Contents

3

Basic Data Types

The basic data types *stack*, *queue*, *list*, *array*, *random number*, *tuple*, and *string* are ubiquitous in computing. Most readers are probably thoroughly familiar with them already. All sections of this chapter can be read independently.

3.1 Stacks and Queues

A *stack* is a last-in-first-out store for the elements of some type E and a queue is a first-in-first-out store. Both data types store sequences of elements of type E; they differ in the set of operations that can be performed on the sequence. In a stack one end of the sequence is designated as the *top* of the stack and all queries and updates on a stack operate on the top end of the sequence. In a *queue* all insertions occur at one end, the *rear* of the queue, and all deletions occur at the other end, the *front* of the queue. The definitions

```
stack<E> S;
queue<E> Q;
```

define a stack S and a queue Q for the element type E, respectively. Both structures are initially empty. The following operations are available on stacks. If x is an object of type E then the insertion $S.push(x)$ adds x as the new top element. We can inspect the contents of a stack: $S.top(\,)$ returns the top element and $S.pop(\,)$ deletes and returns the top element. Of course, both operations are illegal if S is empty. The call $S.empty(\,)$ returns *true* if the stack is empty and *false* otherwise and $S.size(\,)$ returns the number of elements in the stack. So $S.empty(\,)$ is equivalent to $S.size(\,) == 0$. All elements of a stack can be removed by $S.clear(\,)$.

We illustrate stacks by a program to evaluate a simple class of expressions. The character 1 is an expression and if E_1 and E_2 are expressions then $(E_1 + E_2)$ and $(E_1 * E_2)$ are expressions. Thus, $(1+1)$ and $((1+1)*(1+(1+1)))$ are expressions, but $1+1$ and $(1+2)$ are not. The former is not an expression since it is not completely bracketed and the latter is not an expression since we only allow the constant 1 as an operand. We will ask you in the exercises to evaluate more complex expressions. There is a simple algorithm to evaluate expressions. It uses two stacks, a *stack<int> S* to hold intermediate results and a *stack<char> Op* to hold operator symbols. Initially, both stacks are empty. The expression is scanned from left to right. Let c be the current character scanned. If c is an open bracket, we do nothing, if c is a 1, we push it onto S, if c is a $+$ or $*$, we push it onto *Op*, and if c is a closing bracket, we remove the two top elements from S, say x and y, and the top element from *Op*, say *op*, and push the value x *op* y onto S. When an expression is completely scanned, its value is the top element of S, in fact, it is the only element in S. The following program assumes that a well-formed expression followed by a dot is given on standard input. It prints the value of the expression onto standard output.

$\langle$*stack_demo.c*$\rangle\equiv$

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

main()
{ char c;
  stack<int> S;  stack<char> Op;
  while ( (c = read_char("next symbol = ")) != '.' )
  { switch(c)
     {  case '(' : break;

        case '1' : { S.push(1); break; }

        case '+' : { Op.push(c); break; }

        case '*' : { Op.push(c); break; }

        case ')' : { int x = S.pop();  int y = S.pop();
                     char op = Op.pop();
                     if ( op = '+' ) S.push(x+y); else S.push(x*y);
                     break;
                   }
     }
  }
  cout << "\n\nvalue = " << S.pop() << "\n\n";
}
```

On input $((1+1)*(1+(1+1)))$ this program prints 6, on input $(1+(1+1))$ it prints 3, and on input () it crashes because it attempts to pop from an empty stack. This is bad software engineering practice and we will ask you in the exercises to remedy this shortcoming.

We turn to queues. The two ends of a queue are called the *front* and the *rear* of the queue, respectively. An insertion *Q.append*(x) appends x at the rear, *Q.top*( ) returns the front element, and *Q.pop*( ) deletes and returns the front element. Of course, the latter two calls require Q to be non-empty. The function *Q.empty*( ) checks for emptiness and *Q.size*( ) returns the number of elements in the queue. *Q.clear*( ) removes all elements from the queue.

Queues and stacks are implemented as singly linked lists. All operations take constant time except *clear*, which takes linear time. The space requirement is linear. LEDA also offers bounded queues and stacks, for example,

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

defines a stack S that can hold up to n elements. Bounded stacks and queues are implemented by arrays and hence always use the same amount of space independently of the actual number of elements stored in them. They are preferable to unbounded queues and stacks when the maximal size is known beforehand and the number of elements stored in the data structure is always close to the maximal size.

In the remainder of this section we show how to implement a queue by two stacks. This is to demonstrate the versatility of stacks, to illustrate that the same abstract data type can be implemented in many ways, to give an example of an amortized analysis of a data structure, and to amuse the user; it is not the implementation of queues used in LEDA. We use two stacks *Sfront* and *Srear* and split the queue into two parts: If $a_1, \ldots, a_m$ is the current content of *Sfront* and $b_1, \ldots, b_n$ is the current contents of *Srear* with a_m and b_n being the top elements, respectively, then $a_m, \ldots, a_1, b_1, \ldots, b_n$ is the current contents of the queue. Appending an element to the queue is realized by pushing it onto *Srear*. Popping an element from the queue is realized by popping an element from *Sfront*. If *Sfront* is empty, we first move all elements from *Srear* to *Sfront* (by popping from *Srear* and pushing onto *Sfront*). Note that this will reverse the sequence as it should be.

⟨*strange_queue.h*⟩≡

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

template<class E>
class queue {

  stack<E> Sfront, Srear;

public:
  queue<E>(){ } // initialization to empty queue

  void append(const E& x){ Srear.push(x); }

  E pop()
  { if ( Sfront.empty() )
      { while ( !Srear.empty() ) Sfront.push(Srear.pop()); }
    if ( Sfront.empty() ) error_handler(1,"queue: pop from empty queue");
    return Sfront.pop();
  }

  bool empty() { return Sfront.empty() && Srear.empty(); }

  int size()   { return Sfront.size() + Srear.size(); }
};
```

It is interesting to analyze the time complexity of this queue implementation. We claim that a sequence of n queue operations takes total time $O(n)$. To see this we note first that the constructor and the operations *append*, *empty*, and *size* run in constant time. A *pop* operation may take an arbitrary amount of time. More precisely, it takes constant time

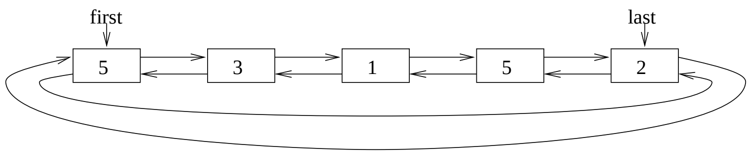

Figure 3.1 A list of five integers.

plus time proportional to the number of elements moved from *Srear* to *Sfront*. Since each element is moved at most once from *Srear* to *Sfront*, we incur a constant cost per element for moving elements from *Srear* to *Sfront*. We conclude that the time spent in all *pop* operations is linear.

Exercises for 3.1

1 Implement the type *stack*.
2 Implement the type *queue*.
3 Extend the expression evaluator such that it complains about illegal inputs.
4 Extend the expression evaluator such that it can handle arbitrary integers as operands.
5 Extend the expression evaluator such that it can handle expressions that are not completely bracketed. The usual precedence rules should be applied, i.e., $a + b * c$ is interpreted as $(a + (b * c))$. More specifically, the evaluator should be able to handle all expressions that are generated by the following four rules:

  A factor is either an integer or a bracketed expression.
  A term is either a factor or a factor times a term.
  An expression is either a term or a term plus an expression.
  That's all.

3.2 **Lists**

Lists are a simple, yet powerful, data type. It is difficult to implement a combinatorial or geometric algorithm without using lists. Moreover, the implementation of several LEDA data types, e.g., stacks, queues, and graphs, is based on lists. In this section we discuss lists for unordered and ordered element types, we sketch the implementation of lists, and in the final subsection we treat singly linked lists.

3.2.1 ***Basics***

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

declares a list L for elements of type E and initializes it to the empty list. Generally, a list L over element type E (type *list*<E>) is a sequence of items (of predefined type *list_item*), each holding an element of type E. Figure 3.1 shows a list of integers. It consists of five items shown as rectangular boxes. The contents of the first item is 5, the contents of the second item is 3, and so on. We call the number of items in a list the length of the list and use $\langle x \rangle$ to denote an item with contents x. Lists offer an extremely rich repertoire of operations.

```
L.empty();
```

checks L for emptiness. Let's assume that L is non-empty. Then

```
E          x = L.head();
list_item it = L.first();
```

assign the contents of the first item of L to x and the first item to *it*. Please pause for a moment to grasp the difference. *L.first*( ) returns the first item and *L.head*( ) returns the contents of the first item. Thus, if L is the list of Figure 3.1, the value of x is now 5 and the value of *it* is the first box. The content of the item (box) *it* can be accessed by *L.contents*(*it*) or $L[it]$. So

```
x == L.contents(it)
```

evaluates to *true* and so do

```
       3 == L.contents(L.succ(L.first());
L.last() != L.first();
     nil == L.pred(L.first());
L.tail() == L[L.cyclic_pred(L.first())];
L.last() == L.cyclic_pred(L.first()).
```

We need to explain these expressions a bit further. For a list L, *L.head*( ) and *L.tail*( ) return the contents of the first and last item of L, respectively (5 and 2 in our example) and *L.first*( ) and *L.last*( ) return the first and last item of L, respectively (the first and the fifth box in our example). The items in a list can be viewed as either arranged linearly or arranged cyclically. The operations *succ* and *pred* support the linear view of a list and the operations *cyclic_succ* and *cyclic_pred* support the cyclic view. Thus, if *it* is an item of a list L different from the last item then *L.succ*(*it*) returns the successor item of *it* and *L.succ*(*L.last*( )) returns *nil* and if *it* is different from the first item then *L.pred*(*it*) returns the predecessor item of *it* and *L.pred*(*L.first*( )) returns *nil*. *L.cyclic_pred*(*it*) and *L.cyclic_succ*(*it*) return the cyclic predecessor and successor, respectively, where the cyclic predecessor of the first item is the last item. So in the next to last expression above both sides evaluate to the contents of the last item of L and in the last expression both sides evaluate to the last item of L.

