	5.45	4.78	4.7	2.98	2.22
double	6.28	5.1	7.12	3.28	2.53
quads	22.1	13.9	16.8	6.3	14.8

Table 5.7 Running times of the four versions of insertion sort and of the sorting routine $L.sort(\,)$ for lists for $n = 500000$ and $f = 50$. The sorting routine for lists uses quicksort with the middle element of the list as the splitting element. It runs in time $O(n \log n)$. Three different key types were used: *int*, *double*, and the type *four_tuple<int, int, int, int>* where an integer i was represented as the quadruple $(0, 0, 0, i)$. This ensures that comparisons between quadruples are expensive. You may perform your own experiments with the sortseq sort demo.

in ascending order, i.e., a pair (i, j) of indices with $1 \le i < j \le n$ and $k_i > k_j$. We use F to denote the total number of inversions and use f_j to denote the number of inversions involving j as their second component, i.e.,

$$f_j = |\{\, i \,;\, i < j \text{ and } k_i > k_j \,\}|$$

If F is zero then the sequence is already sorted. The maximal value of F is $n(n-1)/2$. We show that insertion sort with finger search from the rear runs in time $O(n(1 + \log(F/n)))$ on a sequence with F inversions. So the worst case is $O(n \log n)$, the best case is $O(n)$, and the running time degrades smoothly as F increases. A sequence with a "small" value of F is sometimes called *nearly sorted*. Thus, insertion sort with finger search is fast on nearly sorted sequences.

Assume that we have already sorted $k_1, \ldots, k_{j-1}$ and next want to insert k_j. As in our programs above we use S to denote the resulting sorted sequence. Each key in $k_1, \ldots, k_{j-1}$ which is larger than k_j causes an inversion and hence the number of keys in $k_1, \ldots, k_{j-1}$ larger than k_j is equal to f_j. Thus, k_j needs to be inserted at the f_j-th position from the rear end of S. A finger search from the rear end of S determines this position in time $O(\log f_j)$. We conclude that the total running time of insertion sort with finger search from the rear end is

$$O(\sum_{1 \le j \le n} 1 + \log f_j) = O(n + \log \prod_{1 \le j \le n} f_j).$$

Subject to the constraint $\sum_{1 \le j \le n} f_j = F$, the product $\prod_{1 \le j \le n} f_j$ is maximized if all f_j's are equal and hence are equal to F/n. The claimed time bound of $O(n \cdot (1 + \log(F/n)))$ follows.

Split: There are several operations to combine and split sequences. If S is a sorted sequence and *it* is an item of S then

```
S.split(it, T, U, dir)
```

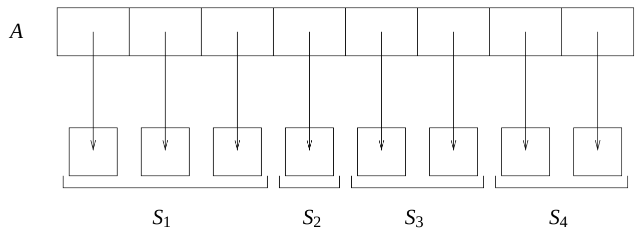


Figure 5.16 A sequence S of eight items that has been split into a sequence of length three, a sequence of length one, and two sequences of length two. The entry $A[i]$ of the array A contains a pointer to the i-th container of S. The sequences S_1, S_3, and S_4 need to be split further. In the sortseq_split program there will be a task in the task stack for each one of them. The task for S_3 has the form (pointer to $S_3, 4, 5$).

splits S after (if $dir = LEDA::after$) or before (if $dir = LEDA::before$) *it* and returns the two fragments in T and U. More precisely, if S is equal to

$$x_1, \ldots, x_{k-1}, it, x_{k+1}, \ldots, x_n$$

and *dir* is *LEDA*::*after* then $T = x_1, \ldots, x_{k-1}, it$ and $U = x_{k+1}, \ldots, x_n$ after the split. If *dir* is before then U starts with *it* after the split. The two sequences T and U must name distinct objects, but S may be one of T or U. If S is distinct from T and U then S is empty after the split. The running time of *split* is $O(\log n)$ for *_sortseqs* and is $O(1 + \log \min(k, n - k))$ for *sortseqs*.

We sketch an application of splitting in order to show the difference between the two time bounds. Assume that S is a sorted sequence of length n and consider the following process to split S into n sequences of length 1 each. We start with S and as long as we have a sequence of length larger than 2 we split this sequence at an arbitrary item.

In the following program we construct a sorted sequence S of n items and store its items in an array A. We also maintain a stack of "tasks". A task is a triple consisting of a pointer to a subsequence of S plus the indices of the first and the last item in the subsequence, see Figure 5.16. Initially there is only one task, namely, the triple $(S, 1, n)$. In each iteration of the loop we take the topmost task from the stack. If the sequence has less than two elements and hence requires no further split, we simply delete it. Otherwise, we split it at a random element and create tasks for the two parts. We continue until there are no tasks left.

⟨*sortseq_split*⟩≡
```
main(){
⟨sortseq split: read n⟩
typedef sortseq<int,int> int_seq;
array<seq_item> A(n);
int_seq* S = new int_seq();;
for (int i = 0; i < n; i++) A[i] = S->insert(i,0);
typedef three_tuple<int_seq*,int,int> task;
stack<task> TS;
TS.push(task(S,0,n-1));
float UT = used_time();
while ( !TS.empty() )
{ task t = TS.pop();
  int_seq* S = t.first();
  int l = t.second();
  int r = t.third();
  if ( r - l + 1 < 2 ) { delete S; continue; }
  int_seq* T = new int_seq();
  int_seq* U = new int_seq();
  int m = rand_int(l,r-1);
  S->split(A[m],*T,*U,LEDA::after);
  delete S;
  TS.push(task(T,l,m));
  TS.push(task(U,m+1,r));
}
⟨sortseq split: report time⟩
}
```

We show that the running time of this program is linear in n. We do so for arbitrary choice of the splitting index m and not only for random choice of m. The less mathematically inclined reader may skip the analysis. We use $T(n)$ to denote the maximal running time of the program on a sequence of n items. Then $T(1) = c$ and

$$T(n) \le \max_{1\le m<n} T(m) + T(n-m) + c(1 + \log\min(m, n-m))$$

for $n > 1$ and a suitable constant c. The recurrence relation reflects the fact that it takes time $c(1+\log\min(m, n-m))$ to split a sequence of length n into sequences of length m and $n-m$ and additional time $T(m)$ and $T(n-m)$ to split these sequences further into sequences of length 1. We need to take the maximum with respect to m since we are interested in the worst case time. We show $T(n) \le c(5n - 2 - 2\log(n+1))$ for all n by induction on n. This is certainly true for n equal to 1. So assume $n > 1$ and let m maximize the right-hand side in the recurrence relation above. Because of the symmetry of the right-hand side in m and $n-m$ we may assume $m \le n/2$. Then

$$\begin{aligned} T(n) &\le T(m) + T(n-m) + c(1 + \log\min(m, n-m)) \\ &\le c(5m - 2 - 2\log(m+1) + 5(n-m) - 2 - 2\log(n-m+1) + 1 + \log m) \\ &< c(5n - 2 - \log(m+1) - 2\log(n-m+1) - 1) \\ &\le c(5n - 2 - 2\log(n+1)), \end{aligned}$$

where the first inequality is our recurrence relation, the second inequality follows from the induction hypothesis, the third inequality is simple arithmetic, and the last inequality follows from the fact that $1 + \log(m+1) + 2\log(n-m+1) \ge 2\log(n+1)$ for all m with $1 \le m \le n/2$. To see this, observe first that the second derivative of $f(m) = 1 + \log(m+1) + 2\log(n-m+1)$ is negative and hence $\min_{1\le m\le n/2} f(m) = \min(f(1), f(n/2))$.

Observe next that $f(1) \geq 2\log(n+1)$ and $f(n/2) \geq 2\log(n+1)$. This completes the induction.

Concatenation and Merging: We turn to concatenation and merging of sequences.

```
S.conc(T,dir)
```

appends T to the rear (if $dir = LEDA::after$) or front (if $dir = LEDA::before$) of S and makes T empty. Of course, we may apply *conc* with $dir = LEDA::after$ only if the key of the last item in S is smaller than the key of the first item in T and with $dir = LEDA::before$ only if the key of the last item in T is smaller than the key of the first item in S. The running time of *conc* is $O(\log(n+m))$ for *_sortseqs* and is $O(1+\log\min(n,m))$ for *sortseqs* where n and m are the lengths of the sequences to be concatenated. *Merge* generalizes *conc*.

```
S.merge(T)
```

merges the list T into the list S and makes T empty. For example, if $S = \langle 5,.\rangle\ \langle 7,.\rangle\ \langle 8,.\rangle$ and $T = \langle 6,.\rangle\ \langle 9,.\rangle$ are sequences with key type *int* then $S = \langle 5,.\rangle\ \langle 6,.\rangle\ \langle 7,.\rangle\ \langle 8,.\rangle\ \langle 9.,\rangle$ after the merge. Of course, S and T can only be merged if the keys of all items are distinct. The time to merge two sequences of lengths n and m, respectively, is $O(\log\binom{n+m}{n})$; *merge* is only supported by *sortseqs*.

We sketch how *merge* is implemented, we compare *merge* with two less sophisticated approaches to merging, and we show how to use *merge* in a robust version of merge sort. We start with a sketch of the implementation. Assume that the sequences S and T are to be merged and that the number of elements in T is at most the number of elements in S. We insert the elements of T one by one into S, starting with the first element of T. In order to locate the position of an element of T in S we use a finger search starting from the position of the last insertion (starting from the first element of S instead of the first element of T).

```
sortseq_item finger = S.min_item();
sortseq_item     it = T.min_item();
while ( it )
{ finger = S.finger_locate(finger,T.key(it));
  S.insert_at_item(finger,T.key(it),T.inf(it));
  it = T.succ(it);
}
```

The running time of this program is easy to analyze. We use m to denote the number of elements in T and n to denote the number of elements in S. Assume that the i-th element of T is to be inserted after the f_i-th element of S for all i with $1 \leq i \leq m$. Set $f_0 = 0$. The finger search that determines the position of the i-th element of T in S takes time $O(\log d_i)$ where $d_i = f_i - f_{i-1}$ is the number of elements of S that are between the position of insertion for the i-th and the $(i-1)$-th element. Clearly, $\sum_i d_i \leq n$. The total time for merging T into S is

$$\sum_i O(1+\log d_i) = O(m + \log\prod_i d_i).$$

Subject to the constraint $\sum_i d_i \le n$, the product $\prod_i d_i$ is maximal if all d_i are equal to n/m. The running time is therefore $O(m + mlog(n/m)) = O(\log \binom{n+m}{n})$. To see the last equality observe first that

$$1 + \log(n/m) = 1 + \log(n+m)/m \le 2\log((n+m)/m)$$

since $n + m \ge 2m$ and observe next that $m\log((n+m)/m) = \log((n+m)/m)^m$ and $((n+m)/m)^m \le \binom{n+m}{m}$.

We next compare *merge* to two less sophisticated merge routines. Let T and U be sorted sequences of length n and m, respectively. There are two ways to merge U into T that come to mind immediately. The first method inserts the elements of U one by one into T. This takes time $O(m\log(n+m))$. The second method scans both files simultaneously from front to rear and inserts the elements of U as they are encountered during the scan. This takes time $O(n+m)$. In the following programs we assume that T and U are of type *sortseq*<K, *int*>.

⟨*three merging routines*⟩≡

```
template < class K >
void merging_by_repeated_insertion(sortseq<K,int>& T, sortseq<K,int>& U)
{ seq_item it = U.min_item();
  while ( it )
  { T.insert(U.key(it),U.inf(it));
    it = U.succ(it);
  }
}

template < class K >
void merging_by_scanning(sortseq<K,int>& T, sortseq<K,int>& U)
{ seq_item it1 = T.min_item();
  seq_item it2 = U.min_item();

  while ( it2 && compare(U.key(it2),T.key(it1)) < 0 )
  { T.insert_at(it1,U.key(it2),U.inf(it2),LEDA::before);
    it2 = U.succ(it2);
  }
  seq_item succ1 = T.succ(it1);
  while ( it2 )
  { K k2 = U.key(it2);
    while ( succ1 && compare(T.key(succ1),k2) < 0 )
    { it1 = succ1;
      succ1 = T.succ(succ1);
    }
    it1 = T.insert_at(it1,k2,U.inf(it2),LEDA::after);
    it2 = U.succ(it2);
  }
}

template < class K >
void merging_by_finger_search(sortseq<K,int>& T, sortseq<K,int>& U)
{ T.merge(U); }
```

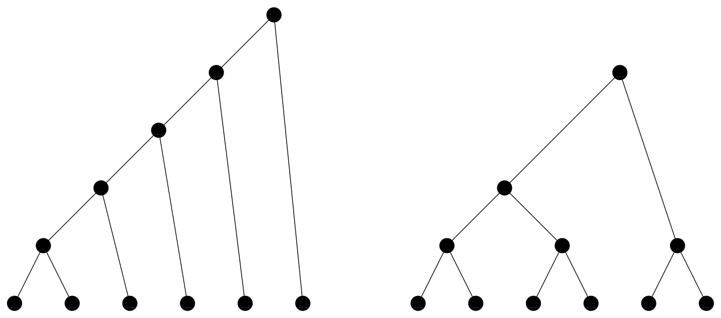


Figure 5.17 Two patterns for merging six sequences of length one. The merge pattern on the left is unbalanced: it first merges two sequences of length one, then merges the resulting sequence of length two with a sequence of length one, then merges the resulting sequence of length three with a sequence of length one, ... The second merge pattern is balanced: it first forms three sequences of length two, then merges two of them to a sequence of length four, and finally merges the sequence of length four with the remaining sequence of length two.

How do the three routines compare theoretically and experimentally? Let us consider three cases: $m = 1$, $m = n$, and $m = n/\log n$. Merging by repeated insertion takes time $O(\log n)$, $O(n \log n)$, and $O(n)$, respectively, merging by scanning takes $O(n)$ in all three cases, and merging based on finger search takes time $O(\log n)$, $O(n)$, and $O(m \log(n/m)) = O(n \operatorname{loglog} n/\log n)$, respectively. We see that merging based on finger search is never worse than the two other methods (it has a larger constant of proportionality, though) and that it is superior to both methods in two of the cases. Table 5.8 shows an experimental comparison of the three methods.

Robust Merge Sort: We use our three merging routines in a version of merge sort. In order to sort a set of n elements, merge sort starts with n sequences of length 1 (which are trivially sorted) and then uses merging to combine them into a single sorted sequence of length n. The *merge pattern*, i.e., the way in which the n sequences are combined into a single sequence can be visualized by a binary tree with n leaves and $n - 1$ internal nodes. The n leaves correspond to the n initial sequences and each internal node corresponds to a merging operation. In this way we associate with every internal node the sorted sequence that results from merging the two sequences associated with its children. Figure 5.17 shows two merging patterns.