We further illustrate the use of items by the member function *print*. It takes two arguments, an output stream O and a character *space* and prints the elements of a list separated by *space* onto O. The default value of *space* is the space character. It requires that the type E offers a function *Print*(x, O) that prints an object x of type E onto O, see Section 2.8 for a discussion of the *Print*-function for type parameters.

```
template<class E>
void list<E>::print(ostream& O, char space = " ")
  { list_item it = first();
```

```
      while ( it != nil )
      {  Print(contents(it),O);
         if ( it != last_item() ) O << space;
         it = succ(it);
      }
    }
```

Note how *it* steps through the items of the list. It starts at the first item. In the general step, we first print the contents of *it* and then advance *it* to its successor item. We do so until *it* falls off the list.

Iterating over the items or elements of a list is a very frequently occurring task and therefore LEDA offers corresponding iteration macros. The iteration statements

```
forall(x,L) << body >>
```

and

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

step through the elements and items of L, respectively, and execute *body* for each one of them. Thus,

```
list_item it;
forall_items(it,L) Print(L[it],cout);

E x;
forall(x,L) Print(x,cout);
```

prints the elements of L twice. The *forall_items* loop is a macro that expands into

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

and the *forall* loop is a macro that essentially expands into

```
for ( list_item it = L.first(); it; it = L.succ(it) )
{ x = L[it];
  << body >>;
}
```

As one can see from the expansions both iteration statements work in time proportional to the length of the list. However, since the assignment $x = L[it]$ may be a costly operation (if E is a complicated type) it is usually more efficient to use the *forall_items* loop. The fact that the iteration statements for lists (and any other LEDA data type, for that matter) are realized as macros is a possible source for programming errors; we advise *to never write forall_items*$(it, f(\,))$*, where* f *is a function that produces a list*, see Sections 2.5 and 13.9 for details.

Next, we turn to update operations on lists.

```
L[it] = x;
```

changes the contents of the item *it* and

```
L.append(x);
```

adds a new item $\langle x \rangle$ after the last item of L and returns the item. We may store the item for later use:

```
list_item it = L.append(x);
```

The operations

```
L.del_item(it);
L.pop();
L.Pop();
```

remove the item *it*, the first item, and the last item of L, respectively. Each operation returns the contents of the item removed. So we may write $x = L.pop(\ )$. The program fragment

```
list<int> L;
L.append(5);
L.append(3);
list_item it = L.append(1);
L.append(5);
L.append(2);
```

builds the list of Figure 3.1 and assigns the third item of L to *it*. So $L[it]$ evaluates to 1 and $L.del\_item(it)$ removes the third item from L, i.e., L consists of four items with contents 5, 3, 5, and 2, respectively, after the call.

Two lists L and *L1* of the same type can be combined by

```
L.conc(L1,dir);
```

where *dir* determines whether *L1* is appended to the rear end ($dir = LEDA::after$) or front end ($dir = LEDA::before$) of L; *before* and *after* are predefined constants. As a side effect, *conc* clears the list *L1*. The lists L and *L1* must be distinct list objects. A list L can be split into two parts. If *it* is an item of L then

```
L.split(it,L1,L2,dir);
```

splits L before ($dir = LEDA::before$) or after ($dir = LEDA::after$) item *it* into lists *L1* and *L2*. The lists *L1* and *L2* must be distinct list objects. It is allowed, however, that one of them is equal to L. If L is distinct from *L1* and *L2* then L is empty after the split. *Split* and *conc* take constant time. Given *split* and *conc*, it is easy to write a function *splice*[1] that inserts a list *L1* after item *it* into a list L. If $it = nil$, *L1* is added to the front of L.

```
if ( it == nil )
   L.conc(L1,LEDA::before);
else
 { list<E> L2;
   L.split(it,L,L2,LEDA::after);
   L.conc(L1,LEDA::after);
   L.conc(L2,LEDA::after);
 }
```

[1] *splice* is a member function of *lists* and so there is no need to define it at the user level. We give its implementation in order to illustrate *split* and *conc*.

The *apply* operator applies a function to all elements of a list, i.e., if f is a function defined for objects of type E then

```
L.apply(f);
```

performs the call $f(x)$ for all items $\langle x\rangle$ of L. The element x is passed by reference. For example, if L is a list of integers then

```
void incr(int& i) { i++; }
L.apply(incr);
```

increases all elements of L by one. *apply* takes linear time plus the time for the function calls.

LEDA provides many ways to reorder the elements of a list.

```
L.reverse_items();
```

reverses the items in L and

```
L.permute();
```

randomly permutes the items of L. Both functions take linear time and both functions are good examples to illustrate the difference between items and their contents. The call *L.reverse_items*( ) does not change the set of items comprising the list L and it does not change the contents of any item, it changes the order in which the items are arranged in the list. The last item becomes the first, the next to last item becomes the second, and so on. Thus,

```
list_item it = L.first();
L.reverse_items();
bool b = ( it == L.last() );
```

assigns *true* to *b*.

For contrast, we give a piece of code that reverses the contents of the items but leaves the order of the items unchanged. It makes use of a function *leda_swap* that swaps the contents of two variables of the same type. We use two items *it0* and *it1* which we position initially at the first and last item of L. We interchange their contents and advance both of them. We do so as long as the items are distinct and *it0* is not the successor of *it1*. The former test guarantees termination for a list of odd length and the latter test guarantees termination for a list of even length. If the list is empty the first and the last item are *nil* and the former test guarantees that the loop body is not entered.

```
/* this is not the implementation of reverse_items */
list_item it0 = L.first();
list_item it1 = L.last();
while ( it0 != it1 && it0 != L.succ(it1) )
{ leda_swap(L[it0],L[it1]);
  it0 = L.succ(it0);
  it1 = L.pred(it1);
}
```

The above code implements

```
L.reverse().
```

We turn to sorting. We will discuss general sorting methods in the next section and discuss bucket sorting now. If f is an integer-valued function on E then

```
L.bucket_sort(f);
```

sorts L into increasing order as prescribed by f. More precisely, *bucket_sort* rearranges the items of L such that the f-values are non-decreasing after the sort and such that the relative order of two items with the same f-value is unchanged by the sort. Such a sort is called *stable*. For an example, assume that we apply *bucket_sort* to the list L of Figure 3.1 with f the identity function. This will make the third item the first item, the fifth item the second item, the second item the third item, the first item the fourth item, and the fourth item the fifth item. *bucket_sort* takes time $O(n + r - l)$, where n is the length of the list and l and r are the minimum and maximum value of $f(e)$ as e ranges over the elements of the list.

We give an application of bucket sort. Assume that L is a list of edges of a graph G (type *list<edge>*) and that *dfs_num* is a numbering of the nodes of G (type *node_array<int>*). Our goal is to reorder L such that the edges are ordered according to the number of the source of the edge, i.e., all edges out of the node with smallest number come first, then all edges out of the node with second smallest number, and so on. For an edge e of a graph G, *G.source*(e) returns the source node of the edge and hence *dfs_num*[*G.source*(e)] is the number of the source of the edge. We define a function *ord* that, given an edge e, returns *dfs_num*[*G.source*(e)] and then call *bucket_sort* with this function.

```
int ord(edge e){ return dfs_num[G.source(e)]; };
L.bucket_sort(ord);
```

3.2.2 *Lists for Ordered Sets*

Recall that a type E is linearly ordered if the function *int compare*(*const E*&, *const E*&) is defined and establishes a linear order on E, cf. Section 2.10. For lists over linearly ordered element types additional operations are available.

```
list_item L.search(E x);
```

searches for an occurrence of x in L. It uses *compare* to compare x with the elements of L. If x occurs in L, the leftmost occurrence is returned and if x does not occur in L, *nil* is returned. The running time of *search* is proportional to the distance of the leftmost occurrence of x from the front of the list. We next show how to use *search* in a primitive but highly effective implementation of the *set* data type, the so-called *self-organizing list implementation*. We realize a set over type E (type *so_set*<E>) as a list over E and use *search* to realize the *member* operation; the prefix “so” stands for self-organizing. We will make the member operation more effective by rearranging the list after each successful access. We use the operation *move_to_front*(*it*) that takes an item *it* of a list, removes it from its current position, and makes it the first element of the list. The effect of moving each accessed item to the front of the list is to collect the frequently accessed items near the front of the list. Since the access time in a list is linear in the distance from the front, this strategy keeps the expected access time small. We refer the reader to [Meh84, III.6.1.1] for the theory of self-organizing lists and turn to the implementation. We derive *so_set*<E> from *list*<E> and accordingly define a *so_set_item* as a new name for a *list_item*. We realize the membership test by *search* followed by *move_to_front* (if the search was successful), we realize *insert* by a membership test followed by append (if the membership test returned false). The other member functions are self-explanatory.

⟨*so_set.h*⟩≡
```
#include <LEDA/list.h>

typedef list_item so_set_item;

template <class E>
class so_set: private list<E>{
public:
  bool member(const E& e)
  { list_item it = search(e);
    if (it) { move_to_front(it); }
    return ( it != nil );
  }

  void insert(const E& e)               { if (!member(e)) append(e); }

  so_set_item first() const             { return list<E>::first(); }

  so_set_item succ(so_set_item it) const { return list<E>::succ(it); }

  E contents(so_set_item it) const    { return list<E>::contents(it); }
};
```

We give an application of our new data type. We read the file containing the source of this chapter, insert all its words into a *so_set*, and finally print the first thirty words in the set.

⟨*so_set_demo*⟩≡
```
main(){

so_set<string> S;

file_istream I("datatype.lw");

string s;

float T = used_time();

while ( I >> s ) S.insert(s);

cout << "time required = " << used_time(T);

so_set_item it = S.first();

for (int i = 0; i < 30; i++)
  { cout << (i % 5 == 0 ? "\n" : " ") << S.contents(it);
    it = S.succ(it);
  }
}
```

The output of this program is:

```
time required = 13.58
} \end{exercises} respectively. and $s$
of length the are $m$
$n$ where $O(n+m)$ time in
runs program that Show substring
a is $p$ if only
success this @ else p.length())
```

As expected, we see frequent English words, because the move-to-front-heuristic tends to keep them near the front of the list, and words that occurred near the end of the text, because they were accessed last.

We turn to merging and sorting. If *cmp* defines a linear order on the element type of L then

```
L.sort(cmp); L1.sort(cmp);
L.merge(L1,cmp)
```

sorts L and *L1* according to the linear order and then merges the two sorted lists. If we call the functions without the *cmp*-argument

```
L.sort(); L1.sort();
L.merge(L1);
```

the default order on the element type is used. Merging two lists of length n and m, respectively, takes time $O(n+m)$ and sorting a list of n elements takes expected time $O(n \log n)$. Let us verify this fact experimentally. We start with n equal to 128000 and repeatedly double n. For each value of n we generate a list of length n, make two copies of the list and merge them, and we permute the items of the list and then sort the list. For each value of n we output n, the measured running time for the merge and the sort, respectively, and the running time divided by n and $n \log n$, respectively.

⟨*sort_merge_times*⟩≡

```
main()
{ int min, max;
  ⟨sort merge times: read max⟩
  for (int n = min; n <= max; n = 2*n)
  { list<int> L;
    for (int j = 0; j < n; j++) L.append(j);
    list<int> L1 = L;
    list<int> L2 = L;
    float T1 = used_time();
    L1.merge(L2);
    T1 = used_time(T1);
    L.permute();
    float T2 = used_time();
    L.sort();
```

	Merging		Sorting	
n	time	normalized	time	normalized
128000	0.07	0.547	0.64	0.425
256000	0.15	0.586	1.35	0.423
512000	0.3	0.586	3.15	0.468
1024000	0.58	0.566	6.31	0.445

Table 3.1 The table produced by the experiment. All running times are in seconds. The normalized time is the $10^6 T/n$ in the case of merging and $10^6 T/(n \log n)$ in the case of sorting. The normalized time of sorting grows slowly. This is due to the increased memory access time for larger inputs. You can produce your own table by running sort_merge_times.

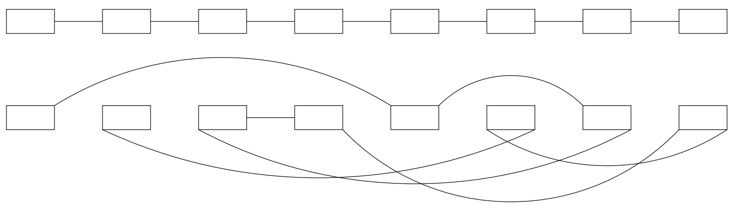

Figure 3.2 The list L before and after the call of *permute*.

```
    T2 = used_time(T2);
    ⟨sort merge times: produce table⟩
  }
}
```

Table 3.1 shows the outcome of the experiment. Does it confirm our statement that the running time of merge is $\Theta(n)$ and that the running time of sort is $\Theta(n \log n)$? In the case of merging one may say yes, since the numbers in the third column of our table are essentially constant, however, in the case of sorting the answer is a definite no, since the numbers in the last column of our table certainly grow. Why is this so? The explanation lies in the influence of cache memory on the running time of algorithms.

The internal memory of modern computers is organized hierarchically. There are at least two levels of internal memory, a small and very fast first-level memory (usually called cache) and a larger and slower second-level memory (usually called main memory). On many machines the hierarchy consists of more than two levels, see [HP90] for an excellent account of computer architecture. In the example above we first allocate a list of n items: this puts the items consecutively into storage. Then we change the order of the items in the list randomly. This leaves the items where they are and changes the links, i.e., after *permute* the links jump around widely in memory, see Figure 3.2. The job of *sort* is to untangle this

Build	Traverse	Permute	Traverse
0.59	0.16	2.77	0.44

Table 3.2 Illustration of cache effects on running time: We built a list of 1000000 items, traversed it, permuted it, and traversed it again. You may perform your own experiments with the cache_effects demo.

mess. In doing so, it frequently has to access items that are not in the fastest memory. This explains the last column of our table, at least qualitatively.

Next, we attempt a quantitative explanation. Consider the following program:

```
list<int> L;
for (int i = 0; i < 1000000; i++) L.append(i);
// L.permute();
float T = used_time();
list_item it = L.first();
while (it != nil) it = L.succ(it);
cout << used_time(T);
```

We make the following assumptions (see [HP90] for a justification): It takes ten machine instructions to execute one iteration of the while-loop. Memory is organized in two levels and the first level can hold 10000 items. An access to an item that is in first level is serviced immediately and an access to an item that is not in the first level costs an additional twenty machine cycles. An access to an item in second level moves this item and the seven items following it in second-level memory from second-level memory to first-level memory. An access to an item that is not in first-level memory in called a *cache miss*.

What behavior will we see? First assume that the list is permuted. Since the first level memory can hold only 10000 items it is unlikely that the successor of the current item is also in memory. We should therefore expect that each iteration of the loop takes thirty machine cycles, ten for the instructions executed in the loop and twenty for the transport of an item into fast memory. Next assume that the list is not permuted. Now we will incur the access time for slow memory only once in eight iterations and hence eight iterations will take a total of 100 machine cycles. In contrast, the eight iterations will take a total of 240 machine cycles on the permuted list. Thus, permuting the list will make the program about 2.4 times slower for large n. For $n = 10000$ we will see no slowdown yet, as the entire list fits in fast memory. For very large n we will see a slowdown of 2.4 and for intermediate n we will see a slowdown less than 2.4.

Table 3.2 shows actual measurements.

3.2.3 *The Implementation of Lists*

Lists are implemented as doubly linked lists. Each item corresponds to a structure (type *dlink*) with three fields, one for the contents of the item and one each for the predecessor and the successor item, and the list itself is realized by a structure (type *dlist*) containing pointers to the first and last item of the list and additional bookkeeping information. The space requirement of a list of n items is $16 + 12n$ bytes plus the space needed for the elements of the list. The contents of an item is either stored directly in the item (if it fits into four bytes) or is stored through a pointer, i.e., the e-field of a *dlink* either contains the contents of the item or a pointer to the contents of the item. In the former case there is no extra space needed for the elements of the list and in the latter case additional space for n objects of type E is needed (here E denotes the type of the objects stored in the list). All of this is discussed in detail in the chapter on implementation.

⟨*storage layout for lists*⟩≡

```
typedef dlink* list_item;
class dlink {
  dlink* succ;
  dlink* pred;
  GenPtr e;           // for the contents of the item
  // space: 3 words = 12 bytes
};
class dlist {
   dlink* h;                      // head
   dlink* t;                      // tail
   link* iterator                 // iterator, historical
   int count;                     // length of list
   // space: four words = 16 bytes
   ⟨member functions of class dlist⟩
};
```

There is no space to show the implementations of all member functions. We show only the implementation of bucket sort. The implementation is very low-level and therefore hard to understand. *Bucket_sort* assumes that a function *ord* and integers i and j are given such that *ord* maps the elements of the list into the range $[i\,..\,j]$. It uses an array *bucket* of linear lists; *bucket*$[i]$ points to the end of the i-th bucket list as shown in Figure 3.3. Initially, all bucket lists are empty. The algorithm runs through the items of the list to be sorted, computes for each item x the index $k = ord(x \to e)$ of the bucket into which the item (recall that $x \to e$ contains the object stored in item x) belongs, and appends the item to the appropriate bucket. Afterwards, it joins all bucket lists into a single list. This is done from right to left.

⟨*list: bucket sort*⟩≡

```
void dlist::bucket_sort(int i, int j)
{
  if (h == nil) return; // empty list
  int n = j-i+1;
  register list_item* bucket = new list_item[n+1];
  register list_item* stop = bucket + n;
```

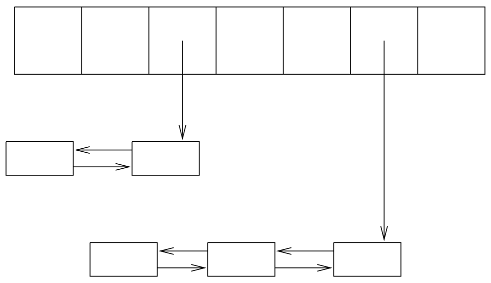

Figure 3.3 Illustration of bucket sort. We have two non-empty buckets. The list items are shown as rectangular boxes, successor pointers go from left to right, and predecessor pointers go from right to left. The pointers from the bucket array to the rears of the bucket lists are shown vertically.

```
 register list_item* p;
 register list_item q;
 register list_item x;
 for(p = bucket; p <= stop; p++)  *p = 0;
 while (h)
 { x = h;
   h = h->succ;
   int k = ord(x->e);
   if ( k >= i && k <= j )
    { // add x at end of k-th bucket
      p = bucket + k - i;
      x->pred = *p;
      if (*p) (*p)->succ = x;
      *p = x;
     }
   else
      error_handler(1,"bucket_sort: value out of range") ;
  }
for(p = stop; *p == 0; p--);
// now p points to the end of the rightmost non-empty bucket
// make it the new tail of the list.
t = *p;
t->succ = nil;
for(q = *p; q->pred; q = q->pred);
         // now q points to the start of this bucket
// link buckets together from right to left:
// q points to the start of the last bucket
// p points to end of the next bucket
while( --p >= bucket )
  if (*p)
  { (*p)->succ = q;
    q->pred = *p;
    for(q = *p; q->pred; q = q->pred);
   }
   h = q;   // head = start of leftmost non-empty bucket
   delete[] bucket;
  }
```

Aren't you glad that one of us wrote this program?

3.2.4 *Singly Linked Lists*

LEDA also offers singly linked lists (type *slist*) in which each item only knows its successor. They require space $16 + 8n$ bytes but offer a smaller repertoire of operations. Singly linked lists are used to implement stacks and queues.

Exercises for 3.2

1 Implement queues by singly linked lists.
2 Implement more operations on lists, e.g., *conc* or *merge*.
3 Write a procedure that reverses the order of the items in a list.
4 Extend the data type *so_set* to a dictionary. Realize a dictionary from K to I as a list of pointer to pairs (*list*<*two_tuple*<K, I> $*$ >). Then proceed in analogy to the text.
5 (Topological sorting) Let L be a list of pairs of integers in the range from 1 to n. Compute an ordering of the integers 1 to n such that if (x, y) is any pair in the list then x precedes y in the ordering, or decide that there is no such ordering. So if n is 4 and L is (2, 1), (1, 4), (3, 4) then 2, 3, 1, 4 is a possible ordering. Hint: 2 can go first because it does not appear as the second component of any pair.
6 Redo the calculation for the slowdown due to cache misses for the case that an iteration of the loop takes 100 clock cycles instead of ten.
7 Find out what a cache miss costs on the machine that you are using.