How do our three merging routines behave? In the balanced merging pattern we perform about $n/2^k$ merges between sequences having length 2^k each and hence obtain a total running time of

$$O\Big(\sum_{0 \le k < \log n} (n/2^k) M(2^k, 2^k)\Big),$$

where $M(x, y)$ is the time to merge two sequences of length x and y. For merging by

		Merging by		
		repeated insertion	scanning	finger search
int	$m = 1$	0	0.75	0
	$m = 10$	0	0.767	0
	$m = 100$	0	0.7	0
	$m = 1000$	0.0333	0.767	0.05
	$m = 10000$	0.4	0.883	0.267
	$m = 100000$	3.65	1.78	1.75
double	$m = 1$	0	0.817	0
	$m = 10$	0	0.8	0
	$m = 100$	0.0167	0.817	0.0167
	$m = 1000$	0.05	0.833	0.0333
	$m = 10000$	0.433	0.95	0.317
	$m = 100000$	4.2	2.02	2.02
quadruple	$m = 1$	0	2.58	0
	$m = 10$	0	2.6	0
	$m = 100$	0.0167	2.67	0.0333
	$m = 1000$	0.183	2.63	0.15
	$m = 10000$	1.65	2.82	1.03
	$m = 100000$	15.8	4.38	6.6

Table 5.8 Running times of the three versions of merging for $n = 500000$ and different values of m. The sequence T consisted of the first n even integers and the sequence U consisted of the integers $2(n/m)i + 1$ for $i = 1, \ldots, m$. Three different key types were used: *int*, *double*, and the type *four_tuple<int, int, int, int>* where an integer i was represented as the quadruple $(0, 0, 0, i)$. This ensures that comparisons between quadruples are expensive. You may perform your own experiments with the sortseq merge demo.

repeated insertion we have $M(x, x) = O(x \log x)$ and hence obtain a total running time of

$$O(\sum_{0 \le k < \log n} (n/2^k)2^k k) = O(n \sum_{0 \le k < \log n} k) = O(n \log^2 n).$$

For merging by scanning and merging by finger search we have $M(x, x) = O(x)$ and hence

	Unbalanced merge tree	Balanced merge tree
merging by repeated insertion	5.07	11.5
merging by scanning	3.44e+03	9.82
merging by finger search	5.9	8.73

Table 5.9 This table was generated by program sortseq_merge_sort. You can perform your own experiments with the sortseq merge demo. Merging by finger search comes in shortly after the winner for both merge patterns.

obtain a total running time of

$$O(\sum_{0\le k<\log n}(n/2^k)2^k) = O(n\log n).$$

We conclude that the latter two merging methods perform optimally in the case of a balanced merging pattern but that merging by repeated insertion does not.

Let us turn to the unbalanced merging pattern. It builds a sequence of length i by merging a sequence of length $i-1$ and a sequence of length 1 for all i, $2 \le i \le n$. We obtain a total running time of

$$O(\sum_{2\le i\le n} M(i,1)).$$

For merging by repeated insertion and merging by finger search we have $M(x,1) = O(\log x)$ and hence obtain a total running time of

$$O(\sum_{2\le i\le n}\log i) = O(n\log n).$$

For merging by scanning we have $M(x,1) = O(x)$ and hence obtain a total running time of

$$O(\sum_{2\le i\le n} i) = O(n^2).$$

We conclude that the two former merging methods perform optimally in the case of an unbalanced merging pattern but that merging by scanning does not. *Only merging by finger searching performs optimally for both merge patterns*.

Table 5.9 shows an experimental comparison. You may perform your own experiments by calling the sortseq merge demo. This program generates n sorted sequences of length one and puts pointers to them into an array A (*intseq* is an abbreviation for *sortseq<int, int>*.). It permutes A to make sorting non-trivial.

⟨*fill A*⟩≡

```
for (i = 0; i < n; i++)
{ A[i] = new int_seq;
  A[i]->insert(i,0);
}
A.permute();
```

It then uses either the unbalanced merge pattern or the balanced merge pattern to merge the n sequences into a single sequence (*merge* is any one of our three merging routines).

⟨*unbalanced merge pattern*⟩≡

```
for (i = 1; i < n; i++)
{ merge(*A[0],*A[i]);
  delete A[i];
}
```

⟨*balanced merge pattern*⟩≡

```
while (n > 1)
{ int k = n/2;
  for (i = 0; i < k; i++)
  { merge(*A[i],*A[k + i]);
    delete A[k+i];
  }
  if ( 2 * k < n ) // n is odd
    { A[k] = A[n - 1]; n = k + 1; }
  else
    { n = k; }
}
```

We close our discussion of merging by showing that merge sort with merging by finger search has running time $O(n \log n)$ for every merge pattern. Recall that a merge pattern is a binary tree T with n leaves and that every internal node of T corresponds to a merge operation. For an internal node let $s(v)$ be the length of the sorted sequence that is the result of the merge operation at node v and for a leaf v let $s(v)$ be equal to one. With this notation the cost of the merge at a node v with children x and y is

$$O(\log \binom{s(v)}{s(x)}) = O(\log(s(v)!/(s(x)!s(y)!))) = O(\log s(v)! - \log s(x)! - \log s(y)!)$$

and the total running time of merge sort is obtained by summing this expression over all nodes v of T. In this sum every node z except for the root and the leaves contributes twice: it contributes $\log s(z)!$ when z is considered as a parent and it contributes $-\log s(z)!$ when z is considered as a child. The two contributions cancel. Therefore everything that remains is the contribution of the root (which is $\log n!$) and the contribution of the leaves (which is $-n \log 1$). We conclude that the total running time is $O(n \log n)$ independent of the merge pattern T.

Operations on Subsequences: We want to mention two further operations. Let a and b be two items in a sorted sequence S with a being equal to or before b. Then

```
S.reverse_items(a,b)
```

reverses the subsequence of items in S starting at a and ending at b, i.e., if

$$S = it_1, it_2, \ldots, it_{i-1}, it_i, it_{i+1}, \ldots, it_{j-1}, it_j, it_{j+1}, \ldots, it_n$$

before the operation and $a = it_i$ and $b = it_j$ then

$$S = it_1, it_2, \ldots, it_{i-1}, it_j, it_{j-1}, \ldots, it_{i+1}, it_i, it_{j+1}, \ldots, it_n$$

after the operation. We will see an application of *reverse_items* in a plane sweep algorithm for segment intersection in Section 10.7.2. *Reverse_items* runs in time proportional to the number of items that are reversed. *Reverse_items* is also available under the name *flip_items*.

The operation

```
S.delete_subsequence(a,b,T)
```

removes the subsequence starting at a and ending at b from S and assigns it to T. The running time is $O(\log \min(m, n - m))$ where n is the number of items in *IT* and m is the number of items that are removed. We will see an application of *delete_subsequence* in Section 5.8 on Jordan sorting.

Sequences and Items: Many of the operations on *sortseqs* take items as arguments, e.g.,

```
S.finger_locate(finger,x)
```

locates x in S by searching from the item *finger*. What happens if *finger* is not an item in S but in some other *sortseq IT*?

The complete specification of *finger_locate* is as follows (and this is, of course, the specification that is given in the manual). Let *IT* be the sorted sequence containing *finger*. Then

```
S.finger_locate(finger,x)
```

is equivalent to

```
IT.finger_locate(finger,x)
```

provided that *IT* has the same type as S. If *IT* and S have different types the semantics of *S.finger_locate*(*finger*, x) is undefined.

A similar statement holds for all other operations having items as arguments. So

```
S.reverse_items(a,b)
```

is applied to the sequence containing the items a and b (of course, a and b must belong to the same sequence).

If the items determine the sequence to which the operation is applied, why does one have to specify a sequence at all? We explored the alternative to make *finger_locate* a static member function of *sortseq*<K, I> and to write

```
sortseq<K,I>::finger_locate(finger,x);
```

We decided against it because in most applications of sorted sequences there is no problem in providing the sequence as an argument and in these situations it is clearer if the sequence is provided as an argument. The price to pay is that in the rare situation where the sequence is not known (the program in Section 5.8 is the only program we have ever written where this happens) one has to “invent” S, i.e., to declare a dummy sequence S and to apply *finger_locate* to it.

Exercise for 5.6

1 A *run* in a sequence of keys is a sorted subsequence. Let $L = k_1, \ldots, k_n$ be any sequence and let k be the number of runs in L, i.e., k is one larger than the number of i with $k_i > k_{i+1}$. Show that insertion sort with finger search from the position of the last insertion sorts a sequence consisting of k runs in time $O(n(1 + \log k))$.

5.7 The Implementation of Sorted Sequences by Skiplists

We first desribe the skiplist data structure. Skiplists were invented by W. Pugh [Pug90a, Pug90b] and our implementation is based on his papers. We go beyond his papers by also providing implementations for finger searches, merging, and deletion of subsequences. We start with an overview of the data structure and then outline the content of the files skiplist.h, _skiplist.c, and sortseq.h. In the bulk of the section we give the implementations of the different operations on skiplists.

5.7.1 *The Skiplist Data Structure*

A skiplist is a sequence of *skiplist_nodes*, see Figure 5.18. We also say *tower* instead of *skiplist_node*. In a skiplist for a sequence of n elements we have $n + 2$ towers, n towers corresponding to the elements of the sequence and two towers called *header* and *STOP* that serve as sentinels. We refer to the former towers as *proper* and to the latter as *improper*.

A tower contains the following information:

— a *key*,
— an *inf*ormation,
— an integer *height*,
— an array *forward* of *height* + 1 pointers to towers,
— a *backward* pointer, and
— a *pred*ecessor pointer.

The keys of the proper towers in a skiplist are strictly increasing from front to rear of the sequence. The sentinels *header* and *STOP* have no keys stored in them although, logically, their keys are $-\infty$ and ∞, respectively. It would make life somewhat easier if the key type K provided the elements $-\infty$ and ∞. Because not all key types do, we have decided to

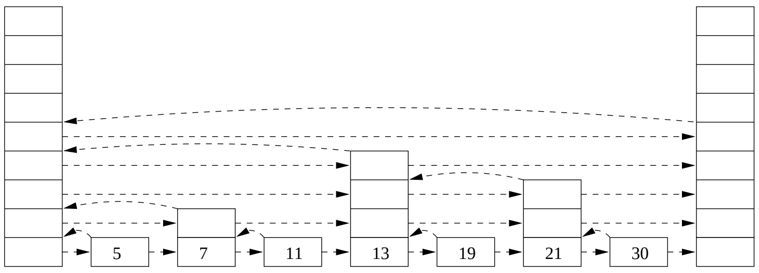


Figure 5.18 A skiplist: The sequence of keys stored in the sequence is 5, 7, 11, 13, 19, 21, 30. The proper towers have height 0, 1, 0, 3, 0, 2, and 0, respectively. Their keys are shown at the bottom of the towers. The two improper towers *header* and *STOP* are the first and last tower, respectively. They have no keys. The forward pointers point horizontally to the right. The backward pointers are shown as curved arcs and the predecessor pointers are not shown. All forward pointers that have no proper tower to point to, point to *STOP*. An object of type skiplist contains pointers to *header* and *STOP*. The header points back to the skiplist object.
A search for 19 proceeds as follows. We start in the header and consider the forward pointer at height 3 (= maximal height of a proper tower) out of the header. It ends in a tower with key 13. Since $19 > 13$ we move forward to the tower with key 13 and consider its forward pointer at height 3. It ends in *STOP* (which has key ∞) and so we drop down to height two. The forward pointer at height 2 out the tower with key 13 ends in the tower with key 21. Since $19 < 21$ we drop down to the height one, … .

store no keys in the sentinels. When formulating invariants we will however assume that the keys of *header* and *STOP* are $-\infty$ and ∞, respectively.

Skiplists represent the sequence stored at different levels of granularity. The tower of height at least zero represent the entire sequence, the towers of height at least one represent a subsequence, the towers of height at least two represent a subsequence of the subsequence, … . The operations on Skiplists gain their efficiency by exploiting the different levels of granularity; Figure 5.18 sketches a search for key 19 in our example skip list. Observe that the search first locates 19 with respect to the list represented by the towers of height at least 3, i.e., the list $(-\infty, 13, +\infty)$, then with respect to the list represented by the towers of height at least 2, i.e., the list $(-\infty, 7, 13, 21, +\infty)$, … .

The height of a proper tower is chosen probabilistically when the tower is created. We will explain this in more detail below. The height of a proper tower is always non-negative. The height of *STOP* is -1 and the height of *header* is equal to *MaxHeight*. We set *MaxHeight* to 32 in our implementation. When we choose the heights of proper towers we will make sure that their height is smaller than *MaxHeight*. The sentinels *header* and *STOP* can therefore be recognized by their height. *Headers* are the only items with height equal to *MaxHeight* and *STOP* nodes are the only items with negative height.

A *header* stores information in addition to the ones listed above: the data member *true_height* is one plus the maximal height of any proper tower (it is zero if there are no proper towers) and the member *myseq* stores a pointer to the skiplist to which *header* belongs. The *header* has type *header_node*, where a *header_node* is an *skiplist_node* with the two additional fields just mentioned.

A pointer to a *skiplist_node* is called an *sl_item* and a pointer to a *header_node* is called a *large_item*.

In the definitions below the flag *SMEM* (= simple memory management) allows us to choose between two schemes for memory allocation. If *SMEM* is defined, the obvious memory allocation scheme is used and *forward* is realized as an array of *sl_items* and if *SMEM* is not defined, a refined and more efficient memory allocation scheme is used. This is explained in more detail in Section 5.7.4.

The flag `__exportC` is used for preprocessing purposes. On UNIX-systems it is simply deleted and on Windows-systems it is replaced by flags which are needed to generate dynamic libraries.

⟨*definition of classes skiplist_node and header_node*⟩≡

```
class __exportC header_node;
class __exportC skiplist_node;

typedef skiplist_node* sl_item;
typedef header_node*     large_item;

const int MaxHeight = 32;

class __exportC skiplist_node
{ friend class __exportC skiplist;

  static leda_mutex mutex_id_count;
  static unsigned long id_count;

  GenPtr key;
  GenPtr inf;
  int    height;
  unsigned long id;           // id number
  sl_item pred;
  sl_item backward;
#ifdef SMEM
  sl_item* forward;  // array of forward pointers
#else
  sl_item forward[1];
#endif

  friend unsigned long ID_Number(skiplist_node* p){return p->id;}
};

class __exportC header_node : public skiplist_node
{ friend class __exportC skiplist;
#ifndef SMEM
  sl_item more_forward_pointers[MaxHeight];
#endif
  int true_height;
  skiplist* myseq;
};
```

A header node can be viewed as a *skiplist_node* and as a *header_node*. If v is an *sl_item* which is known to be a *large_item* (because $v \to$ *height* $=$ *MaxHeight*) then we can cast v to a large item by (*large_item*)v and access the skiplist containing v by ((*large_item*)v) $\to$ *myseq*.

We can now complete the definition of the skiplist data structure by defining the values

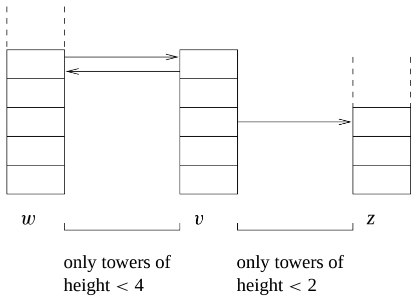


Figure 5.19 Forward and backward pointers: $v \to$ *forward*[2] points to the closest successor tower of height at least 2 and $v \to$ *backward* points to the closest predecessor tower of height at least 4.

of the various pointers stored in a tower, see Figure 5.19. Let v be any tower and let h be the height of v (view *header* as a tower of height *true_height* for this paragraph). Then:

- for all i, $0 \le i \le h$, the i-th forward pointer of v points to closest successor tower of height at least i (to *STOP* if there is no such tower),
- the backward pointer points to the node w with the highest forward pointer into v, i.e., the h-th forward pointer of w points to v,
- and the predecessor pointer of v points to the tower immediately preceding v.

The procedure *validate_data_structure* checks the invariants in time $O(\mathit{true\_height} \cdot n)$.