3.3 **Arrays**

Arrays are what they are supposed to be: collections of variables of a certain type E that are indexed by either an interval or a two-dimensional box of integers. The declarations

```
  array<string> A(3,5);
  array<string> B(10);
  array2<int>   C(1,2,4,6);
```

define two one-dimensional arrays and one two-dimensional arrays: A is a one-dimensional array of strings with index set $[3 .. 5]$, B is a one-dimensional array of strings with index set $[0 .. 9]$, and C is a two-dimensional array of integers with index set $[1 .. 2] \times [4 .. 6]$, respectively. Each entry is initialized with the appropriate default value. So each entry of A and B is initialized to the empty string and each entry of C is initialized to some integer.

We use the standard C++ subscript operator for the selection of variables in one-dimensional arrays. So $A[4]$ evaluates to the variable with index 4 in A. For two-dimensional arrays we need to use round brackets since C++ does not allow the use of angular brackets with two arguments. So $C(1, 5)$ evaluates to the variable with index (1, 5) in C. Arrays check whether their indices are legal (this can be turned off by the compiler flag `-DLEDA_CHECKING_OFF`) and hence we get an error in the following assignment:

```
A[6] = "Kurt" //  "ERROR array:: index out of range"
```

An array knows its index set. The calls *A.low*( ) and *A.high*( ) return the lower and upper index bound of A, respectively. For two-dimensional arrays we have the corresponding functions *low1*, *high1*, *low2*, and *high2*.

We illustrate arrays by two sorting functions: straight insertion sort and merge sort. Both operate on an *array*<*E*> A and assume that the element type E is linearly ordered by the function *compare*, see Section 2.10. We use $[l\,..\,h]$ to denote the index range of A. Straight insertion sort follows a very simple strategy; it sorts increasingly larger initial segments of A. Assume that we have already sorted an initial segment $A[l], \ldots, A[i-1]$ of A for some i. Initially, $i = l + 1$. In the incremental step we add $A[i]$ to the sorted initial segment by inserting it at the proper position. We determine j with $A[j] \le A[i] < A[j+1]$, move $A[j+1], \ldots, A[i-1]$ one position to the right, and put $A[i]$ into $A[j+1]$, see Figure 3.4. Straight insertion sort is a stable sorting method. Its running time is quadratic.

⟨*straight_insertion_sort*⟩≡

```
template<class E>
void straight_insertion_sort(array<E>& A)
{ int l = A.low();
  int h = A.high();
  for (int i = l + 1; i <= h; i++)
     { E x = A[i];
       int j = i - 1;
       while ( j >= l && compare(x,A[j]) < 0 )
       { A[j+1] = A[j];
         j--;
       }
       A[j+1] = x;
     }
}
```

We turn to merge sort. It is much more efficient than straight insertion sort and runs in time $O(n \log n)$ on an array of size n. The underlying strategy is also simple. Merge sort operates in phases. At the beginning of the k-th phase, $k \ge 0$, the array is partitioned into sorted blocks of size 2^k. These blocks are paired and any pair is merged into a single sorted block. In the program below we use K to denote 2^k and we use an auxiliary array B with the same index set as A. In even phases the merge step reads from A and writes into B, and in odd phases the roles of A and B are interchanged. In this way the data moves back and forth between A and B. If it ends up in B at the end of *merge_sort*, we need to copy it back to A. We use a boolean variable *even_phase* that is true iff the next phase is even. The actual merging is done by the function *merge*. A call *merge*(X, Y, i, K, h) takes the blocks of X

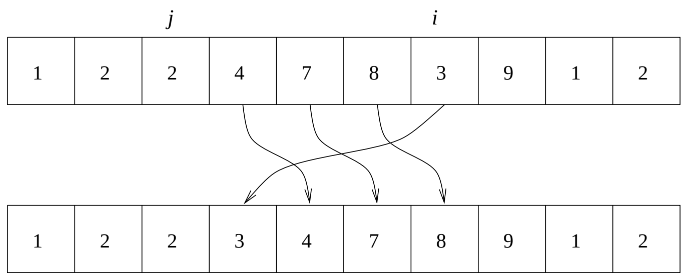

Figure 3.4 We insert $A[i]$ into the already sorted initial segment by inserting it into the proper position, say position $j+1$, and moving elements $A[j+1], \ldots, A[i-1]$ one element to the right.

starting at positions i and $i+K$, respectively, and merges them into the block of Y starting at position i and having the combined size of the two blocks to be merged. The last element of the two blocks to be merged is to be found at position h; this information is important if the size of A is not a power of two.

⟨*merge_sort*⟩≡
  ⟨*merge routine*⟩

```
template<class E>
void merge_sort(array<E>& A)
{ int l = A.low();  int h = A.high(); int n = h - l + 1;
  array<E> B(l,h);
  bool even_phase = true;
  for (int K = 1; K < n; K = 2*K)
  { for (int i = l; i <= h; i = i + 2*K)
    { if ( even_phase ) merge(A,B,i,K,h);
      else merge(B,A,i,K,h);
    }
    even_phase = !even_phase;
  }
  if ( !even_phase )
    { for (int i = l; i <= h; i++) A[i] = B[i]; }
}
```

It remains to define *merge*(X, Y, i, K, h). Our goal is to fill the block of Y starting at position i and extending to position m where $m = \min(i + 2K - 1, h)$ from the two blocks of X starting at positions i and $i + K$, respectively. The two blocks in X extend to $ml = \min(i + K - 1, h)$ and m, respectively. We maintain one index in each of the three blocks to control the merging process: The index j indicates the position in Y that is to be filled next and the indices *il* and *ih* point to the smallest remaining elements in the two blocks of X. We always move the smaller of $X[il]$ and $X[ih]$ to $Y[j]$. We break ties in favor of $X[il]$. This makes merge sort a stable sorting method.

n	2	4	8	16	32	64	128	256
insertion sort	0.83	3.27	13.5					
merge sort	0.03	0.06	0.14	0.29	0.66	1.28	2.85	5.99
insertion sort	0.47	1.74	7.07					
merge sort	0.02	0.03	0.09	0.17	0.38	0.78	1.66	3.49
member function	0.01	0.01	0.03	0.07	0.15	0.3	0.6	1.33

Table 3.3 Running times of our sorting routines and the member function *sort*. All running times are in seconds and for an array of $1000n$ integers. Insertion sort and merge sort have been compiled without and with the flag `-DLEDA_CHECKING_OFF`. You may produce you own table by calling array_sort_times.

⟨*merge routine*⟩≡

```
#include <LEDA/misc.h>  // to include Min

template<class E>
void merge(array<E>& X, array<E>& Y, int i, int K, int h)
{ int il = i; int ih = i + K;
  int ml = Min(i + K - 1,h); int m = Min(i + 2*K - 1,h);
  for (int j = i; j <= m ; j++)
    { if ( ih <= m && ( il > ml || compare(X[ih],X[il]) < 0 ) )
         { Y[j] = X[ih]; ih++; }
      else
         { Y[j] = X[il]; il++; }
    }
}
```

Table 3.3 shows the running times of our two sorting procedures in comparison to the member function *sort* for the task of sorting an array of n *ints*. Observe how the running time of insertion sort explodes. Since its running time grows proportional to n^2, it quadruples whenever n is doubled. In contrast, the running time of the two other methods is $O(n \log n)$ and hence basically doubles whenever n is doubled. The member function *sort* beats our implementation of merge sort because it exploits the fact that the objects to be sorted are *ints*, see Section 13.5.

Arrays are implemented by C++ arrays. There are important differences, however:

- The index sets may be arbitrary intervals of integers and arrays check whether their indices are legal. The index check can be turned off by the compiler flag `-DLEDA_CHECKING_OFF`.
- The entries of an array are initialized to the default value of the entry type.
- An assignment $A = B$ assigns a copy of B to A, i.e., A is made to have the same

number of variables as B and these variables are initialized with copies of the values of the corresponding variables in A. Thus, it is perfectly legal to assign an array of size 100 to an array of size 5.

- One-dimensional arrays offer some additional higher level functions which we discuss next.

We can reorder the elements of a one-dimensional array according to a linear order on the type E. The linear order may either be the default order of the type or be given by a compare function. Thus,

```
A.sort();
```

sorts the entries of our array A according to the lexicographic ordering on strings. On a sorted array we may use binary search.

```
A.binary_search("Stefan");
```

returns the index $i \in [3..5]$ containing `"Stefan"` if there is such an index and returns $A.low() - 1$ if there is no such index. We can permute the entries of an array by

```
A.permute();
```

The space required for an array of n elements is n times the space required for an object of type E. All access operations on arrays take constant time, *sort* takes time $O(n \log n)$ and *binary_search* takes time $O(\log n)$.

In many applications one needs arrays with large but sparsely used index sets, e.g., we may have 10^4 indices in the range from 0 to 10^9. In this situation it would be a complete waste of space and time to allocate an array of 10^9 elements and therefore a different data structure is called for. The data types *map* and *h_array* are appropriate. They will be discussed in Section 5.1.

Exercises for 3.3

1 Implement other sorting routines for arrays. Candidates are bubble sort, shell sort, heap sort, quick sort, and others.

2 Implement the type *array* by C++ arrays.

3 (Sparse arrays) Use lists to realize arrays whose index ranges are the integers from 0 to 2^{20}. Call the type *sparse_array*$<E>$. The constructor for the class should have an argument of type E. All elements of the array are initialized with this value. The time efficiency of your method is not important. However, the space requirement should be proportional to the number of indices for which the subscript operator was executed.

3.4 Compressed Boolean Arrays (Type int_set)

Boolean arrays are often used to represent sets. In this situation one also wants to perform the set operations *union*, *intersection*, and *complement* besides the usual operations on arrays (read the value of an entry or set the value of an entry). The data type *int_set* provides these operations in a space- and time-efficient way. It stores boolean arrays as bit-vectors, i.e., λ entries are stored in a single word on a machine with word size λ, and it uses the parallelism available at the word level to perform the set operations for λ entries in a single machine instruction. A speed-up of about λ is thus obtained for the set operations. On the other hand, reading or setting a single entry takes slightly longer than for an array.

```
int_set S(n),T(n),R(n);
```

defines S, T, and R as subsets of $[0\,..\,n-1]$ and initializes them to the empty set; the alternative definition *int_set* $S(a,b)$ defines S as a subset of $[a\,..\,b]$. If x is an integer with $0 \le x \le n-1$ then

```
S.insert(x);
S.del(x);
S.member(x);
```

inserts x into S, deletes x from S, and tests for membership of x, respectively. *S.clear*( ) makes S the empty set. The set operations union, intersection, and complement are denoted by the corresponding logical operator. So

```
S = T | R;
S = T & R;
S = ~T;
```

assigns the union of T and R, the intersection of T and R, and the complement of T to S, respectively. We also have the shorthands $S \mathrel{|}= R$ for $S = S \mid R$ and $S \mathrel{\&}= R$ for $S = S \mathbin{\&} R$. Note that the shorthands are more efficient than the verbose versions since the verbose versions first construct a temporary object and then copy that object into the left-hand side (except if your compiler is clever). The space requirement of int_sets is $O(n/\lambda)$; *insert*, *del*, and *member* take time $O(1)$, and the other operations take time $O(n/\lambda)$.

As an application of compressed boolean arrays we give an algorithm for the multiplication of boolean matrices that runs in time $O(n^3/\lambda)$. Let A and B be boolean matrices with index sets $[0\,..\,n-1] \times [0\,..\,n-1]$, and let C be their product, i.e.,

$$C(i,k) = \bigvee_{j=0}^{n-1} A(i,j) \wedge B(j,k)$$

for all i and k. The obvious method to obtain C from A and B takes time $O(n^3)$. We can obtain a faster algorithm by observing that for each i, $0 \le i < n$, the i-th row of C is the bit-wise *or* of certain rows of B, namely those that are selected by the i-th row of A. If we represent the rows of B and C as compressed arrays we obtain each row of C in time $O(n^2/\lambda)$ and hence can multiply two matrices in time $O(n^3/\lambda)$.

We give the details. First we compute a compressed version of B.

```
array<int_set*> B_compressed(0,n-1);
int i;
for (i = 0; i < n; i++)
```

```
    { B_compressed[i] = new int_set(0,n-1);
      for (int j = 0; j < n; j++)
         if ( B(i,j) ) B_compressed[i]->insert(j);
    }
```

Next we perform the multiplication. We compute each row first in compressed form and then expand it into C.

```
  int_set compressed_row(0,n-1);
  for (i = 0; i < n; i++)
    { for (int j = 0; j < n; j++)
         if (A(i,j)) compressed_row |= *B_compressed[j];
      for (j = 0; j < n; j++)
         C(i,j) = compressed_row.member[j];
      compressed_row.clear();
    }
```

Exercise for 3.4

1 Compare the method described above with the following variant of the traditional method.

```
  for (int i = 0; i < n; i++)
    for (int k = 0; k < n; k++)
      { C(i,k) = false;
        for (int j = 0; j < n; j++)
          if ( A(i,j) && B(j,k) )
            { C(i,k) = true;
              break;
            }
      }
```

How do the two algorithms perform when A and B contain only zeros and ones, respectively? Is there a way to combine the advantages of both methods?

3.5 **Random Sources**

We frequently need random values in our programs. A *random source* provides an unbounded stream of integers in some range [*low* .. *high*], where *high* and *low* are *ints* with $low \le high$ and $high - low < 2^{31}$. The size restriction comes from the fact that the implementation of random sources uses *long*s. The definition

```
random_source S(7,319);
```

defines a random source S and sets its range to [7 .. 319]. Ranges of the form $[0 .. 2^p]$ are particularly useful. Therefore we have also the definition

```
random_source S(p);
```

that sets the range to $[0\,..\,2^p{-}1]$ ($1 \leq p \leq 31$ is required) and the definition *random_source S* that sets the range to $[0\,..\,2^{31}-1]$. The random source *rand_int* is already defined in the header file `random.h`; it has range $[0\,..\,2^{31}-1]$. A random value is extracted from a source by the operator $\gg$. So

```
S >> x >> y;
```

extracts two integers in the range $[low\,..\,high]$ and assigns them to x and y; this assumes that x and y are defined as ints. Note that we are using the C++ input stream syntax for random sources, i.e., $S \gg x$ assigns to x and returns a reference to S.

We may also extract characters, unsigned integers, bools, and doubles from a random source. For the first three types this works as follows: first an integer from the range $[low\,..\,high]$ is extracted and then this integer is converted to the appropriate type. Thus, if b is a boolean variable then $S \gg b$ extracts a truth value. Note that the value of b is not uniformly distributed if $high - low + 1$ is an odd number. In particular, if $low = 0$ and $high = 2$ then we should expect the value *false* about twice as often as the value *true* (as 0 and 2 are converted to *false* and only 1 is converted to *true*). We *recommend* to extract characters and boolean values only from sources whose range spans a power of two. If a source S is asked for a double d by $S \gg d$ then a random integer $u \in [0\,..\,2^{31}-1]$ is extracted and $u/(2^{31}-1)$ is assigned to d, i.e., the value assigned to d lies in the unit interval.

The range of a random source can be changed either permanently or for a single operation: The operations *S.set_range*$(low, high)$ and *S.set_range*(p) change the range of S to $[low\,..\,high]$ and $[0\,..\,2^p-1]$, respectively, and $S(low, high)$ and $S(p)$ change the range for a single operation and return an integer in $[low\,..\,high]$ and $[0\,..\,2^p-1]$, respectively.

Of course, the stream of integers generated by a random source is only pseudo-random. It is generated from a *seed* that can either be supplied by the user (by *S.set_seed*(s)) or is generated automatically from the internal clock. If a seed is supplied then the source behaves deterministically; this is particularly useful during debugging. If no seed is supplied the sequence produced depends on the time of the day.

In the remainder of this section we describe several uses and the implementation of random sources.

A Chance Experiment: We use random sources for a chance experiment that is relevant to the analysis of merge sort for secondary memory; see [Moo61] and [Knu81, section 5.4.1]. Assume that we have to sort a set S that is too large to fit into main memory. Merge sort for external memory approaches this problem in two phases. In the first phase it partitions S and sorts each subset and in the second phase it merges the sorted subsets (usually called *runs*). Of course, it is desirable that the number of runs produced in the first phase is kept small, or in other words, that the runs produced in the first phase are long. Assume that M elements of S can be kept in main memory. Then runs of length M can be produced by reading M elements into main memory and sorting them. Longer runs can be produced by a method called *replacement selection*. This method partitions its internal memory into a priority queue Q and a reservoir R that can together store M elements. The production of runs starts by reading M elements into the priority queue. A run is generated by repeated selection of the minimum element *Q_min* from Q. This element is added to the current run (and written to secondary memory) and the spot freed in main memory is filled by the next element x from S. If x is smaller than *Q_min* then x is added to R and it is added to Q otherwise. We continue until Q becomes empty. When this is the case, the elements in R are moved to Q and the production of the next run starts. Each run produced by replacement selection has length at least M. The two extreme situations arise when S is sorted: if S is sorted in descending order then each run has exactly length M and if S is sorted in ascending order then a single run will be produced.

The program below simulates the behavior of replacement selection for a set S of random *doubles.* We maintain a priority queue Q and a stack S. We initialize Q with M random doubles and R to the empty stack. Then we start the production of runs. In each iteration we remove the smallest element *Q_min* from Q and then produce a new random double x. If $x <$ *Q_min* we add x to Q, and we add it to R otherwise. When Q is empty we move all elements from R to Q and start the production of the next run. For each run we record the quotient of the length of the run and M.

```
⟨runlength⟩≡
  main(){
  int M, n;
  ⟨read M and n⟩
  p_queue<double,int> Q;  // second type parameter is not used
  stack<double> R;
  random_source S;
  double x;
  int i;
  for (i = 0; i < M; i++) { S >> x; Q.insert(x,0); }
  array<double> RL(1,n);  // RL[i] = length of i-th run
  for (i = 1; i <= n; i++)
  { // production of i-th run
    int runlength = 0;
    while ( !Q.empty() )
    { double Q_min = Q.del_min(); runlength++ ;
      S >> x;
      if (x < Q_min) R.push(x);
      else           Q.insert(x,0);
    }
    RL[i] = (double)runlength / M;
    while ( !R.empty() ) Q.insert(R.pop(),0);
  }
  ⟨produce table runlength⟩
  }
```

Table 3.4 shows the output of a sample run; we used $M = 10^5$. The length of the i-th run

Round	Length	Round	Length	Round	Length	Round	Length
1	1.717	6	1.998	11	2	16	2.001
2	1.95	7	1.998	12	2	17	2.002
3	1.998	8	2.002	13	1.999	18	1.992
4	2.002	9	1.997	14	2.002	19	2
5	1.996	10	2	15	2	20	2.003

Table 3.4 Run formation by replacement selection, we used $M = 10^5$ and $n = 20$. You may perform your own experiments by calling program runlength.

seems to converge to $2M$ as n grows. We refer the reader to [Moo61] and [Knu81, section 5.4.1] for a proof of this fact.

We give a second interpretation of the chance experiment above. Consider a circular track on which a snow plow is operating. When the snow plow starts to operate there are M snow flakes on the track (at random locations). In every time unit the snow plow removes one snow flake and one new flake falls (at a random location). We compute how many snow flakes the snow plow removes in its i-th circulation of the track.

Random Permutations and Graphs: We show how to generate more complex random objects, namely random permutations and random graphs.