⟨*miscellaneous*⟩≡

```
void skiplist::validate_data_structure()
{ assert(header == header->myseq->header);
  assert (header->height == MaxHeight);
  assert(STOP->height == -1);

  int max_proper_height = -1;
  sl_item p = (sl_item) header;

  while (p != STOP)
  { assert(p->height >= 0);
    if (p != header && p->height > max_proper_height)
                         max_proper_height = p->height;
    p = p->forward[0];
  }
  assert(header->true_height == max_proper_height + 1);

  p = (sl_item) header;
  while (p != STOP)
  { sl_item q = p->forward[0];
    assert(p == q->pred);                  //condition three

    if (p != header && q != STOP)          //check order
                        assert(cmp(p->key,q->key) < 0);

    for(int h=0; h<=Min(p->height,header->true_height);h++)
    { sl_item r = p->forward[0];
```

```
      while (r->height < h && r != STOP) r = r->forward[0];
      assert ( r == p->forward[h]);           //condition one
      if ( h == r->height ) assert(r->backward == p);
    }                                         //condition two
    p = q;
  }
  assert(STOP->backward == (sl_item) header);
}
```

As a preview for later sections we describe briefly how one can search for a key x in a skiplist. We keep a node v and a height h such that $v \rightarrow key < x$ and $x \le v \rightarrow forward[h] \rightarrow key$. Initially, $v = header$ and $h = true\_height$. In the basic search step we find a node v with the same property and h one less. This is easy to achieve. We only have to start a walk at node v taking forward pointers at height $h - 1$.

```
h--;
w = v->forward[h];
while (key > w->key)
{ v = w;
  w = v->forward[h];
}
```

The while-loop re-establishes the invariant $v \rightarrow key < x \le v \rightarrow forward[h] \rightarrow key$. Continuing in this way down to $h = 0$ locates x among the items in the skiplist. The complete program for a search in a skiplist is therefore as follows:

```
sl_item v = header;
int h = header->true_height;
while ( h > 0 )
{ h--;
  w = v->forward[h];
  while (w != STOP && key > w->key)
  { v = w;
    w = v->forward[h];
  }
}
```

The search in skiplists is efficient because skiplists represent the underylying sequence at different levels of granularity. The forward pointers at level 0 represent the entire sequence, the forward pointers at level 1 represent the subsequence formed by the towers of height at least 1, the forward pointers at level 2 represent the subsequence formed by the towers of height at least 2, ... In a search we locate x with respect to the subsequence of towers of height at least h for decreasing values of h. This is trivial at the highest level and requires only little additional work for each smaller value of h.

The height of a proper tower is chosen probabilistically when the tower is created. It is set to h with probability $p^h(1 - p)$ where p with $0 < p < 1$ is a parameter that is fixed when the skiplist is created. In our implementation we use $1/4$ as the default value for p. We draw three easy consequences from this probabilistic definition of height.

The probability that a proper tower has height h or more is $\sum_{k\ge h} p^k(1-p) = p^h$ and therefore the expected value of *height* can be computed as[15]:

$$\mathrm{E}[\mathit{height}] = \sum_{h\ge 1} p^h = p \sum_{h\ge 0} p^h = p/(1-p).$$

Since the space requirement for a tower of height h is $(6+h)\cdot 4$ bytes plus the space for the key and the information we conclude that the expected space requirement for a skiplist of n items is about $(6+p/(1-p))4n$ bytes plus the space for the keys and informations. For $p = 1/4$ we have $\mathrm{E}[h] = 1/3$ and hence the expected space requirement for a skiplist of n items is about $76/3n = 25.333n$ bytes. The refined memory allocation scheme needs a bit more, see Section 5.7.4.

The fact that p^h is the probability that a proper tower has height h or more implies that the probability that some tower in a collection of n proper towers has height h or more is at most $\min(1, np^h)$. This is one for $h \le \log_{1/p} n$ and is at most p^l for $h = \lceil \log_{1/p} n\rceil + l$. Since *true_height* is one plus the maximal height of any proper tower, we can compute the expected value of *true_height* as:

$$\begin{aligned}
\mathrm{E}[\mathit{true\_height}] &= \sum_{h\ge 1} \mathrm{prob}(\mathit{true\_height} \ge h) = \sum_{h\ge 1} \mathrm{prob}(\text{maximal height} \ge h-1)\\
&\le \sum_{h\ge 0} \min(1, p^h) \le \sum_{0\le h<\lceil \log_{1/p} n\rceil} 1 + \sum_{h\ge \lceil \log_{1/p} n\rceil} p^h\\
&\le 1 + \log_{1/p} n + \sum_{l\ge 0} p^l = 1 + \log_{1/p} n + 1/(1-p).
\end{aligned}$$

Finally, if v is any tower then the probability that $v \to \mathit{backward}$ has height larger than v is p. Observe that $v \to \mathit{backward}$ has at least the height of v and that the conditional probability that a tower has height $h+1$ or more given that it has height h or more is $p^{h+1}/p^h = p$. Thus, the probability that $v \to \mathit{backward}$ has height larger than v is p.

We use this observation to bound the cost of a search. Consider a search for a key x and let $v_0, v_1, \ldots, v_k$ be the path traced by the variable v in the program above. Then $v_0 = \mathit{header}$ and $v_i = v_{i+1}{\to}\mathit{backward}$. By the above, the probability that the height of v_i is larger than the height of v_{i+1} is p and hence the expected number of nodes traversed at any particular height is $1/p$. We start at height zero and end at height *true_height*. The expected length of the path is therefore bounded by

$$1/p \cdot (1 + \log_{1/p} n + 1/(1-p)).$$

This concludes our discussion of skiplist nodes.

We turn to the class representing skiplists. In an *skiplist* we store the items *header* and *STOP* and some quantities related to the random process: *prob* contains the parameter p in use, and *randomBits* contains an integer whose last *randomsLeft* bits are random. We use

[15] If X is a random variable which assumes non-negative integer values and $q_h = \mathrm{prob}(X \ge h)$ and $p_h = \mathrm{prob}(X = h)$ for all $h \ge 0$ then $\mathrm{E}[X] = \sum_{h\ge 0} p_h \cdot h = \sum_{h\ge 1} p_h \cdot h = \sum_{h\ge 1}(q_h - q_{h+1})\cdot h = \sum_{h\ge 1} q_h$.

randomBits as the random source in the construction of skiplist nodes. Whenever all bits in *randomBits* are used up we refill it using the LEDA random number generator.

⟨*data members of class skiplist*⟩≡

```
large_item  header;
sl_item STOP;
float prob;
int randomBits;
int randomsLeft;
```

⟨*private member functions of class skiplist*⟩≡

```
void fill_random_source()
{ randomBits = rand_int(0,MAXINT-1);
  randomsLeft = 31;
}
```

5.7.2 *The Files sortseq.h, skiplist.h, and _skiplist.c*

The definition of type *sortseq*$<K, I>$ follows the strategy laid out in Section 13.4. We define two classes: an abstract data type class *sortseq*$<K, I>$ and an implementation class *skiplist*. The class *sortseq*$<K, I>$ is a parameterized class with type parameters K and I. The keys and infs in the implementation class are generic pointers.

The implementation class is defined in incl/LEDA/impl/skiplist.h and src/dict/_skiplist.c. We have already seen the chunks ⟨*definition of classes skiplist_node and header_node*⟩ and ⟨*data members of class skiplist*⟩. In the other chunks of skiplist.h we define a set of virtual functions that are later redefined in the abstract data type class and we define the functions that realize all operations on sorted sequences. The virtual functions are discussed in Section 5.7.3 and the other functions are discussed starting in Sections 5.7.5. In _skiplist.c we assemble the implementations of all member functions (except for the trivial ones which are given directly in the header file).

The compile-time constant SMEM is explained in Section 5.7.4.

⟨*skiplist.h*⟩≡

```
#ifndef SKIPLIST_H
#define SKIPLIST_H
// #define SMEM  remove comment for use of simple memory scheme
#include <LEDA/basic.h>
#include <assert.h>
⟨definition of classes skiplist_node and header_node⟩
class __exportC skiplist
{ ⟨data members of class skiplist⟩
  ⟨virtual functions of class skiplist⟩
  ⟨private member functions of class skiplist⟩
public:
  ⟨public member functions of class skiplist⟩
};
```

⟨*implementation of inline functions*⟩
```
#endif
```

⟨*_skiplist.c*⟩≡
```
#include <LEDA/impl/skiplist.h>
```
⟨*memory management*⟩
⟨*constructors and related functions*⟩;
⟨*search functions*⟩;
⟨*insert and delete functions*⟩;
⟨*concatenate and related functions*⟩;
⟨*miscellaneous*⟩;

The abstract data type class is derived from the implementation class (which we rename as IMPL to save ink) and an *seq_item* is nothing but an *sl_item*. The definition of *sortseq*$<K, I>$ has two large sections: in ⟨*redefinition of virtual functions*⟩ all virtual functions of the implementation class are redefined (see Section 5.7.3) and in ⟨*public member functions of sortseq*⟩ all operations on sorted sequences are defined by calling the corresponding function of the implementation class (see Section 5.7.10).

⟨*sortseq.h*⟩≡
```
#ifndef SORTSEQ_H
#define SORTSEQ_H
#if !defined(LEDA_ROOT_INCL_ID)
#define LEDA_ROOT_INCL_ID 360010
#include <LEDA/REDEFINE_NAMES.h>
#endif
#include <LEDA/basic.h>
#include <LEDA/impl/skiplist.h>
#define IMPL skiplist
typedef sl_item seq_item;
template<class K, class I>
class sortseq : public virtual IMPL {
```
⟨*redefinition of virtual functions*⟩
```
public:
```
⟨*public member functions of sortseq*⟩
```
};
#if LEDA_ROOT_INCL_ID == 360010
#undef LEDA_ROOT_INCL_ID
#include <LEDA/UNDEFINE_NAMES.h>
#endif
#endif
```

5.7.3 ***Virtual Functions and their Redefinition***

The class *skiplist* has virtual functions *cmp*, *clear_key*, *clear_inf*, *copy_key*, *copy_inf*, *print_key*, *print_inf* and *key_type_id*. All of them are redefined in *sortseq*$<K, I>$.

⟨*virtual functions of class skiplist*⟩≡

```
virtual int cmp(GenPtr x, GenPtr y) const
{ error_handler(1,"cmp should never be called"); return 0; }
virtual void copy_key(GenPtr&)  const  {  }
virtual void copy_inf(GenPtr&)  const  {  }
virtual void clear_key(GenPtr&) const
{ error_handler(1,"clear_key should never be called"); }
virtual void clear_inf(GenPtr&) const
{ error_handler(1,"clear_inf should never be called"); }
virtual void print_key(GenPtr)  const
{ error_handler(1,"print_key should never be called"); }
virtual void print_inf(GenPtr)  const
{ error_handler(1,"print_inf should never be called"); }
virtual int key_type_id() const
{ error_handler(1,"key_type_id should never be called");
  return 0;
}
```

⟨*redefinition of virtual functions*⟩≡

```
leda_cmp_base<K> cmp_def;
const leda_cmp_base<K> *cmp_ptr;
int cmp (GenPtr x, GenPtr y) const
{ return (*cmp_ptr) (LEDA_CONST_ACCESS(K,x), LEDA_CONST_ACCESS(K,y)); }
int ktype_id;
int key_type_id () const { return ktype_id; }
void clear_key(GenPtr& x) const  { LEDA_CLEAR(K,x); }
void clear_inf(GenPtr& x) const  { LEDA_CLEAR(I,x); }
void copy_key(GenPtr& x)  const  { LEDA_COPY(K,x);  }
void copy_inf(GenPtr& x)  const  { LEDA_COPY(I,x);  }
void print_key(GenPtr x)  const  { LEDA_PRINT(K,x,cout);  }
void print_inf(GenPtr x)  const  { LEDA_PRINT(I,x,cout);  }
```

What are these virtual functions good for? The implementation class uses them to manipulate keys and information fields. It calls *cmp* to compare two keys, it calls *copy_key*, *clear_key*, or *print_key* to copy, destroy or print a key (and analogously an inf), respectively, and it calls *key_type_id* to determine the kind of the key type (integer, double, or otherwise). The latter function allows us to optimize the treatment of integer and double keys. Keys and informations are stored as generic pointers in the implementation class and only the abstract class knows K and I. All virtual functions are redefined in the abstract class. For example, *cmp*(x, y) is redefined as *LEDA_COMPARE*(K, x, y) which in turn amounts to converting x and y to type K and then calling the compare function of type K. Similar statements hold for the other virtual functions, see Section 13.4.

Except for *copy_key* and *copy_inf* the virtual functions are only called in their redefined form. In order to double-check we have included appropriate asserts into the bodies of the virtual functions. *Copy_key* and *clear_key* are also called by the copy-constructor of *skiplist*

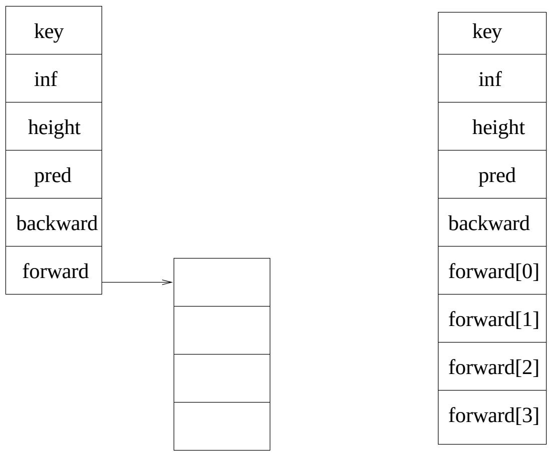


Figure 5.20 A skiplist node with four forward pointers. The left part shows the simple memory management scheme and the right part shows the refined memory management scheme.

and their original versions are used there. For this reason the original versions of *copy_key* and *copy_inf* are defined as functions with no effect.

5.7.4 *Memory Management*

We implemented two schemes for memory management: a simple scheme and a refined scheme. The refined scheme increases the speed of our implementation by almost a factor of two (if insertions and deletions have about the same frequency as lookups). The simple scheme can be selected by defining the constant `SMEM` in skiplist.h. Both schemes are illustrated by Figure 5.20.

In the simple scheme we construct an array of $h + 1$ forward pointers by

```
forward = new sl_item[h+1];
```

This calls the built-in new function and does not use LEDA's memory manager. An access to a forward pointer goes through a level of indirection as shown in Figure 5.20. The refined scheme avoids this level of indirection.

In the refined scheme we observe that the space required for a tower of height h is the size of an *skiplist_node* plus h times the size of a pointer. Recall that a node has already room for one forward pointer and that a tower of height h has $h + 1$ forward pointers. This suggests using the LEDA memory manager to allocate

$$int(sizeof(skiplist\_node)) + (h) * int(sizeof(skiplist\_node*))$$

bytes for a node of height h. Since C++ does not check array bounds and *forward* is the last field in *skiplist_node* this is equivalent to allocating space for the data member of an *skiplist_node* and an array *forward* of $h + 1$ pointers.

The scheme just described has the disadvantage that it leads to *true_height* different node sizes. The life of the LEDA memory manager becomes simpler if the number of different node sizes is small. We therefore modify the scheme slightly and round h to the next power of two if $h > 2$. We show that this modification uses very little additional space. The modified scheme never allocates more than twice the number of forward pointers that are actually needed and it allocates no additional forward pointer if $h \leq 2$. Since p^h is the probability that a tower has height h or more, the additional number of forward pointers per tower required by the modified scheme is therefore bounded by $\sum_{h \geq 3} p^h = p^3/(1-p)$. For $p = 1/4$ this is equal to 1/48, i.e., an expected additional 1/12 bytes per tower. We conclude that the expected space requirement for a skiplist with n items is about $25.42 \cdot n$ bytes plus the space for the keys and informations.