Let A be an array. We want to permute the elements of A randomly. Let $a_0, \ldots, a_{n-1}$ be the elements of A. We can generate a random permutation of these elements by selecting a random element and putting it into the last position of the permutation, selecting a random element from the remaining elements and putting it into the next to last position of the permutation, and so on. In the program below we realize this process in-place. We keep an index j into A, initially $j = n - 1$. We maintain the invariant that the elements in position 0 to j have not been selected for the permutation yet and that positions $j + 1$ to $n - 1$ contain the part of the permutation that has been produced so far. In order to fill the next position of the permutation we choose a random integer i in $[0 .. j]$ and interchange $A[i]$ and $A[j]$. We obtain

```
random_source S;
for (int j = n - 1; j >= 1; j--)  leda_swap(A[j],A[S(0,j)]);
```

where *leda_swap* interchanges its arguments. The method just described is used in operation *permute*( ) of types *array* and *list*.

Our next task is to generate a random graph with n nodes and m edges. This is very easy. We start with an empty graph G, then add n nodes to G, and finally choose m pairs of random nodes and create an edge for each one of them. A node can be added to a graph G by *G.new_node*( ). This call also returns the newly added node. We store the nodes in an *array<node>* V. In order to add a random edge we choose two random integers, say l and k, in $[0 \mathinner{..} n-1]$ and then add the edge from $V[l]$ to $V[k]$ to G.

```
random_source S;
graph G;            //empty graph
array<node> V(0,n-1);
  for (int i = 0; i < n; i++) V[i] = G.new_node();
  for (int i = 0; i < m; i++)
          G.new_edge( V[S(0,n-1)] , V[S(0,n-1)] );
```

The program above realizes the function *random_graph*(G, n, m). LEDA also offers functions to generate other types of random graphs, e.g., random planar graphs. We discuss these generators in later chapters.

Non-Uniform Distributions: We show how to generate integers according to an arbitrary discrete probability distribution. The method that we are going to describe is called the *alias-method* and has been invented by Walker [Wal77]. Let $w[0 \mathinner{..} n-1]$ be an array of positive integers. For all i, $0 \le i < n$, we interpret $w[i]$ as the weight of i. Our goal is to generate i with probability $w[i]/W$, where $W = w[0] + \ldots + w[n-1]$. We start with the simplifying assumption that n divides W and let $K = W/n$. We will remove this restriction later. We view W as an n by K arrangement of squares, n columns of K squares each and label $w[i]$ squares by i for all i, $0 \le i \le n$, see Figure 3.5. In order to generate an integer we select a random square and return its label. This makes the generation of a random integer a constant time process. The drawback of this method is that it requires space W. The space requirement can be improved to $O(n)$ by observing that there is always a labeling of the squares such that at most two different labels are used in any column. This can be seen as follows. Call a weight *small* if it is less than or equal to K and call it *large* otherwise. Clearly, there is at least one small weight. Let $w[i]$ be an arbitrary small weight. If $w[i]$ is equal to K then we assign an entire column to i and if $w[i]$ is less than K then we take an arbitrary large weight (there must be one!), say $w[j]$, and assign $w[i]$ squares to i and $K - w[i]$ squares to j. We also reduce $w[j]$ by $K - w[i]$. In either case, we have reduced the number of weights by one and are left with $n-1$ weights whose sum is $K(n-1)$. Proceeding in this way we label each column by at most two numbers.

We still need to remove the assumption that n divides W. We redefine K as $K = \lceil W/n \rceil$ and add an additional weight $w[n] = K(n+1) - W$. This yields $n+1$ weights whose sum is equal to $K(n+1)$. We can now construct a labeling as described above. We also need to modify the generation process slightly, because it is now possible that the number n is generated. When this happens we declare the generation attempt a failure and repeat. The probability of success is $W/(K(n+1))$ and hence the expected number of iterations required is $K(n+1)/W$. We need to bound this quantity. We have $W \ge n$ since each weight $w[i]$ it at least one and we have $Kn < W + n$ and hence $W > (K-1)n$ by the definition of K. Thus if $K = 1$ then $K(n+1)/W \le (n+1)/n \le 2$ and if $K \ge 2$ then $K(n+1)/W \le K(n+1)/((K-1)n) \le 4$. In either case we conclude that the expected number of iterations required is bounded by 4.

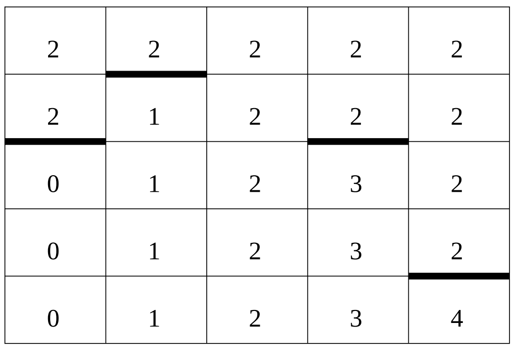

Figure 3.5 Illustration of alias-method. We have $n = 5$, $w = (3, 4, 14, 3, 1)$, and $K = 5$. The labeling shown is succinctly encoded by the vectors $T = (3, 4, 5, 3, 1)$, $L = (0, 1, 2, 3, 4)$, and $U = (2, 2, \_, 2, 2)$: for each column j the lowest T_j squares are labeled L_j and the highest $K - T_j$ squares are labeled U_j.

We turn to an implementation. We define a class *random_variate*. Its constructor takes an *array<int>* w of non-negative integers and index range $[l\,..\,h]$ and sets up the vectors T, L, and U and the integer K defined above. Its member function *generate* generates any integer $i \in [l\,..\,h]$ with probability $w[i]/W$ where $W = \sum_i w[i]$.

⟨*definition of class random_variate*⟩≡

```
class random_variate{
  array<int> T, L, U;
  int l, h, n, K;
public:
  random_variate(const array<int>& w) { ⟨random variate: constructor⟩ }
  int generate()                      { ⟨random variate: generate⟩ }
};
```

The constructor operates in two phases. In the first phase we compute the total weight W, the number n of non-zero weights, the integer K, and an array *array<int>* $u(l, h+1)$ with the additional weight $u[h+1] = K(n+1) - W$.

⟨*random variate: constructor*⟩≡

```
l =  w.low(); h = w.high();
int W = 0;
array<int> u(l,h+1);
n = 0;  // number of non-zero weights
int i;
for (i = l; i <= h; i++)
{ W += u[i] = w[i];
  if ( u[i] < 0 )
     error_handler(1,"random variate: negative weight");
  if ( u[i] > 0 ) n++;
}
```

```
if ( n == 0 ) error_handler(1,"random_variate: no non-zero weight");
K = W/n + (W % n == 0? 0 : 1);
u[h + 1] = K*(n+1) - W; n++;
```

In the second phase we set up the arrays T, L, and U. We use two stacks *Small* and *Large*: In *Small* we store all all i such that $u[i]$ is small and in *Large* we store all i such that $u[i]$ is large. We store the labeling in three arrays T, L, and U such that for every column c, $0 \leq c \leq n-1$, squares 1 to $T[c]$ are labeled $L[c]$ and squares $T[c]+1$ to K are labeled $U[c]$.

⟨*random variate: constructor*⟩+≡

```
stack<int> Small,Large;
for (i = 1; i <= h + 1; i++)
{ if ( u[i] == 0 )  continue;
  if ( u[i] <= K )  Small.push(i);
  else              Large.push(i);
}
U = T = L = array<int>(n);
for (int c = 0; c < n; c++)
{ int i = Small.pop();
  T[c] = u[i];
  L[c] = i;
  if ( u[i] < K )
  { int j = Large.pop();
    U[c] = j;
    u[j] -= (K-u[i]);
    if ( u[j] <= K ) Small.push(j); else Large.push(j);
  }
}
```

The generator chooses a random *row* and a random *column* and looks up the table entry defined by this row and column. If the table entry is different from $h+1$, it is returned. Otherwise the process is repeated.

⟨*random variate: generate*⟩≡

```
int r;
do { int row = rand_int(1,K);
     int column = rand_int(0,n-1);
     r = (row <= T[column] ? L[column] : U[column]);
   }
while (r == h + 1);
return r;
```

Random Walks in Graphs (Simulating Markov Chains): We give an application of class *random_variate*. We perform a random walk on a graph. Let $G = (V, E)$ be a directed graph and for each edge e let $w[e]$ be a non-negative weight. We start our walk in an arbitrary node of G and move according to the following rule: Suppose that we are currently in node v and let $e_0, \ldots, e_{d-1}$ be the edges out of v. We follow edge e_i with probability proportional to $w[e_i]$ for all i, $0 \le i < d$. If there is no edge out of v the walk terminates. We define a class *markov_chain* that allows us to simulate such a process.

⟨*definition of class markov_chain*⟩≡

```
class markov_chain {
  graph& G;
  int    N;
  node_array<int> visits;
  node   vcur;
  node_array<array<node> > neighbors;
  node_array<random_variate*> variate;
public:
  markov_chain(const graph& g, const edge_array<int>& w,
               node s = nil): G(g)
    { ⟨markov chain: constructor⟩ }
  void step(int T = 1) { ⟨markov chain: step⟩ }
  int  number_of_visits(node v) { return visits[v]; }
  ⟨markov chain: further member functions⟩
};
```

The constructor takes a graph G, an edge array of weights, and a start vertex. If no start vertex is specified the first node of G is taken as the start vertex. The function *step*(T) performs T steps of the random walk and the function *number_of_visits*(v) returns the number of visits to node v. We give the details below.

The constructor sets up the required data structures. We build two data structures for each node v: an *array<node> neighbors*$[v]$ that stores for each i, $0 \le i < outdeg(v)$, the target of the i-th edge out of v and a random variate *variate*$[v]$ that produces i with probability proportional to the weight of the i-th edge out of v. We set up both data structures by scanning through the edges out of v, collecting the target of the edges out of v in *neighbors*$[v]$ and their weights in a temporary array *weights*. Then we use the latter array to construct the random variate for v.