The macro *NEW_NODE*(v, h) allocates space for a node of height h and the macro *FREE_NODE*(v) frees that space again. Both macros use the LEDA memory management scheme. The macros *NEW_HEADER*(v) and *FREE_HEADER*(v) do the same for header nodes. Recall that a header always contains *MaxHeight* + 1 forward pointers.

⟨*memory management*⟩≡

```
inline int NODE_SIZE(int l)
{ int l1 = 0;
  if ( l > 0 )  // compute smallest power of two >= l
  { l1 = 1;
    while (l1 < l) l1 <<= 1;
  }
  return int(sizeof(skiplist_node))+
              (l1)*int(sizeof(skiplist_node*));
}

#define NEW_NODE(v,l)                                    \
v = (sl_item)std_memory.allocate_bytes(NODE_SIZE(l));    \
v->height = l;

#define FREE_NODE(v)                                     \
std_memory.deallocate_bytes(v,NODE_SIZE(v->height))

inline int HEADER_SIZE()
{ int l1 = 1;
  while (l1 < MaxHeight) l1 <<= 1;
  return int(sizeof(header_node))+
             (l1)*int(sizeof(skiplist_node*));
}

#define NEW_HEADER(v)                                    \
v = (large_item)std_memory.allocate_bytes(HEADER_SIZE());\
v->height = MaxHeight;

#define FREE_HEADER(v)                                   \
std_memory.deallocate_bytes(v,HEADER_SIZE())
```

5.7.5 *Construction, Assignment and Destruction*

The class *skiplist* has two constructors. The first constructor constructs an empty skiplist and the second constructor copies its argument.

Let us look more closely at the first constructor. We allocate a tower of height *MaxHeight* for *header* and a tower of height -1 for *STOP*. The *true_height* of the *header* is 0 and hence only the level 0 forward pointer of *header* is initialized.

The copy constructor first constructs an empty skiplist and then copies its argument L element by element. Since the constructor of class *skiplist* uses the trivial versions of the virtual functions *copy_key* and *copy_inf*, the calls of *copy_key* and *copy_inf* in *insert_at_item* have no effect, and we therefore have to use L's version of these functions to do the copying. This is a problem which arises in the implementation of all copy constructors; see Section 13.1 for a general discussion. *insert_at_item* is defined in Section 5.7.8.

The default constructor takes constant time and the copy constructor takes linear expected time plus the time to copy n keys and informations.

⟨*constructors and related functions*⟩≡

```
skiplist::skiplist(float p)
{ prob = p;
  randomsLeft = 0;
#ifdef SMEM
  header = new header_node;
  header->forward = new sl_item[MaxHeight+1];
  header->height = MaxHeight;
  STOP = new skiplist_node;
  STOP->height = -1;
#else
  NEW_HEADER(header);
  NEW_NODE(STOP,-1);
#endif
  header->true_height = 0;
  header->myseq = this;
  STOP->backward= (sl_item) header;
  STOP->pred= (sl_item) header;
  header->forward[0] = STOP;
}

skiplist::skiplist(const skiplist& L)
{ prob = L.prob;
  randomsLeft = 0;
#ifdef SMEM
  header = new header_node;
  header->forward = new sl_item[MaxHeight+1];
  header->height = MaxHeight;
  STOP = new skiplist_node;
  STOP->height = -1;
#else
  NEW_HEADER(header);
  NEW_NODE(STOP,-1);
#endif
  header->true_height = 0;
  header->myseq = this;
  STOP->backward= (sl_item) header;
  STOP->pred= (sl_item) header;
```

```
    header->forward[0] = STOP;
    sl_item p = L.STOP->pred;
    while (p!= L.header)
    { insert_at_item(header,p->key,p->inf);
      L.copy_key(p->key);
      L.copy_inf(p->inf);
      p = p->pred;
    }
  }
```

We come to the assignment operator, the function *clear*, and the destructor. The assignment operator first clears the skiplist and then copies its argument. The *clear* function deletes all nodes of a skiplist and the destructor first calls *clear* and then deletes the two non-proper towers.

It would not do to copy the body of *clear* into the destructor since ∼*skiplist* uses the trivial versions of the virtual functions *clear_key* and *clear_inf* and hence does not know how to destroy a key or inf. This is a problem which arises in the implementation of all destructors; see Section 13.4.3 for a general discussion.

All three functions take linear expected time plus the time to copy or clear *n* keys and informations.

⟨*constructors and related functions*⟩+≡

```
  skiplist& skiplist::operator=(const skiplist& L)
  { clear();
    sl_item p = L.STOP->pred;
    while (p!= L.header)
    { insert_at_item(header,p->key,p->inf,after);
      p = p->pred;
     }
    return *this;
   }
  void skiplist::clear()
  { register sl_item p,q;
    p = header->forward[0];
    while(p!=STOP)
    { q = p->forward[0];
      clear_key(p->key);
      clear_inf(p->inf);
  #ifdef SMEM
      delete p->forward;
      delete p;
  #else
      FREE_NODE(p);
  #endif
      p = q;
     }
   header->true_height = 0;
   header->forward[0] = STOP;
   STOP->pred= (sl_item) header;
```

```
}

skiplist::~skiplist()
{ clear();
#ifdef SMEM
  delete header->forward;
  delete header;
  delete STOP;
#else
  FREE_HEADER(header);
  FREE_NODE(STOP);
#endif
}
```

5.7.6 *Search Operations*

Skiplists offer a wide variety of search operations. We first give a fairly general search function called *search* and then derive the other search functions from it. *Search* takes a *key*, an item v and an integer h and returns a node q and an integer l. The node v has height at least h and *key* is known to lie between $v \rightarrow key$ (exclusive) and $v \rightarrow forward[h] \rightarrow key$ (inclusive). In the formulation of this precondition we used our simplifying assumption that the keys of *header* and *STOP* are $-\infty$ and ∞, respectively. *Search* finds the unique node q such that *key* lies between $q \rightarrow pred \rightarrow key$ (exclusive) and $q \rightarrow key$ (inclusive). If *key* is equal to $q \rightarrow key$ then $l \geq 0$; otherwise, $l < 0$.

The principle underlying *search* is simple. It maintains items p and q and a height k, $k \geq -1$, such that p's height is at least $k+1$, q is the level $k+1$ successor of p and $p \rightarrow key < key \leq q \rightarrow key$. Initially $k = h - 1$. If k is -1 then q is returned. If $k \geq 0$ then we search through level k starting at $p \rightarrow forward[k]$ to determine the new p and q.

```
q = p->forward[k];
while (key > q->key)   {  p = q; q = p->forward[k]; }
```

The basic strategy can be slightly optimized as follows. Before making a comparison between keys we check whether the current q has height k (otherwise, it is already known that $key \leq q \rightarrow key$). This optimization is worthwhile when a comparison between keys is considerably more expensive than a comparison between integers. This is the case when the comparison is made by calling *cmp* and it is not the case when the comparison is made by the operator $<$ for *ints* or *doubles*.

The expected running time of *search* is $O(1+h)$ since $h+1$ levels are visited and since the expected time spent on each level is constant. The easiest way to see the latter fact is to traverse the search path backwards and to recall that after following a constant expected number of backward pointers a higher tower is reached.

We give three versions of *search*, one called *gen_search* and working for arbitrary key type K, one called *double_search* and working only for keys of type *double*, and one called *int_search* and only working for keys of type *int*. *Search* selects the appropriate version by switching on the value of *key_type_id*. A general discussion of this optimization strategy can be found in Section 13.5.

⟨*search functions*⟩≡

```
sl_item skiplist::search(sl_item v, int h, GenPtr key, int& l) const
{ switch (key_type_id()) {
  case INT_TYPE_ID:    return int_search(v,h,key,l);
  case DOUBLE_TYPE_ID: return double_search(v,h,key,l);
  default:             return gen_search(v,h,key,l);
  }
}

sl_item skiplist::gen_search(sl_item v, int h, GenPtr key, int& l) const
{ register sl_item p = v;
  register sl_item q = p->forward[h];
  l = 0;

#ifdef CHECK_INVARIANTS
  assert(p->height == MaxHeight || cmp(key,p->key) > 0);
  assert(q->height < 0 || cmp(key,q->key) <= 0);
#endif

  if (q->height >= 0 && cmp(key,q->key) == 0)  return q;

  int k = h - 1;
  int c = -1;

  while (k >=0)
  { /* p->key < key < p->forward[k+1]->key and c = -1 */
    q = p->forward[k];
    while (k == q->height && (c = cmp(key,q->key)) > 0)
    { p = q;
      q = p->forward[k];
    }
    if (c == 0) break;
    k--;
  }
  l = k;

#ifdef CHECK_INVARIANTS
  p = q->pred;
  assert(p->height == MaxHeight || cmp(key,p->key) > 0);
  assert(q->height <  0 || cmp(key, q->key) <= 0);
  assert(l >= 0 && cmp(key,q->key) == 0 ||
  ( l < 0 && (q->height < 0 || cmp(key,q->key) < 0)));
#endif

  return q;
}
```

In the versions of *search* for integer and double keys we perform the following optimizations: we avoid the call of *cmp* and call the comparison operators $<$, $\leq$, $=$, $\ldots$ instead. Moreover, we drop the comparison `k == q->height`, as it does not pay for integer keys.

⟨*search functions*⟩+≡

```
sl_item skiplist::int_search(sl_item v, int h, GenPtr key, int& l) const
{ sl_item p = v;
  sl_item q = p->forward[h];
  l = 0;
  int ki = LEDA_ACCESS(int,key);
  int k = h - 1;
  STOP->key = key;
  while (k >= 0)
  { /* p->key < key <= p->forward[k+1]->key */
    q = p->forward[k];
    while ( ki > LEDA_ACCESS(int,q->key) )
    { p = q;
      q = p->forward[k];
    }
    if ( ki == LEDA_ACCESS(int,q->key) && q != STOP ) break;
    k--;
  }
  l = k;
#ifdef CHECK_INVARIANTS
  p = q->pred;
  assert(p->height==MaxHeight || ki>LEDA_ACCESS(int,p->key));
  assert(q->height <  0 || ki <= LEDA_ACCESS(int,q->key));
  assert(l >= 0 && ki == LEDA_ACCESS(int,q->key) ||
  ( l < 0 && (q->height<0 || ki<LEDA_ACCESS(int,q->key))));
#endif
  return q;
}
```

We refrain from showing the version for double keys. For all other search functions we will only show the generic version.

It is easy to derive the other search functions from the basic routine *search*. The call *locate_succ*(k) returns the item $\langle k1, i\rangle$ with $k \leq k1$ and $k1$ minimal (*nil* if there is no such item), *locate_pred* is symmetric to *locate_succ*, *locate* is synonymous to *locate_succ* and *lookup*(k) returns the item $\langle k, i\rangle$ (*nil* if there is no such item). All operations in this section take logarithmic time.

⟨*search functions*⟩+≡

```
sl_item skiplist::locate_succ(GenPtr key) const
{ int l;
  sl_item q = search(header,header->true_height,key,l);
  return (q == STOP) ? 0 : q;
}
sl_item skiplist::locate(GenPtr key) const { return locate_succ(key); }
sl_item skiplist::locate_pred(GenPtr key) const
{ int l;
  sl_item q = search(header,header->true_height,key,l);
  if (l < 0) q = q->pred;
  return (q == header) ? 0 : q;
}
sl_item skiplist::lookup(GenPtr key) const
{ int k;
  sl_item q = search(header,header->true_height,key,k);
  return (k < 0) ? 0 : q;
}
```

5.7.7 *Finger Searches*

We describe four versions of finger search.

The first three versions take a *key* and locate an item q and an integer l such that $q \rightarrow pred \rightarrow key < key \leq q \rightarrow key$ and $l \geq 0$ iff $key = q \rightarrow key$ and run in time $O(\log d)$, $O(\log(n-d))$, and $O(\log \min(d, n-d))$, respectively, if q is the d-th item in a list of n items. We first show how to obtain the time bounds $O(\log d)$ and $O(\log(n-d))$, respectively.

To achieve the first bound we compare *key* with the key of $header \rightarrow forward[k]$ for k equal to 0, 1, ... until a key at least as large as *key* is found. When this is the case we start a standard search at level k from the header.

```
k = 0;
while ( k < true_height )
{ if ( key <= header->forward[k]->key ) break;
  k++;
}
search(header,k,key,l);
```

Since the expected maximal height among the first d towers is $O(\log d)$ the expected maximal value of k is $O(\log d)$ and the time bound follows.

In order to achieve the second bound we compare *key* with the key of the rightmost tower q_k of height at least k for k equal to 0, 1, ... until a key smaller than *key* is found. When this is the case we start a standard search at level k from q_k. We can find q_k from q_{k-1} by following an expected constant number of backward pointers.

```
k = 0;
q = STOP->pred;
while ( k < true_height )
{ if ( key > q->key ) break;
  k++;
  while ( q->height < k ) q = q->backward;
}
search(q,k,key,l);
```

Since the expected maximal height among the last $n-d$ towers is $O(\log(n-d))$ the expected maximal value of k is $O(\log(n-d))$ and the time bound follows.

In order to obtain the minimum of both time bounds we perform the two searches simultaneously (also called dove-tailed), i.e., we merge the two loop bodies into one, and stop as soon as one of the two searches tells us to stop.

As in the case of standard searches we provide optimizations for keys of type *int* or *double*.

⟨*search functions*⟩+≡

```
sl_item skiplist::finger_search_from_front(GenPtr key, int& l) const
{ switch (key_type_id()) {
  case INT_TYPE_ID:    return int_finger_search_from_front(key,l);
  case DOUBLE_TYPE_ID: return double_finger_search_from_front(key,l);
  default:             return gen_finger_search_from_front(key,l);
  }
}
sl_item skiplist::gen_finger_search_from_front(GenPtr key, int& l) const
{ sl_item q = STOP->pred;
  int th = header->true_height;
  if (th == -1) return STOP;
  l = 0;
  int k = 0;
  int c1;
  while ( k < th )
  { if ( cmp(key,header->forward[k]->key) <= 0 ) break;
    k++;
  }
  return search(header,k,key,l);
}
```

and similarly

⟨*search functions*⟩+≡

```
sl_item skiplist::gen_finger_search_from_rear(GenPtr key, int& l) const
{ sl_item q = STOP->pred;
  int th = header->true_height;
  if (th == -1) return STOP;
  l = 0;
  int k = 0;
  while ( k < th )
  { if ( cmp(key, q->key) > 0 ) break;
    k++;
    while (k > q->height)  q = q->backward;
  }
  return search(q,k,key,l);
}
```

and

⟨*search functions*⟩+≡

```
sl_item skiplist::gen_finger_search(GenPtr key, int& l) const
{ sl_item q = STOP->pred;
  int th = header->true_height;
  if (th == -1) return STOP;
  l = 0;
```

```
    int k = 0;
    int c1,c2;
    while ( k < th )
    { c1 = cmp(key,header->forward[k]->key);
      c2 = cmp(key, q->key);
      if ( c1 <= 0 || c2 > 0 ) break;
      k++;
      while (k > q->height)  q = q->backward;
    }
    if (c1 <= 0)
      return search(header,k,key,l);
    else
      return search(q,k,key,l);
}
```

The fourth version of finger search takes an item v and a *key* and returns an integer l and an item q such that $q \to pred \to key < key \le q \to key$ and $l \ge 0$ iff $key = q \to key$. It runs in time $O(\log \min(d, n-d))$ where d is the number of items between v and q. The search is performed in the skiplist containing v and not in the skiplist which is given by *this*; recall the discussion in the paragraph preceding Section 5.7. This implies that we must not use the variables *header*, *STOP*, nor *true_height* in the program below. However, once we have determined the STOP node or the header node of the skiplist containing v (recall that STOP nodes are the only towers with negative height and that header nodes are the only towers with height *MaxHeight*) we can find the skiplist containing v as follows: if p is the header node of the skiplist containing v then $((large\_item)\ p) \to myseq$ is the skiplist containing v and if p is the STOP node of the skiplist containing v then $p \to backward$ is the corresponding header node and we are back to the situation where we know the header node.

The strategy used by *finger_search* is simple. If v is either the header or the STOP node of the skiplist containing v then we simply call the first version of finger search. So assume otherwise.

Assume first that *key* is larger than the key of v. For $k \ge 0$ let p_k be the rightmost tower to the left of or equal to v that has height k or more. We find the minimal k such that either p_k is a header node or $p_k \to forward[k]$ is a STOP node or *key* lies between the key of p_k and $p_k \to forward[k]$. In the first case we finish the search by calling the first version of finger search and in the last two cases (note that the second case is really a special case of the third case under the convention that the key of STOP is ∞) we start a standard search from p_k at level k. If *key* is smaller than the key of v, we use the symmetric strategy.

The running time of *finger_search* is readily determined. Assume for simplicity that q is to the right of v (the other case being symmetric) and that v is the n_1-th item in the sequence. Then v and q split the list into three parts of length n_1, $n_2 = d$, and $n_3 = n - n_1 - n_2$, respectively. Use h_i to denote the maximal height of a tower in the i-th part. Then $E[h_i] = \log n_i + O(1)$. The maximal value assumed by the variable k is equal to $h_0 = \min(h_1, h_2) = \log \min(d, n-d) + O(1)$. If the backward walk reaches the header then $h_1 \leq h_2$ and the second part of the search is the dove-tailed search of the preceding section that takes time $\min(\max(h_1, h_2), h_3)) = \min(h_2, h_3) = \log\min(d, n-d)+O(1)$. If the backward walk does not reach the header then the second part of the search is a standard search that takes time $O(h_0)$ as well.

As before we have three versions of *finger_search*, one for general keys, one for keys of type *int*, and one for keys of type *double*.

⟨*search functions*⟩+≡

```
sl_item skiplist::gen_finger_search(sl_item v, GenPtr key, int& l) const
{ l = 0;
  sl_item p = v;
  if ( p->height < 0 ) p = p->backward;
  // if p was a STOP node then it is a header now
  if ( p->height == MaxHeight )
    return ((large_item) p)->myseq->finger_search(key,l);
  int dir = cmp(key, v->key);
  if  ( dir == 0 ) return v;
  int k = 0;
  int c ;
  if (dir > 0)
  { while ( p->height < MaxHeight && p->forward[k]->height  >= 0 &&
              (c = cmp(key,p->forward[k]->key )) >= 0 )
    { if ( c == 0 ) return p->forward[k];
      k++;
      while ( k > p->height ) p = p->backward;
    }
    if ( p->height == MaxHeight )
      return ((large_item)p)->myseq->finger_search(key,l);
  }
  else
  { while ( p->height < MaxHeight && p->forward[k]->height >= 0 &&
              (c = cmp(key, p->key)) <= 0 )
    { if ( c == 0 )  return p;
      k = p->height;
      p = p->backward;
    }
    if (p->forward[k]->height  < 0 )
    { p = p->forward[k]->backward;
      return ((large_item)p)->myseq->finger_search(key,l);
    }
  }
#ifdef CHECK_INVARIANTS
assert(p->height == MaxHeight || cmp(key, p->key) > 0);
assert(p->forward[k]->height < 0 ||
             cmp(key, p->forward[k]->key) < 0);
#endif
  return search(p,k,key,l);
}
```

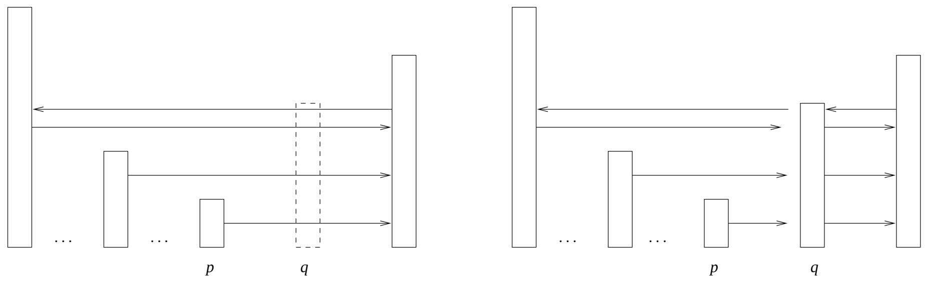


Figure 5.21 Insertion of a tower q after a tower p. All pointers that are "intersected" by the new tower are redirected.

5.7.8 *Insertions and Deletions*

We discuss the various procedures to insert into and to delete from a skiplist.

The procedure *insert_item_at_item*(q, p, dir) inserts the item q after and before p, respectively, as prescribed by *dir*. This requires to redirect pointers as shown in Figure 5.21. The *true_height* of the header is also adjusted to the maximum of the old height and 1 plus the height of the new item.

The running time of *insert_item_at_item* is proportional to the height of the new item. The expected height of the new item is constant.

⟨*insert and delete functions*⟩≡

```
void skiplist::insert_item_at_item(sl_item q, sl_item p, int dir)
{ if (dir == before) p = p->pred;
  /* insert item q immediately after item p */
  sl_item x;
  q->pred = p;
  p->forward[0]->pred = q;
  for (int k = 0; k <= q->height; k++ )
  { while (k > p->height) p = p->backward;
    x = p->forward[k];
    if (p->height == MaxHeight && x->height < 0 )
    {/* we have reached header and STOP and need to
           increase true_height */
     ((large_item) p)->true_height = k + 1;
     p->forward[k+1] = x;
    }
    q->forward[k] = x;
    p->forward[k] = q;
    if ( x->height == k ) x->backward = q;
   }
   q->backward = p;
}
```

The function *insert_at_item*(p, key, inf) modifies the skiplist in the vicinity of item p. If p's key is equal to *key* then its information is changed to *inf*. Otherwise a new item is created and inserted before or after p as dictated by *key*. The height of the new node is chosen randomly by a call *randomLevel*( ). The expected running time is constant.

⟨*insert and delete functions*⟩+≡

```
sl_item skiplist::insert_at_item(sl_item p, GenPtr key, GenPtr inf)
{ sl_item q;
  if (p->height < 0) p = p->pred;
  else
  { if ( p->height < MaxHeight )
    { int c = cmp(key,p->key);
      if (c == 0)
      { clear_inf(p->inf);
        copy_inf(inf);
        p->inf = inf;
        return p;
      }
      if ( c<0 ) p = p->pred;
    }
  }
  int k = randomLevel();
  if ( k >= MaxHeight ) k = MaxHeight - 1;
#ifdef SMEM
  q = new skiplist_node;
  q->forward = new sl_item[k+1];
  q->height = k;
#else
  NEW_NODE(q,k);
#endif
  copy_key(key);
  copy_inf(inf);
  q->key = key;
  q->inf = inf;
  insert_item_at_item(q,p,after);
  return q;
}
int skiplist::randomLevel()
{ int height = 0;
  int b = 0;
  if ( prob == 0.25 )
  { while ( b == 0 )
    { b = randomBits&3;    // read next two random bits
      randomBits >>= 2;
      randomsLeft -= 2;
      if ( b == 0 ) height++;
                      // increase height with prob 0.25
      if (randomsLeft < 2) fill_random_source();
    }
  }
  else            // user defined prob.
  { double p;
```

```
      rand_int >> p;
      while ( p < prob )
      { height++;
        rand_int >> p;
      }
    }
    return height;
  }
```

There is also a version of *insert_at_item* which inserts before or after p as directed by *dir*. The expected running time is again constant.

⟨*insert and delete functions*⟩+≡

```
  sl_item skiplist::insert_at_item(sl_item p,
                                   GenPtr key, GenPtr inf, int dir)
  { sl_item q;
    int k = randomLevel();
  #ifdef SMEM
    q = new skiplist_node;
    q->forward = new sl_item[k+1];
    q->height = k;
  #else
    NEW_NODE(q,k);
  #endif
    copy_key(key);
    copy_inf(inf);
    q->key = key;
    q->inf = inf;
    insert_item_at_item(q,p,dir);
    return q;
  }
```

This completes the discussion of the insertion procedures which insert at a given item.

Insert(k, i) inserts a new item $\langle k, i\rangle$ or changes the information of the item with key k (if there is such an item) and *del*(k) removes the item with key k.

⟨*insert and delete functions*⟩+≡

```
  sl_item skiplist::insert(GenPtr key, GenPtr inf)
  { int k;
    sl_item p = search(header,header->true_height,key,k);
    if ( k >= 0 )
    { clear_inf(p->inf);
      copy_inf(inf);
      p->inf  = inf;
      return p;
    }
    p = insert_at_item(p,key,inf,before);
    return p;
  }
```

Remove_item removes an item and *del_item* removes an item, frees its storage and also adjusts the height of the skiplist if required. The first function is used in the second and in *reverse_items*. A call *reverse_items*(p, q) with p equal or left of q reverses the subsequence with endpoints p and q.

Reverse_item has expected running time $O(d)$, where d is the length of the subsequence to be reversed. The other functions run in constant expected time.

⟨*insert and delete functions*⟩+≡

```
void skiplist::remove_item(sl_item q)
{
  if (q->height == MaxHeight || q->height < 0)
    error_handler(1,"cannot remove improper item");
  sl_item p = q->backward;
  sl_item x;
  for(int k = q->height; k >= 0; k--)
  { while ( p->forward[k] != q ) p = p->forward[k];
    x = q->forward[k];
    p->forward[k] = x;
    if ( x->height == k ) x->backward = p;
   }
  x->pred = p;
}

void skiplist::del_item(sl_item q)
{
  if (q->height == MaxHeight || q->height < 0)
    error_handler(1,"cannot delete improper item");
  remove_item(q);
  clear_key(q->key);
  clear_inf(q->inf);
  sl_item p = q->forward[q->height];
#ifdef SMEM
  delete q->forward;
  delete q;
#else
  FREE_NODE(q);
#endif
  if ( p->height < 0 )
  { large_item r = (large_item) p->backward;
    int& h = r->true_height;
    while( h > 0 && r->forward[h - 1] == p) h--;
  }
}

void skiplist::del(GenPtr key)
{ int k;
  sl_item q = search(header,header->true_height,key,k);
  if ( k>=0 ) del_item(q);
}

void skiplist::reverse_items(sl_item p, sl_item q)
{ sl_item r;
```

```
    while ( p != q )
    { r = p;
      p = p->forward[0];
      remove_item(r);
      insert_item_at_item(r,q,after);
    }
  }
```

5.7.9 *Concatenate, Split, Merge and Delete_Subsequence*

We discuss concatenation, splitting, merging, and the deletion of subsequences.

Concatenation: We describe how to concatenate two skiplists of size n_1 and n_2, respectively, in time

$$O(\log \min(n_1, n_2)).$$

Assume that the two lists to be concatenated are given by *this* and *S1*. We first make sure that *this* is the higher list (by swapping *header* and *STOP* of *this* and *S1*, if necessary) and then append *S1* to either the front or the rear of *this*. Assume that we need to append *S1* to the rear of *this*, the other case being symmetric.

There are two strategies for performing the concatenation. The first strategy places the skiplists next to each other and then removes the STOP node of the left list and the header of the second list. The work required is proportional to the height of the higher list.

The second strategy places *S1* between the last element of *this* and the STOP node of *this* and then the header node and the STOP node of *S1*. The work required is proportional to the smaller height.

We use the second strategy. The details are as follows. For any k less than the height of *S1* the k-th forward pointer out of the rightmost tower in *this* of height at least k is redirected to the first item in *S1* of height at least k and the k-th forward pointer out of the rightmost tower in *S1* of height at least k is redirected to the STOP node of *this*, see Figure 5.22.

The running time of *conc* is proportional to the smaller of the two heights and is therefore $O(\log \min(n_1, n_2))$.

⟨*concatenate and related functions*⟩≡

```
  void skiplist::conc(skiplist& S1, int dir)
  { if (header->true_height < S1.header->true_height)
    { leda_swap(header->myseq,S1.header->myseq);
      leda_swap(header,S1.header);
      leda_swap(STOP,S1.STOP);
      dir = ((dir == after) ? before : after);
    }
    if (S1.STOP->pred == S1.header)  return;
    /* S1 is non-empty and since height >= S1.height this is
       also non-empty */
    if (dir == after)
```

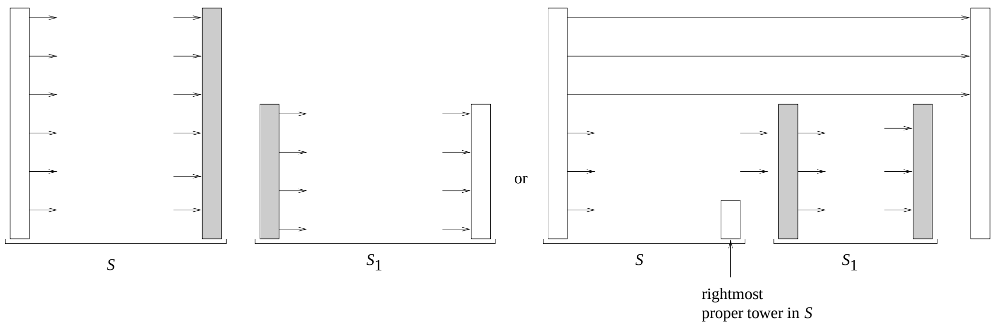


Figure 5.22 Concatenation of two skiplists S and S_1. S_1 is assumed to have smaller height and is appended at the rear of S. Only the header and STOP nodes of the lists are shown; their *true_height* is indicated as the height of the corresponding rectangles. The left part illustrates the first strategy and the right part illustrates the second strategy. The shaded towers are removed.

```
{ sl_item p = STOP->pred;
  sl_item q = S1.STOP->pred;
  assert(cmp(p->key, S1.header->forward[0]->key) < 0);
  STOP->pred = q;
  S1.header->forward[0]->pred = p;
  for (int k = 0; k < S1.header->true_height; k++)
  { /* p and q are the rightmost items of height at
       least k in this and S1, respectively */
    sl_item r = S1.header->forward[k];
    p->forward[k] = r;
    if ( r->height == k ) r->backward = p;
       q->forward[k] = STOP;
    while (p->height == k) p = p->backward;
    while (q->height == k) q = q->backward;
  }
}
else
{ sl_item q = S1.STOP->pred;
  assert(cmp(q->key, header->forward[0]->key) < 0);
  S1.header->forward[0]->pred= (sl_item) header;
  header->forward[0]->pred = q;
  for (int k = 0; k < S1.header->true_height; k++)
  { // q is the rightmost item of height at least k in S1
    sl_item r = header->forward[k];
    q->forward[k] = r;
    if (r->height == k) r->backward = q;
    r = S1.header->forward[k];
    header->forward[k] = r;
    if (r->height == k) r->backward= (sl_item) header;
    while (q->height == k) q = q->backward;
  }
}
S1.header->true_height = 0;
```

```
    S1.STOP->pred = (sl_item) S1.header;
    S1.header->forward[0] = S1.STOP;
  #ifdef CHECK_INVARIANTS
    this->check_data_structure("this in conc");
    check_data_structure(S1,"S1 in conc");
  #endif
  }
```

We need to explain the last call of *check_data_structure*. In *delete_subsequence* we call *conc* with an argument *S1* that is locally defined within *delete_subsequence*. This *S1* is a skiplist but not a *sortseq* and hence its virtual functions have never been redefined. We therefore use *this* as the implicit argument of the last call of *check_data_structure* and in this way give it access to the redefined versions of the virtual functions.