⟨*markov chain: constructor*⟩≡

```
N = 0;
visits = node_array<int>(G,0);
vcur = s; if ( s == nil) vcur = G.first_node();
neighbors = node_array<array<node> >(G);
variate   = node_array<random_variate*>(G);

node v; edge e;
forall_nodes(v,G)
{ if (G.outdeg(v) == 0) continue;
  neighbors[v] = array<node>(G.outdeg(v));
  array<int> weights(G.outdeg(v));
  int i = 0;
```

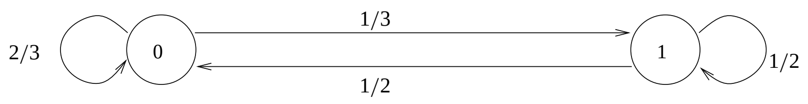

Figure 3.6 A graph with two nodes. The edge probabilities are shown next to each edge.

```
    forall_adj_edges(e,v)
    { neighbors[v][i] = G.target(e);
      weights[i] = w[e];
      i++;
    }
    variate[v] = new random_variate(weights);
  }
```

Given these data structures it is easy to perform T steps of the walk. If the outdegree of the current node is zero we stay put. Otherwise, we generate a neighbor at random and move to the neighbor.

⟨*markov chain: step*⟩≡

```
  if (T <= 0 ) return;
  for (int i = 0; i < T; i++)
  { if ( G.outdeg(vcur) == 0) return;
    vcur = neighbors[vcur][variate[vcur] -> generate()];
    visits[vcur]++;
    N++;
  }
```

Let us perform a random walk on the graph shown in Figure 3.6.

⟨*random_walk_example*⟩≡

```
  main(){
  graph G;
  node v0 = G.new_node();
  node v1 = G.new_node();
  edge e00 = G.new_edge(v0,v0); edge e01 = G.new_edge(v0,v1);
  edge e10 = G.new_edge(v1,v0); edge e11 = G.new_edge(v1,v1);
  edge_array<int> weight(G);
  weight[e00] = 2; weight[e01] = 1;
  weight[e10] = 1; weight[e11] = 1;
  while( true )
  { int N = read_int("number of steps = ");
    markov_chain M(G,weight);
    M.step(N);
    cout << "# of visits of v0 = " << M.number_of_visits(v0) <<"\n";
    cout << "# of visits of v1 = " << M.number_of_visits(v1) <<"\n";
  }
  }
```

n	1	10	100	1000	10000	100000	1000000	10000000
v_0	0	3	63	570	6058	60180	600704	6003568
v_1	1	7	37	430	3942	39820	399296	3996432

Table 3.5 The statistics of a random walk on the graph of Figure 3.6. Each column gives the number of visits to both nodes in the first n steps of the walk. You may perform your own experiments by calling random_walk.

Table 3.5 shows a sample output of this program. There is a simple analytical explanation for the output based on the theory of Markov chains, see [KSK76] for an introduction to Markov chains. Let $p_{i,n}$ be the relative frequency of node i during the first n steps of the random walk. It is known that the $p_{i,n}$ converge to so-called stationary probabilities π_i and that the stationary probabilities satisfy a system of linear equations directly related to the transition graph. For each node j there is an equation expressing π_j as a sum over all edges directed into j. The contribution to this sum of an edge (i, j) is $q_{ij} \cdot \pi_i$, where q_{ij} is the transition probability of the edge. In our example we obtain:

$$\begin{aligned} \pi_0 &= 2/3 \cdot \pi_0 + 1/2 \cdot \pi_1 \\ \pi_1 &= 1/3 \cdot \pi_0 + 1/2 \cdot \pi_1. \end{aligned}$$

This system has solution $\pi_0 = 6/10$ and $\pi_1 = 4/10$. In Table 3.5 we see the convergence of the visit frequencies to the stationary probabilities.

Dynamic Random Variates: We generalize the class *random_variate* to a class called *dynamic_random_variate* which offers an additional operation *set_weight* that allows the user to change weights dynamically. More precisely, if R is a dynamic random variate with weight vector w and i is in the index range of w then *set_weight*(i, g) changes $w[i]$ to g; g is an arbitrary non-negative integer. The generation process of dynamic random variates is less efficient than the one for (static) random variates; it takes time $O(\log n)$, where n is the size of the index range of w.

The implementation is fairly simple. We put the weights into the leaves of a balanced binary tree with n leaves and $n - 1$ internal nodes. In each node we store the sum of the weights of the leaves in its subtree. In particular, $W = \sum_i w[i]$ is stored in the root of the tree. A weight change amounts to updating the weights along one leaf to root path. In order to generate a random variate we choose a random integer s in $[0 \,..\, W - 1]$. If s is less than the total weight of the left subtree, we proceed recursively to the left subtree and if s is larger or equal to the total weight of the left subtree, we subtract the weight of the left subtree and proceed recursively to the right subtree. In this way, changing a weight and generating a random variate takes time proportional to the height of the tree. If a balanced tree is used the height is $O(\log n)$.

A particularly simple implementation results when the nodes of the tree are numbered

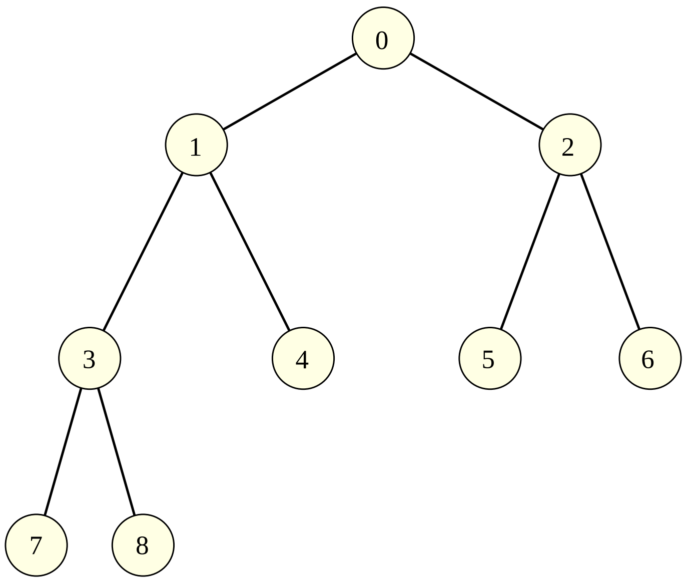

Figure 3.7 A tree with five leaves and a total of nine nodes. The number of each node is shown. The children of node i have numbers $2i$ and $2i+1$.

with the integers 1 to $2n-1$ in preorder, i.e., the root is given the number 1, the children of the node with number i, $1 \le i < n$ have numbers $2i$ and $2i+1$, and the leaves are numbered n to $2n-1$. See Figure 3.7 for an example. The parent of node i, $2 \le i \le 2n-1$ has number $\lfloor i/2 \rfloor$.

In the implementation we use an *array<int> u* with index range $[1 .. 2n-1]$ to store the tree.

⟨*definition of class dynamic_random_variate*⟩≡

```
class dynamic_random_variate{
private:
  array<int> u;
  int n, h, l;
public:
  dynamic_random_variate(const array<int>& w)
  { ⟨dynamic random variate: constructor⟩ }
  int generate()                { ⟨dynamic random variate: generate⟩ }
  int set_weight(int i, int g) { ⟨dynamic random variate: set weight⟩ }
};
```

The constructor stores the weight vector w in the entries n to $2n-1$ of u and then fills each entry u_i, $n-1 \ge i \ge 1$ as the sum of the entries of its children.

⟨*dynamic random variate: constructor*⟩≡

```
l = w.low(); h = w.high(); n = h - l + 1;
u = array<int>(1,2*n - 1);
int i;
for (i = 0; i < n ; i++)
{ u[n + i] = w[l + i];
  if ( u[n + i] < 0 ) error_handler(1,"dynamic variate: negative weight");
}
for (i = n - 1; i > 0; i--)
  u[i] = u[2*i] + u[2*i + 1];
if (u[1] == 0 ) error_handler(1,"dynamic variate: no non-zero weight");
```

The generator chooses a random integer s in $[0\,..\,W-1]$ and then walks down a path in the tree. When the walk reaches node i, s is a random integer in $[0\,..\,u[i]-1]$. If i is a leaf we return $l+(i-n)$ since the leaf numbered i corresponds to entry $l+(i-n)$ of weight vector w. If i is not a leaf and $s < u[2i]$, we proceed to child $2i$ and if $s \geq u[2i]$, we subtract $u[2i]$ from s and proceed to child $2i+1$.

⟨*dynamic random variate: generate*⟩≡

```
int s = rand_int(0,u[1] - 1);
int i = 1;
while ( i < n )
{ int j = 2*i;
  if ( s < u[j] )
    i = j;
  else
    { i = j + 1;
      s -= u[j];
    }
}
return l + i - n;
```

In order to change weight i to g we walk the path from leaf $n+(i-l)$ to the root and change all entries of u along the path by $delta = g - u[i]$. The old value of $u[i]$ is returned.

⟨*dynamic random variate: set weight*⟩≡

```
int ui = u[i];
i = n + (i - l);
int delta = g - u[i];
if ( g < 0 ) error_handler(1,"dynamic variate: negative weight");
while (i > 1)
{ u[i] += delta;
  i = i/2;
}
u[1] += delta;
if ( u[1] == 0 ) error_handler(1,"dynamic variate: no positive weight");
return ui;
```

n	Static	Dynamic
100	32.02	52.7
10000	41.07	90.34

Table 3.6 Running time of random variate generation: We set up a weight vector with n entries and then generated 10^7 random variates according to it. We used classes *random_variate* and *dynamic_random_variate*.
You can make your own experiments using the random_variate_demo.

Table 3.6 illustrates the speed of our two methods for generating random variates. Surprisingly, the $O(\log n)$ method is faster than the constant time method.

Dynamic Markov Chains: The use of dynamic random variates instead of static random variates in Markov chain data type yields a dynamic Markov chain data type which also supports the change of edge weights.

Simulating a Supermarket Check-Out: We use dynamic random variates to simulate a supermarket check-out. We consider a supermarket with n check-out stations. We assume that there is a queue (maybe empty) in front of every check-out station and use $q[i]$ to denote the queue length in front of the i-th check-out station. Servicing a customer at a check-out station takes either 1 (probability $2/3$) or 2 (probability $1/3$) time units. Thus the average servicing time is $4/3$ time units.

We assume that $3n/4$ customers arrive at every time unit. Customers tend to choose check-out stations with short queues. We assume that a customer chooses queue i with probability proportional to $1/(1+q[i])$.