Split: *S.split_at_item*$(p, S1, S2, dir)$ splits the skiplist containing p before or after item p into lists *S1* and *S2* as directed by *dir* in time proportional to the logarithm of the shorter result. We use P to denote the skiplist containing p. Clearly, *S1* and *S2* must be distinct, but one of them may be equal to P . If both of them are different from P then P is empty after the split. The primary argument *S* may be any skiplist. It must have the same type as P, *S1*, and *S2*.

A method whose running time is proportional to the logarithm of the size of P is easy to describe. We simply erect two new improper towers before or after p.

In order to obtain a running time that is proportional to the height of the smaller result list, we have to reuse *header* and *STOP* of P for the larger output list. We proceed as follows. We first determine the lower of the two outputs by simultaneously walking from p and its successor (this assumes $dir == after$) to *header* and *STOP* until one of the two walks reaches its destination. Let *max_lev* be the maximal level reached, i.e., both sublists contain a tower of height *max_lev* and for one of the sublists this is the maximal height.

Assume first *max_lev* is the maximal height of a tower in *S1*, i.e., in the left sublist. Then $1 + max\_lev$ is the height of *S1* and *height* is the height of *S2* after the split. We want to reuse *header* and *STOP* of P for *S2*. We interchange *header* and *STOP* of *S2* and P (this makes *S2* the input list and, if P and *S2* are distinct, makes P empty) and then remove *S1* from *S2*.

To remove *S1* from *S2* we do the following for each k, $0 \le k \le max\_lev$. Let *p_k* be the rightmost item in *S1* of height at least k. The k-th forward pointer out of *S1.header* is redirected to the destination of the k-th forward pointer out of *S2.header*, the k-th forward pointer out of *S2.header* is redirected to the destination of the k-th forward pointer out of *p_k* and the k-th forward pointer out of *p_k* is redirected to *S1.STOP*.

Assume next that *max_lev* is the maximal height of a tower in *S2*. Then the height of *S2* is $1 + max\_lev$ after the split and *height* is the height of *S1* after the split. We interchange *header* and *STOP* of *S1* and P and then remove *S2* from *S1* in a way similar to the one described above.

The running time is $O(max\_lev)$ and, if n_1 and n_2 denote the sizes of the two parts, respectively, then the expected value of *max_lev* is

$$O(\log \min(n_1, n_2)).$$

⟨*concatenate and related functions*⟩+≡

```
void skiplist::split_at_item(sl_item p,skiplist& S1,
                                     skiplist& S2,int dir)
{ if (dir == before) p = p->pred;
  sl_item p1 = p;
  sl_item p2 = p->forward[0];
  int max_lev = -1;
  while ( p1->height < MaxHeight && p2->height >= 0 )
  { /* p1 and p2 are proper towers of height
       larger than max_lev    */
    max_lev++;
    while (p1->height == max_lev) p1 = p1->backward;
    while (p2->height == max_lev) p2 = p2->forward[max_lev];
  }
  /* we have seen proper towers of height max_lev on both
     sides of the split and either p1 or p2 is a sentinel */
  large_item pheader;
  if (p1->height == MaxHeight)
    pheader = (large_item) p1;
  else
    pheader = (large_item) p2->backward;
  skiplist* Pp = pheader->myseq;
  if (Pp != &S1)  S1.clear();
  if (Pp != &S2)  S2.clear();
  if (p1->height == MaxHeight)
  { /* we reuse pheader and pSTOP for S2 */
    if (Pp != &S2)
    { leda_swap(Pp->header->myseq, S2.header->myseq);
      leda_swap(Pp->header,S2.header);
      leda_swap(Pp->STOP,S2.STOP);
    }
    S1.header->true_height = 1+max_lev;
    p1 = p;
    for (int k =0; k <= max_lev; k++)
    { // p1 is the rightmost item in S1 of height at least k
      sl_item q = S2.header->forward[k];
      S1.header->forward[k] = q;
      if (q->height == k) q->backward = (sl_item) S1.header;
      S2.header->forward[k] = p1->forward[k];
      if (p1->forward[k]->height == k)
         p1->forward[k]->backward = (sl_item) S2.header;
      p1->forward[k] = S1.STOP;
      while (k == p1->height) p1 = p1->backward;
    }
    S1.header->forward[max_lev + 1] = S1.STOP;
    /* the next line sets the predecessor of S1.STOP
       correctly if S1 is non-empty; if it is empty
       the last line corrects the mistake */
    S1.STOP->pred = p;
    S2.header->forward[0]->pred = (sl_item) S2.header;
    S1.header->forward[0]->pred = (sl_item) S1.header;
  }
  else
  { /* we want to reuse pheader and pSTOP for S1 */
    if (Pp != &S1)
    { leda_swap(Pp->header->myseq,S1.header->myseq);
      leda_swap(Pp->header,S1.header);
      leda_swap(Pp->STOP,S1.STOP);
    }
    S2.header->true_height = 1 + max_lev;
    p1 = p;
    p2 = S1.STOP->pred;
    for (int k =0; k <= max_lev; k++)
    { /* p1 and p2 are the rightmost items in S1 and S2
         of height at least k, respectively */
      sl_item q = p1->forward[k];
      S2.header->forward[k] = q;
      if (q->height == k) q->backward = (sl_item) S2.header;
      p1->forward[k] = S1.STOP;
      p2->forward[k] = S2.STOP;
      while (k == p1->height) p1 = p1->backward;
      while (k == p2->height) p2 = p2->backward;
    }
    S2.header->forward[max_lev + 1] = S2.STOP;
    /* the next line sets the predecessor of S2.STOP
       correctly if S2 is non-empty; if it is empty then
       the next line corrects the mistake */
    S2.STOP->pred = S1.STOP->pred;
    S2.header->forward[0]->pred = (sl_item) S2.header;
    S1.STOP->pred = p;
    S1.header->forward[0]->pred = (sl_item) S1.header;
  }
  if (Pp != &S1 && Pp != &S2)
  { /* P is empty if distinct from S1 and S2 */
    Pp->header->forward[0] = Pp->STOP;
    Pp->STOP->pred = Pp->STOP->backward =
                              (sl_item) Pp->header;
    Pp->header->true_height = 0;
  }
#ifdef CHECK_INVARIANTS
  this->check_data_structure("this in split");
  Pp->check_data_structure("P in split");
  check_data_structure(S1,"S1 in split");
  check_data_structure(S2,"S2 in split");
#endif
}
```

Merge: We describe how to merge two skiplists of length $n1$ and $n2$, respectively, in time $O(\binom{n1+n2}{n1})$.

Assume that the lists to be merged as given by *this* and by *S1*. We first determine the shorter list (by stepping through both lists in lock-step fashion and stopping as soon as the end of the shorter list is reached) and make sure that *this* is the larger list (by interchanging *header* and *STOP* of *this* and *S1* otherwise). We then erect a finger at the first item of *this* and consider the items of *S1* one by one. We locate the item by a finger search, insert the item into *this* and advance the finger to the point of insertion.

For the running time analysis we assume without loss of generality that $n_1 \le n_2$. For i, $1 \le i \le n_1$, let d_i be the stride of the finger search when inserting the i-th item of *S1* into *this*. Then $n_2 = \sum_i d_i$ and the total running time is $O(n_1 + \sum_i \log d_i)$. This sum is maximal if all the d_i's are equal to n_2/n_1 and is hence bounded by

$$O(n_1(1 + \log(n_2/n_1))) = O(\binom{n1+n2}{n1}).$$

⟨*concatenate and related functions*⟩$+\equiv$

```
  void skiplist::merge(skiplist& S1)
  { sl_item p= (sl_item) header;
    sl_item q = S1.header;
    while ( p->height  >= 0 && q->height >= 0 )
    { p = p->forward[0];
      q = q->forward[0];
    }
    if (q->height >= 0)
    { /* swap if this is shorter than S1 */
      leda_swap(header->myseq,S1.header->myseq);
      leda_swap(header,S1.header);
      leda_swap(STOP,S1.STOP);
    }
    /* now S1 is at most as long as this */
    sl_item finger= (sl_item) header;
    p = S1.header->forward[0];
    while (p->height >= 0)
    { sl_item q = p->forward[0];
      int l;
      finger = finger_search(finger,p->key,l);
      if (l >= 0) error_handler(1,"equal keys in merge");
      insert_item_at_item(p,finger,before);
      finger = p; // put finger at newly inserted item
      p = q;
    }

    S1.header->true_height = 0;
    S1.STOP->pred = (sl_item) S1.header;
    S1.header->forward[0] = S1.STOP;
  #ifdef CHECK_INVARIANTS
    check_data_structure("this in merge");
  S1.check_data_structure("S1 in merge");
#endif
}
```

Deletion of Subsequences: We describe how to delete a subsequence from a skiplist. More precisely, if a and b are items in a list P with a left of or equal to b then the call *S.delete_subsequence*$(a, b, S1)$ deletes the subsequence starting at a and ending at b from P and assigns it to *S1*. The running time is $O(\log \min(n_1, n - n_1))$ where n and n_1 are the length of P and *S1* respectively. *S* only provides the type.

The items a and b split P into three parts. We first determine the lowest of the parts by simultaneously walking from $a \rightarrow pred$ and from b to the left and from $b \rightarrow forward[0]$ to the right until we reach *header*, a tower left of a, or *STOP*, respectively.

If either the first or the last subsequence is lowest then the operation can be reduced to two splits and one conc. If what is to become *S1* is lowest we directly insert *S1*'s *header* and *STOP* before a and after b, respectively.

Let h_i be the height of the i-th part. Then

$$\mathrm{E}[h_2] = O(\log n_1),\ \ \mathrm{E}[h_1] = O(\log(n - n_1)),\ \ \text{and } \mathrm{E}[h_3] = O(\log(n - n_1)).$$

The time to determine the lowest part is $\min(h_1, h_2, h_3)$. If h_2 is smallest then the running time of actually deleting the subsequence is $O(h_2)$. If h_2 is not the smallest then the times for the two splits and one conc are $\min(\max(h_1, h_2), h_3))$, $\min(h_1, h_2)$, and $\min(h_1, h_3)$, respectively. All three quantities are bounded by $\min(h_2, \max(h_1, h_3))$. The expected running time is therefore

$$O(\log \min(n_1, n - n_1))$$

in both cases.

⟨*concatenate and related functions*⟩+≡

```
void skiplist::delete_subsequence(sl_item a,
  sl_item b,skiplist& S1)
{ S1.clear();
  sl_item p1 = a->pred;
  sl_item p2 = b;
  sl_item p3 = b->forward[0];
  int k = -1;
  while ( p1->height < MaxHeight && p3->height >= 0 &&
          p2->height < MaxHeight && cmp(p2->key,a->key) >= 0 )
  { k++;
    while ( p1->height == k)  p1 = p1->backward;
    while ( p2->height == k)  p2 = p2->backward;
    while ( p3->height == k)  p3 = p3->forward[k];
  }
  if (p1->height == MaxHeight || p3->height  < 0)
  { if (p1->height < MaxHeight) p1 = p3->backward;
    skiplist* Pp = ((large_item) p1)->myseq;
```

```
      skiplist S2,S3;
      split_at_item(b,S2,S3,after);
      split_at_item(a,*Pp,S1,before);
      Pp->conc(S3,after);
      return;
    }
    // the middle list is the lowest and we have to do some work
    p1 = a->pred;
    p2 = b;
    /* correct predecessor pointers */
    a->pred = (sl_item) S1.header;
    S1.STOP->pred = b;
    b->forward[0]->pred = p1;
    /* height of S1 */
    S1.header->true_height = 1 + k;
    S1.header->forward[1+k] = S1.STOP;
    for (int i = 0; i <= k; i++)
    { /* p1 and p2 are the rightmost items of height at least
         i in the first and second part, respectively */
      sl_item q = p1->forward[i];
      S1.header->forward[i] = q;
      if (q->height == i) q->backward = S1.header;
      q = p2->forward[i];
      p1->forward[i] = q;
      if (q->height == i) q->backward = p1;
      p2->forward[i] = S1.STOP;
      while (i == p1->height)  p1 = p1->backward;
      while (i == p2->height)  p2 = p2->backward;
    }
  }
```

It takes a lot of trivial stuff to complete the implementation of *skiplist*. We do not include it here to save space.

5.7.10 ***Member Functions of Class*** *sortseq*

The purpose of the file LEDAROOT/incl/LEDA/sortseq.h is to define the abstract data type class *sortseq* and to implement the abstract functions in terms of the concrete functions. We follow the general technique discussed in Section 13.4. Every abstract function (e.g. *lookup*) calls the concrete function with the same name after converting any arguments of type K or I to a generic pointer (by means of function *leda_cast*) and after converting any argument of type *sortseq*$<K,I>$ to a *skiplist* (by a cast). Similarly, any result of type K or I is converted back from generic pointer (by means of the *LEDA_ACCESS* macro). Two examples should suffice to show the principle.

```
K key(seq_item it)   const { return LEDA_ACCESS(K,IMPL::key(it)); }
seq_item lookup(K k) const { return IMPL::lookup(leda_cast(k)); }
```

5.7.11 *A Final Word*

We have given the implementation of the data type *sortseq*. We glossed over some of the trivial stuff. The complete source code can be found in the LEDA source code directory.

Exercises for 5.7

1 Implement operations *union*, *intersection*, *setminus*, and *setdifference* for sorted sequences. Start from the implementation of *merge*.

2 Add the implementation parameter mechanism to the type *sortseq*$<K, I>$. Follow the construction of the type *_sortseq*$<K, I,$ *IMPL*$>$.

3 Add the finger search operations to the ab-tree implementation or the randomized search tree implementation of sorted sequences. Inspect [Meh84a] and [AS89] for the relevant theory.

5.8 An Application of Sorted Sequences: Jordan Sorting

Let C be a Jordan curve in the plane[16] that is nowhere tangent to the x-axis. Let $x_1, x_2, \ldots, x_n$ be the abscissas of the intersection points of C with the x-axis, listed in the order the points occur on C (see Figure 5.23). Call a sequence $x_1, x_2, \ldots, x_n$ of real numbers obtainable in this way a *Jordan sequence*. The reader should convince himself at this point that the sequence 1, 3, 4, 2 is not a Jordan sequence. We describe a linear time algorithm to recognize and sort Jordan sequences due to Hoffmann et al. [HMRT85]. The Jordan demo allows you to exercise the algorithm.

As a sorting algorithm, *Jordan_sort* is not competitive with general purpose sorting algorithms, like quicksort and mergesort, despite its linear running time. We include the *Jordan_sort* program in the book as an example of how much LEDA simplifies the implementation of complex algorithms.

The Jordan sorting problem arises in the following context. Suppose we are given a simple polygon (as a sequence of edges) and a line and are asked to compute the points of intersection in the order they occur on the line. A traversal of the polygon produces the intersections in the order they occur on the polygon. Sorting the sequence of intersections produces the order on the line.

A Jordan sequence together with its intersections with the x-axis gives rise to two nested sets of parentheses, simply cut the plane at the x-axis into two half-planes (see Figure 5.24). We call a matching pair of parentheses a *bracket*. A nested set of brackets gives rise to an ordered forest in a natural way. Each bracket corresponds to a node of the tree and the children of a node correspond to the brackets directly nested within a bracket. The ordering of the children of a bracket corresponds to the left to right ordering of the subbrackets. We can turn the ordered forest into a tree by adding a fictitious bracket $(-\infty, +\infty)$. Figure 5.25

[16] A Jordan curve is a curve without self-intersections, i.e., a continuous injective mapping from the unit interval into the plane.

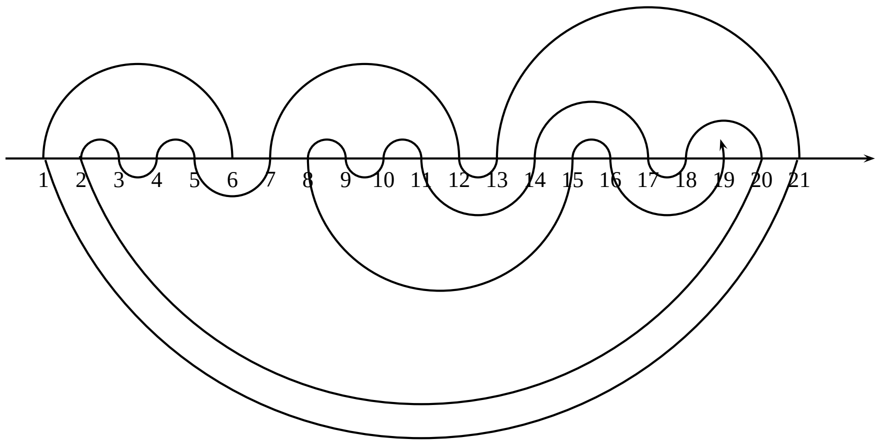


Figure 5.23 A Jordan curve and its intersections with the x-axis: The curve intersects the x-axis 21 times. We assumed for the drawing that the abscissas of the intersections are the integers 1 to 21. As the curve is traversed starting at 6 the sequence 6, 1, 21, 13, 12, 7, 5, 4, 3, 2, 20, 18, 17, 14, 11, 10, 9, 8, 15, 16, 19 is obtained.

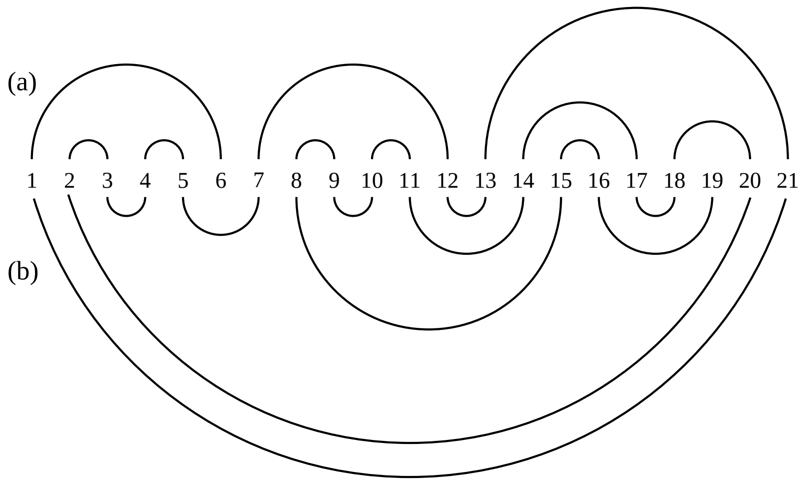


Figure 5.24 The nested parentheses corresponding to the Jordan curve of Figure 5.23; each pair of parentheses is drawn as a half-circle: (a) The parentheses corresponding to the upper half-plane; (b) The parentheses corresponding to the lower half-plane.

shows the ordered trees corresponding to the brackets of Figure 5.24. We call these trees the *lower* and the *upper tree*, respectively.

To sort a Jordan sequence $x_1, x_2, \ldots, x_n$ we process the numbers x_i in increasing order

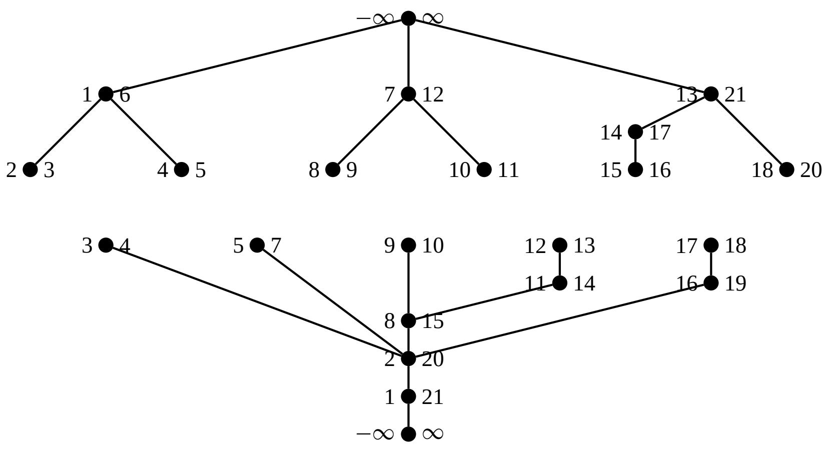


Figure 5.25 The upper and lower tree for the Jordan curve of Figure 5.23. The smaller and larger element of each bracket is on either side of the corresponding tree node.

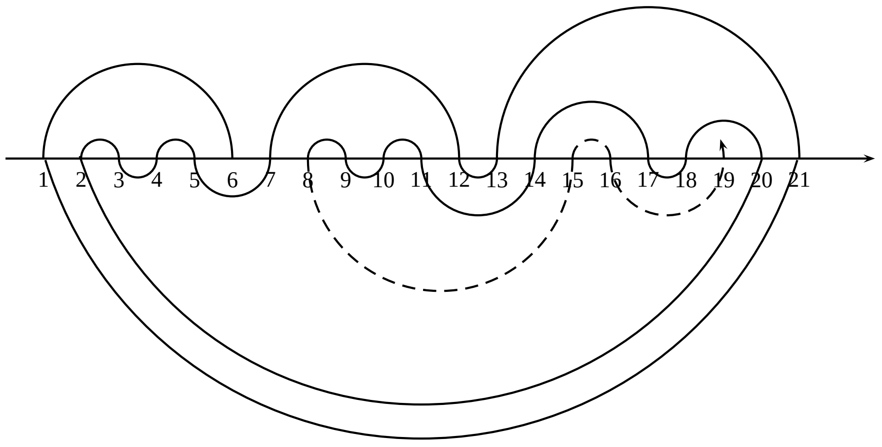
on i, constructing three objects simultaneously: the sorted list of the numbers so far processed, and the upper and lower tree of the brackets corresponding to the numbers so far processed. Figure 5.26 shows the state of the algorithm after having processed number 8 in our example.

Initially, the upper and the lower tree consist of the bracket $(-\infty, +\infty)$ and the initial sorted list is $-\infty, x_1, +\infty$. We also assume for concreteness that the curve C crosses the x-axis from bottom to top at x_1.

Assume now that we have processed $x_1, \ldots, x_i$ for some $i \geq 1$ and want to process x_{i+1} next. Assume for concreteness that the crossing at x_i is from top to bottom. So we have to insert a bracket with endpoints x_i and x_{i+1} into the lower tree. In our running example this is the bracket (8,15). Let l_i and r_i with $l_i < x_i < r_i$ be the two neighbors of x_i in the sorted list; if one of them is equal to x_1 and we insert into the lower tree then we take the neighbor in distance two. In our example we have $l_i = 7$ and $r_i = 9$. Let l_i be the bracket in the lower tree containing l_i and let r_i be the bracket containing r_i. In our example we have $l_i = (5, 7)$ and $r_i = (9, 10)$. We now distinguish cases.

Assume first that l_i is equal to r_i, i.e., (l_i, r_i) is a bracket. If x_{i+1} does not lie between l_i and r_i then we abort since the sequence is not Jordan. If x_{i+1} lies between l_i and r_i then we make the bracket $(\min(x_i, x_{i+1}), \max(x_i, x_{i+1}))$ the single child of (l_i, r_i) and insert x_{i+1} at the appropriate position into the sorted list.

Assume next that l_i is not equal to r_i. Then one of the two brackets, call it T_i, does not contain x_i. We locate x_{i+1} in the ordered sequence of siblings of T_i. Two cases can occur: either x_{i+1} is contained in one of the siblings of T_i or it is not. If x_{i+1} is contained in a sibling of T_i then we abort since the sequence is not Jordan. If x_{i+1} is not contained in a sibling of T_i then we change the lower tree as follows. We create a new node corresponding to bracket $(\min(x_i, x_{i+1}), \max(x_i, x_{i+1}))$, make all siblings of T_i that are enclosed in the



(a)

(b) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 17 18 20 21

(c)

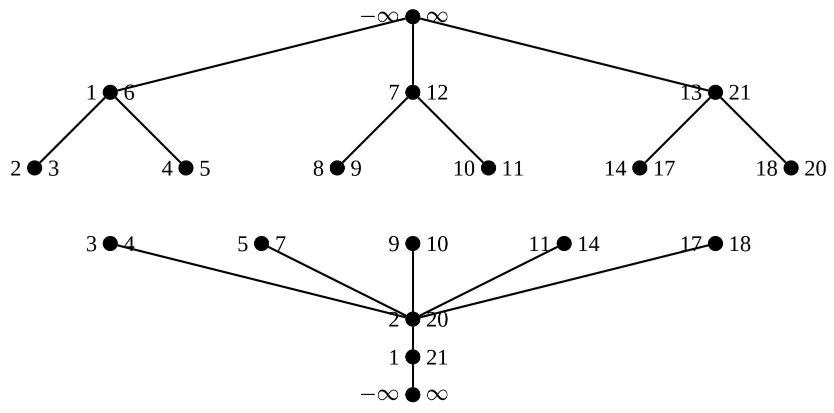


Figure 5.26 (a) The Jordan curve after reaching point 8. (b) The sorted sequence of points processed so far. (c) The lower and upper tree.

new bracket children of the new bracket, and add the new bracket to the list of siblings of T_i. We also insert x_{i+1} at the appropriate position into the sorted list of numbers processed so far.

In our example neither l_i nor r_i contains x_i and so either one of them can be T_i. The ordered list of siblings of T_i is (3, 4), (5, 7), (9, 10), (11, 14), (17, 18) and number 15 lies between brackets (11, 14) and (17, 18). So we make (9, 10) and (11, 14) children of the new bracket (8, 15) and let (8, 15) take their place in the list of children of bracket (2, 20). We also insert 15 between 14 and 17 into the sorted list of numbers processed so far. Figure 5.27 shows the lower tree after inserting the bracket (8, 15).

We proceed to describe the implementation of a procedure

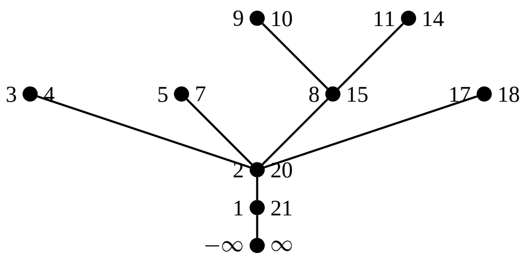


Figure 5.27 The lower tree after inserting the bracket (8,15).

```
bool Jordan_sort(const list<double>& In, list<double>& Out,
                                         window* Window = 0);
```

It takes a sequence *In* of doubles and tests whether the sequence is Jordan. If so, it returns the sorted output sequence in *Out*. If the third argument is non-nil then the execution of the algorithm is animated in *Window*. We define three files: the file Jordan.h contains the declaration of procedure *Jordan_sort*, the file Jordan.c contains its implementation, and the file Jordan_demo.c contains a demo. The latter file is not shown in the book, but can be found in LEDAROOT/demo. It includes Jordan.c as a subfile.

⟨*Jordan.h*⟩≡

```
#include <LEDA/list.h>
class window;
bool Jordan_sort(const list<double>&, list<double>&, window* Window = 0);
```

The global structure of Jordan.c is as follows:

⟨*Jordan.c*⟩≡

```
#include <LEDA/list.h>
#include <LEDA/window.h>
#include <LEDA/sortseq.h>
⟨Jordan.h⟩
⟨global variables⟩
⟨data structure⟩
⟨global functions⟩
⟨procedure Jordan sort⟩
```

As outlined above, we construct three data structures simultaneously: the sorted list L of the intersections processed so far and the upper and lower tree of brackets. We define appropriate classes. While reading these class definitions the reader may want to inspect Figure 5.28; it shows how the subtree of the upper tree rooted at the bracket $(7, 12)$ is represented.

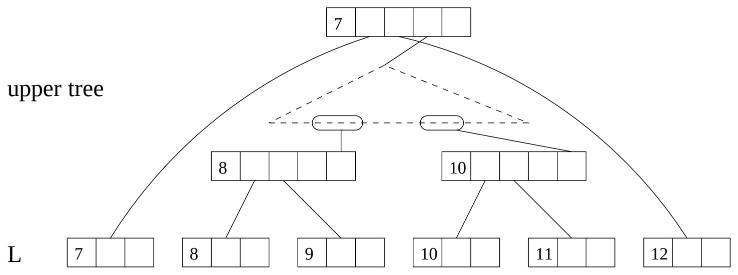


Figure 5.28 The representation of the subtree of the upper tree rooted at the bracket (7, 12). This bracket contains subbrackets (8, 9) and (10, 11). The items (= *Intersections*) of the list *L* are shown as rectangular boxes with three fields and brackets are shown as rectangular boxes with five fields. Solid lines correspond to pointers. Each intersection points to the bracket containing it which in turn points back to the intersection. Each bracket contains a *children_seq*. The *children_seq* contained in the bracket (7, 12) is shown as a dotted triangle. It has two items corresponding to the two subbrackets (8, 9) and (9, 10). The key of each item is a pointer to the subbracket and each subbracket stores in *pos_among_sibs* the item representing it in the *children_seq* of its parent. This allows, for example, the bracket (8, 9) to find the bracket (10, 11), namely if *b* is a pointer to the former bracket then $b \rightarrow$ *pos_among_sibs* is an item in the *children_seq* of bracket (7, 12), and the successor of this item is the item corresponding to (10, 11).

⟨*data structure*⟩≡

```
class intersection;
typedef intersection* Intersection;
class bracket;
typedef bracket*      Bracket;
list<Intersection> L;
```

We defined the list *L* as a list of pointers to intersections rather than a list of intersections as this will avoid frequent copying of intersections. Each intersection needs to know the bracket containing it in either tree. Therefore, an intersection contains its abscissa (a *double*) and pointers to the brackets in the two trees containing it. The constructor constructs an intersection with a particular x-coordinate.

⟨*data structure*⟩+≡

```
class intersection{
  public:
  double x;
  Bracket containing_bracket_in[2];
  intersection(double xcoord)
  { x = xcoord;
    containing_bracket_in[upper] = nil;
    containing_bracket_in[lower] = nil;
  }
};
```

A node of either tree corresponds to a bracket. A bracket needs to know its two endpoints (as items in L), its position among its siblings (a *seq_item*), and its sorted sequence of sub-brackets (a *sortseq*<*Bracket*, *int*>). We also store the x-coordinate of the left endpoint of the bracket. In order to save ink we use *children_seq* as an abbreviation for *sortseq*<*Bracket*, *int*>. The information type *int* in *children_seq* is irrelevant. *Children_seqs* need to be able to compare brackets. Brackets are compared by comparing the x-coordinates of their left endpoints. Because of the circularity (the class *bracket* needs to know about *children_seq* and *children_seq* needs a function *compare* for *Brackets*) we declare *compare* at the beginning of the next program chunk and define it at its end.

Bracket has two constructors. The first constructor takes two items a and b in L and the indicator *side* and constructs a bracket with endpoints a and b. The left endpoint is the endpoint with the smaller x-coordinate. Its x-coordinate is stored in *left_x*. The list item corresponding to the left endpoint is stored in *endpt*[*left*] and the appropriate reverse pointer is stored in the list item. The same holds true for the right endpoint. The *children* and the *pos_among_sibs* fields will be filled later.

The second constructor initializes only *left_x*. It is used to convert an x-coordinate into a bracket so that we can search[17] for the x-coordinate in a *children_seq*.

A bracket contains a number x, if x lies between the abscissa of the endpoints of the bracket.

⟨*data structure*⟩+≡

```
int compare(const Bracket&,const Bracket&);
typedef sortseq<Bracket,int> children_seq;
class bracket{
  public:
  double left_x;
  list_item endpt[2];
  children_seq children;
  seq_item pos_among_sibs;
  bracket(list_item a, list_item b, SIDE side)
  { if (L[a]->x > L[b]->x) leda_swap(a,b);
    left_x = L[a]->x;
    endpt[left] = a;
    L[a]->containing_bracket_in[side] = this;
    endpt[right] = b;
    L[b]->containing_bracket_in[side] = this;
  }
  bracket(double x){ left_x = x; }
  bool contains(double x)
  { return ( L[endpt[left]]->x < x && x < L[endpt[right]]->x ); }
```

[17] The key type of *children_seq* is *Bracket* and hence we can only search for a *Bracket* in a *children_seq*. We will have to search for a *double* and can do so only by converting the *double* into a *bracket*. This slight inconvenience would not arise if the search functions in a *sortseq*<K, I> would use a comparison function *compare*(*const* K&, *const K1*&) where the second argument type is allowed to be different from the first and in this way allowed to search for any object that can be compared with the keys of the sorted sequence.

```
};
int compare(const Bracket & b1,const Bracket & b2)
{ return compare(b1->left_x,b2->left_x); }
```

We complete the definition of the data structure with the definition of some global variables. We need a representation of ∞ for the bracket $(-\infty, \infty)$, we need a variable *side* that tells us in which tree we are working in, and we need the first abscissa *x1* and the corresponding item *x1_item* in L. We also define enumeration types {*upper*, *lower*} and {*left*, *right*} that are used to distinguish the upper and lower tree and the left and right endpoint of a bracket.

⟨*global variables*⟩≡

```
const double infty = MAXDOUBLE;
double x1;
list_item x1_item;
enum SIDE {upper,lower};
SIDE side;
enum {left,right};
```

We can now give the global structure of the Jordan sorting procedure. It takes a list *In* of *doubles* and decides whether it is Jordan. If so, it also produces a sorted version *Out* of *In*.

If the input list has length at most one then sorting is trivial. If it has length at least two then we first initialize L with $-\infty$, x_1, ∞ and build trivial upper and lower trees. Then we insert the elements of *In* one by one alternatingly into the lower or upper tree; the variable *side* keeps track of where we are. At the end we produce the sorted output list *Out*.

⟨*procedure Jordan sort*⟩≡

```
bool Jordan_sort(const list<double>& In, list<double>& Out,
                                                   window* Window)
{ if ( In.length() <= 1 ) {  Out = In; return true; }
  ⟨initialize L with x1 and construct trivial lower and upper trees⟩;
  /* we now process x_2 up to x_n */
  list_item it = In.succ(In.first()); // the second item
  side = upper;
  while (it)
  { ⟨process next input⟩;
    it = In.succ(it);
    side = ((side == upper)? lower : upper); // change sides
  }
  ⟨produce the output by copying L to Out⟩;
  return true;
}
```

We now discuss the three phases of *Jordan_sort*: initialization, processing an input, and producing the output list.

We initialize the list L with $-\infty$, x_1, ∞, and the upper and lower trees with the brackets

$(-\infty, \infty)$. We also store x_1 in *x1* and the corresponding item of L in *x1_item*. We set *xi_item* to *x1_item*; generally, *xi_item* corresponds to the last number inserted into L. This was called x_i in the discussion above.

⟨*initialize L with x1 and construct trivial lower and upper trees*⟩≡

```
x1 = In.head();
L.clear();
list_item minus_infty_item = L.append(new intersection(-infty));
list_item xi_item = x1_item = L.append(new intersection(x1));
list_item plus_infty_item = L.append(new intersection(infty));
bracket upper_root(minus_infty_item,plus_infty_item,upper);
bracket lower_root(minus_infty_item,plus_infty_item,lower);
```

We turn to the insertion part. The number to be inserted is $x = In[it]$. This was called x_{i+1} in the discussion above. Recall that *xi_item* is the item of list L holding *xi*. So the new bracket has endpoints x and *xi*. The new bracket needs to be inserted into the *side* tree.

We first determine the items *l_item* and *r_item* to the left and to the right of the current item and their corresponding intersections l and r; if one of them is equal to *x1* and *side* == *lower*, we skip it, since there is no bracket in the lower tree containing *x1*. We also retrieve the brackets *lB* and *rB* containing l and r. Then we distinguish cases according to whether the brackets *lB* and *rB* are identical or not and branch to the two sub-cases. Both sub-cases modify the list L and the *side* tree and set *x_item* to the item of L containing the new intersection.

After returning from the two sub-cases we update *xi_item*.

If *Window* is non-nil we also draw an appropriate half-circular arc into it. We divide the plane in half at $y = 50$ and draw red arcs in the upper half and black arcs in the lower half. The operation *draw_arc*(*x1*, *y1*, *x2*, *y2*, *r*, *c*) of class *window* draws a counterclockwise oriented circular arc starting in (*x1*, *y1*), ending in (*x2*, *y2*), and having radius *r* and color *c*.

⟨*process next input*⟩≡

```
double x = In[it];
double xi = L[xi_item]->x;
if (x == xi || x == x1) return false;
list_item l_item = L.pred(xi_item);
if (l_item == x1_item && side == lower) l_item = L.pred(l_item);
list_item r_item = L.succ(xi_item);
if (r_item == x1_item && side == lower) r_item = L.succ(r_item);
Intersection l = L[l_item];
Intersection r = L[r_item];
Bracket lB = l->containing_bracket_in[side];
Bracket rB = r->containing_bracket_in[side];
list_item    x_item;
if (Window != nil)
{ double r = (xi - x)/2; if (r < 0) r = -r;
 if ( side == upper)
   Window->draw_arc(point(xi,50),point((xi+x)/2,50+r), point(x,50),red);
```

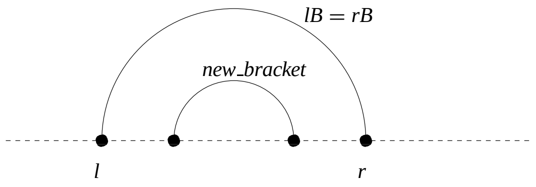


Figure 5.29 The two neighboring brackets *lB* and *rB* are indentical and hence the new bracket becomes their child.

```
 else
   Window->draw_arc(point(xi,50),point((xi+x)/2,50-r), point(x,50),black);
   }
int dir = ( x > xi ? LEDA::after : LEDA::before);
if (lB == rB)
  { ⟨lB and rB are identical⟩ }
else
  { ⟨lB and rB are distinct⟩ }
xi_item = x_item;
```

If the brackets *lB* and *rB* are identical then we only need to check whether the bracket contains the new abscissa x. If not, we abort because the input sequence is not Jordan. Otherwise we insert x next to *xi* into list L, create a new bracket, and make it the only child of *lB*, see Figure 5.29.

⟨*lB and rB are identical*⟩≡

```
if (!(lB->contains(x))) return false;
x_item = L_insert(x,xi_item,dir);
Bracket new_bracket = new bracket(x_item,xi_item,side);
new_bracket->pos_among_sibs = lB->children.insert(new_bracket,0);
```

The procedure *L_insert* is essentially identical to *L.insert*. A small difference arises from the fact that *x1_item* is not part of a bracket on the lower side and hence if the new intersection is to be inserted next to *x1* then its position with respect to *x1* is not yet known.

⟨*global functions*⟩≡

```
list_item L_insert(double x, list_item it, int dir)
{ if ( side == lower &&
     (dir == LEDA::before && L.pred(it) == x1_item && x < x1) ||
     (dir == LEDA::after &&  L.succ(it) == x1_item && x > x1) )
     it = x1_item;
  return L.insert(new intersection(x),it,dir);
}
```

We come to the case in which the brackets *lB* and *rB* are not identical. We distinguish two cases.

If x lies between l and r then the new bracket (xi, x) does not enclose any brackets. We

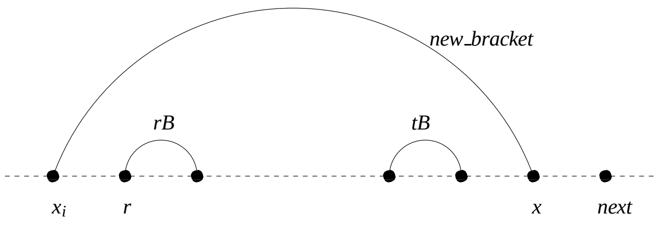


Figure 5.30 The new bracket extends to the right and x is larger than r; *tB* is the rightmost sibling of *rB* whose left endpoint is less than x. x must not be contained in *tB* and it must be contained in the parent bracket of *tB*. The children of the new bracket start at *rB* and end at *tB*.

therefore only have to insert x either before or after *xi* into L and make the new bracket either the left sibling of *rB* (if *lB* contains x) or the right sibling of *lB* (otherwise).

If x does not lie between l and r, we have to work harder.

⟨*lB and rB are distinct*⟩≡

```
children_seq S; // just for the type
if ( l->x < x && x < r->x )
{ x_item = L_insert(x,xi_item,dir);
  Bracket new_bracket = new bracket(xi_item,x_item,side);
  new_bracket->pos_among_sibs =
    ( lB->contains(x) ?
      S.insert_at(rB->pos_among_sibs,new_bracket,0,LEDA::before) :
      S.insert_at(lB->pos_among_sibs,new_bracket,0,LEDA::after) );
}
else
 if ( dir == LEDA::after )
 { ⟨new bracket has subbrackets and extends to the right⟩ }
 else
 { ⟨new bracket has subbrackets and extends to the left⟩ }
```

We come to the case that the new bracket extends to the right and that x is at least as large as r. Let *tB* be the rightmost sibling of *rB* whose left endpoint is less than or equal to x, cf. Figure 5.30. We determine *tB* by a finger search starting at *rB*. The right endpoint of *tB* must be smaller than x and x must be contained in the parent bracket of *tB* (which is also the parent bracket of *rB*); otherwise the sequence is not Jordan. The latter is guaranteed if x is smaller than the x-coordinate of the successor item of the right endpoint of *tB* (we skip *x1* if *side* == *lower*, as *x1* is not an endpoint of a bracket in the lower tree). Assume that both conditions hold. We add x after the right endpoint of *tB* to L, insert the new bracket (xi, x) before *rB*, delete the subsequence starting at *rB* and ending at *tB*, and make the subsequence the children sequence of the new bracket.

⟨*new bracket has subbrackets and extends to the right*⟩≡

```
Bracket query_bracket = new bracket(x);
seq_item x_pos = S.finger_locate_pred(rB->pos_among_sibs, query_bracket);
Bracket tB = S.key(x_pos);
```

```
list_item next = L.succ(tB->endpt[right]);
if ( next == x1_item && side == lower ) next = L.succ(next);
if ( x <= L[tB->endpt[right]]->x || x >= L[next]->x ) return false;
x_item = L_insert(x,tB->endpt[right],LEDA::after);
Bracket new_bracket = new bracket(xi_item,x_item,side);
new_bracket->pos_among_sibs =
  S.insert_at(rB->pos_among_sibs,new_bracket,0,LEDA::before);
S.delete_subsequence(rB->pos_among_sibs, x_pos, new_bracket->children);
```

If the new bracket has subbrackets and extends to the left we proceed symmetrically to the case above, i.e, we replace *pred* by *succ* and vice-versa, less than by greater than, … .

⟨*new bracket has subbrackets and extends to the left*⟩≡

```
Bracket query_bracket = new bracket(x);
seq_item x_pos = S.finger_locate_succ(lB->pos_among_sibs, query_bracket);
Bracket tB = S.key(x_pos);
list_item next = L.pred(tB->endpt[left]);
if ( next == x1_item && side == lower ) next = L.pred(next);
if ( x >= L[tB->endpt[left]]->x || x <= L[next]->x ) return false;
x_item = L_insert(x,tB->endpt[left],LEDA::before);
Bracket new_bracket = new bracket(x_item,xi_item,side);
new_bracket->pos_among_sibs =
  S.insert_at(tB->pos_among_sibs,new_bracket,0,LEDA::before);
S.delete_subsequence(x_pos,lB->pos_among_sibs, new_bracket->children);
```

Preparing the output is easy. After deleting the sentinels $-\infty$ and ∞ the output is available in L. We copy it to *Out*.

⟨*produce the output by copying L to Out*⟩≡

```
Out.clear();
L.pop(); L.Pop();
forall_items(it,L) Out.append(L[it]->x);
```

We described an algorithm to recognize and to sort Jordan sequences. The algorithm runs in linear time, see [HMRT85] for a proof[18]. As a sorting algorithm, *Jordan_sort* is not competitive with general purpose sorting algorithms, like quicksort and mergesort, despite its linear running time. We included the *Jordan_sort* program in the book as an example of how much LEDA simplifies the implementation of complex algorithms.

[18] The idea underlying the proof is as follows: in each iteration of Jordan sort a new bracket is constructed. This takes time $O(\log \min(k, m-k))$ where k is the number of subbrackets and $m-k$ is the number of siblings of the new bracket. One then proceeds as in the analysis of repeated splits on page 69.

Bibliography

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AMOT90] R.K. Ahuja, K. Mehlhorn, J.B. Orlin, and R.E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 3(2):213–223, 1990.

[AS89] C. Aragon and R. Seidel. Randomized search trees. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science (FOCS'89)*, pages 540–545, 1989.

[ASE92] N. Alon, J.H. Spencer, and P. Erdös. *The Probabilistic Method*. John Wiley & Sons, 1992.

[AVL62] G.M. Adel'son-Velskii and Y.M. Landis. An algorithm for the organization of information. *Soviet Math. Doklady*, 3:1259–1262, 1962.

[BM80] N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11:303–320, 1980.

[CGS97] B.V. Cherkassky, A.V. Goldberg, and C. Silverstein. Buckets, heaps, lists, and monotone priority queues. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 83–92, 1997.

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

[DKM$^+$94] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal of Computing*, 23(4):738–761, 1994.

[ES90] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[FKS84] M.L. Fredman, J. Komlos, and E. Szemeredi. Storing a sparse table with $o(1)$ worst case access time. *Journal of the ACM*, 31:538–544, 1984.

[FT87] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[GS78] L.J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (FOCS'78)*, pages 8–21, 1978.

[GT85] H.N. Gabow and R.E. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[HM82] S. Huddlestone and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[HMRT85] K. Hoffmann, K. Mehlhorn, P. Rosenstiehl, and R.E. Tarjan. Sorting jordan sequences in linear time. *Proceedings of the 1st Annual ACM Symposium on Computational Geometry (SCG'85)*, pages 196–203, 1985.

[LL97] A. LaMarca and R.E. Ladner. The influence of caches on the performance of sorting. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, pages 370–379, 1997.

[Meh84a] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer, 1984.

[Meh84b] K. Mehlhorn. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. Springer, 1984.

[Nos85] K. Noshita. A theorem on the expected complexity of Dijkstra's shortest path algorithm. *Journal of Algorithms*, 6(3):400–408, 1985.

[NR73] J. Nievergelt and E. Reingold. Binary search trees of bounded balance. *SIAM Journal of Computing*, 2:33–43, 1973.

[Pug90a] W. Pugh. A skip list cookbook. Technical Report CS-TR-2286.1, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, MD, June 1990.

[Pug90b] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[SV87] J.T. Stasko and J.S. Vitter. Pairing heaps: Experiments and analysis. *Communications of the ACM*, 30:234–249, 1987.

[Tar75] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

Index