In the program we define random variates R and S; S is a static random variate which models the distribution of service times and R is a dynamic random variate which yields check-out stations. In R we use $\lfloor M/(1+q[i])\rfloor$ as the weight of i, where M is a large constant. In each time step we first generate $3n/4$ customers. For each customer we choose the service length by calling *S.generate*( ) and the service station by calling *R.generate*( ). We update the queue lengths after each generation of a customer.

We collect all customers requiring short service in a list *short_service* and all customers requiring long service in a list *long_service*. After having generated the new customers we service all customers in *short_service*, update queue lengths appropriately, and move all customers in *long_service* to *short_service*.

⟨*supermarket check-out*⟩≡

```
array<int> q(n);
array<int> w(n);
int M = 10000;
for (int i = 0; i < n; i++) { q[i] = 0; w[i] = M; }
```

```
dynamic_random_variate R(w);
array<int> w1(1,2); w1[1] = 2; w1[2] = 1;
random_variate S(w1);
list<int> short_service, long_service;
for (int t = 0; t < T; t++)
{ for (int k = 0; k < 3*n/4; k++)
  { int i = R.generate(); q[i]++; R.set_weight(i,M/(1 + q[i]));
    if ( S.generate() == 1)
      short_service.append(i);
    else
      long_service.append(i);
  }
  int i;
  forall(i,short_service)
  { q[i]--; R.set_weight(i,M/(1 + q[i])); }
  short_service.clear();
  short_service.conc(long_service);
  ⟨report queue lengths⟩
}
```

Implementation of random_source: Our implementation of random sources follows the description in [Knu81, Vol2, section 3.2.2]. We first give the mathematics and then the program. Internally, we always generate a sequence of integers in the range $[0\mathinner{..}2^{31}-1]$. We define 32 unsigned longs $X_0, X_1, \ldots, X_{31}$ by

$$X_0 = \mathit{seed}$$

and

$$X_i = (1103515245 \cdot X_{i-1} + 12345) \bmod m$$

for $1 \le i \le 31$. Here $m = 2^{32}$. We extend this sequence by

$$X_i = (X_{i-3} + X_{i-32}) \bmod m$$

for $i \ge 32$. In this way an infinite sequence $X_0, X_1, \ldots$ of unsigned longs is obtained. Following [Knu81, Vol2, section 3.2.2], we discard the first 320 elements of this sequence (they are considered as a warm-up phase of the generator) and we also drop the right-most bit of each number (since it is the least random). Thus, the i-th number output by the internal generator is

```
(X[i + 320] >> 1) & 0x7fffffff.
```

We next show how to generate a number uniformly at random in $[\mathit{low}\mathinner{..}\mathit{high}]$. Let X be a number produced by the internal generator. Then $\mathit{low}+X \bmod(\mathit{high}-\mathit{low}+1)$ is a number in the range $[\mathit{low}\mathinner{..}\mathit{high}]$. However, this number is not uniformly distributed (consider the case where $\mathit{low} = 0$ and $\mathit{high} = 2^{31}-2$ and observe that in this case the number 0 is generated with probability twice as large as any other number). We therefore proceed differently. Our approach is based on the observation that if X is a random number in $[0 \mathrel{..} 2^{31}-1]$ and p is an integer less than 32 then $X \bmod 2^p$ is a random number in $[0 \mathrel{..} 2^p-1]$. Let $\mathit{diff} = \mathit{high}-\mathit{low}$ and let p be such that $2^{p-1} \le \mathit{diff} < 2^p$. We generate random numbers X using the internal source until $X \bmod 2^p \le \mathit{diff}$ and then output $\mathit{low} + X \bmod 2^p$. Since $2^{p-1} \le \mathit{diff}$ at most two X's have to be tried on average. The complete program follows.

⟨*generation of a random number in [low..high]*⟩≡

```
int diff = high - low;
/* compute pat = 2^p - 1 with 2^{p-1} <= diff < 2^p */
unsigned long pat = 1;
while (pat <= diff) pat <<= 1;
pat--;
/* pat = 0...01...1 with exactly p ones.
   Now, generate random x in [0 .. pat]
   until x <= diff and return low + x    */
unsigned long x = internal_source() & pat;
while ( x > diff) x = internal_source() & pat;
return (int)(low + x);
```

Exercises for 3.5

1 Add an operator $\gg$ to the type *random_source* that allows you to extract a random point in the two-dimensional unit square.

2 Consider the following program.

```
int i,j,x;
array<int> A(0,n-1);
for (i = 0; i < n; i++)
  { while (true)
      { x = rand_int(0,n-1);
        for (j = 0; j < i && x != A[j]; j++) ; // empty body
        if (j == i) break;
      }
    A[i] = x;
  }
```

a) Does it generate a random permutation of the integers 0 to $n-1$?
b) What is the expected running time of the program?

3 Change the random graph generator such that it generates graphs without self-loops, i.e., no edges (v, v), and without parallel edges, i.e., no two edges with the same source and target.

4 Let $d_0, \ldots, d_{n-1}$ be non-negative integers whose sum is even. Generate a random undirected graph where node i has degree d_i for all i, $0 \le i < n$. Hint: Create an array A of length $2m = \sum_i d_i$, write the integer i into d_i entries of A for all i, permute A, and then generate the edge $(A[2j], A[2j+1])$ for all j, $0 \le j < m$.

5 Balls and bins: Throw n balls randomly into m bins, i.e, choose n random integers in the range $[0 \mathrel{..} m-1]$ and tabulate how often each number is chosen. Perform the experiment

with $n = 10^6$ and $m = 100$, $m = 1000$, …, $m = 10^6$. If you want to understand the outcome of the experiment analytically consult [MR95].

6 Use classes *random_variate* and *so_set* to perform the following experiment. Let w be any vector of n non-negative integers with $w_0 \geq w_1 \geq \ldots \geq w_{n-1}$. Store the integers 0 to $n-1$ in a *so_set* and perform N access operations. For each i, $0 \leq i < n$ access i with probability proportional to w_i. Determine the total cost of all accesses where the cost of an access is the distance of the accessed item from the front of the list (you need to modify *so_set*::*member* slightly in order to get this information) and compare it to $C = N \sum_i w_i(i+1)/W$ where $W = \sum_i w_i$. Note that C is the expected cost of the accesses if the list were arranged in order of decreasing weight.

3.6 Pairs, Triples, and such

A tuple is an aggregation of variables of arbitrary types. LEDA offers two-tuples, three-tuples, and four-tuples. We use two-tuples as our running example in this section. For any types A and B and objects a and b belonging to these types the declarations

```
two_tuple<A,B> p;
two_tuple<A,B> q(a,b);
```

define a two-tuple p and a two-tuple q, respectively. The components of p are initialized to the default values of A and B, respectively, and the components of q are initialized with copies of a and b, respectively. The operations *first* and *second* return the two variables contained in a two-tuple. So we may write

```
a = p.first();
p.second() = b;
```

The operators $==$, $\ll$, $\gg$ and the functions *compare* and *Hash* are defined for two-tuples. They assume that the corresponding functions are defined for the component types. The operators $\ll$ and $\gg$ read and write a two-tuple, respectively, the operator $==$ realizes component-wise equality, *compare* amounts to the lexicographic ordering of two-tuples and *Hash* returns the bitwise exclusive or of the hash values of the components. All of these functions and operators are defined as template functions. For example,

```
template <class A, class B>
int compare(const two_tuple<A,B>& p, const two_tuple<A,B>& q)
{ int s = compare(p.first(),q.first());
  if (s != 0) return s;
  return compare(p.second(),q.second());
}
```

If one uses two-tuples in a situation that requires the compare function for two-tuples, e.g., if one defines a *list<two_tuple<int, int> > L* and then calls *L.sort( )*, it is wise to give the compiler a hint that it should make the compare function for *two_tuple<int, int>*. In the following program this is done by defining a variable p of type *two_tuple*<*int*, *int*> and calling *compare*(p, p).

⟨*two_tuple_test*⟩≡

```
main()
{ list< two_tuple<int,int> > L;
  two_tuple<int,int> p;
  compare(p,p); // dummy compare
  L.sort();
}
```

3.7 Strings

A *string* is a sequence of characters, where a character is an element of the C++ type *char*. The number of characters in a string is called the length of the string and the characters in a string are numbered starting at zero. So *u* is the character at position one in *Kurt*. The string of length zero is called the empty string; it is the default value of the type. Strings are related to the *char*∗ type of C++. There are, however, two significant differences:

- The value of a variable of type string is a sequence of characters, it is not a pointer. In particular, assignment and parameter passing by value work properly for strings. Strings are a primitive type, see Section 2.3.
- Strings offer a large number of additional operations, e.g., pattern matching, substring replacement, and comparison according to the lexicographic ordering. We have to admit, however, that some programming languages, e.g., PERL and AWK, offer much more elaborate string classes.

Let us see strings at work.

```
string s("Stefan");
```

defines a string variable s and initializes it with the value `"Stefan"`.

```
string t = s + s;
```

defines another string variable t and initializes it to `"StefanStefan"`; the operator $+$ is the concatenation operation on strings. The expression $t(2, 5)$ returns the substring of t starting at position 2 and ending at position 5. Since we start counting at 0 this is the string `"efan"`. We can also search for the occurrence of one string in another string: If a and b are strings then *a.pos*(b) searches for an occurrence of b in a. If b does not occur in a then *pos* returns -1 and if b does occur then it returns the first position in a at which b occurs. Thus

```
t.pos("efa");
```

returns 2, i.e., the first position in t at which an occurrence of `"efa"` starts, and *t.pos*(“*Kurt*”) returns -1. Another useful operation on strings is substring replacement. It comes in several forms: *a.replace*(i, j, b) returns $a(0, i-1) + b + a(j+1, a.length(\,) - 1)$, i.e., b is substituted for the substring $a(i, j)$, and *a.replace*$(b1, b2, n)$ replaces the n-th occurrence of $b1$ in a by $b2$, and finally *a.replace_all*$(b1, b2)$ replaces all occurrences of $b1$ in a by $b2$. It is important to notice that all three versions do not change the string a. Rather, they return a new string. So after

```
string u = t.replace(2,5,"Kurt");
string v = t.replace(s,"Kurt",2);
```

we have a string u with value `"StKurtStefan"`, i.e., the substring of t starting at position 2 and ending at position 5 is replaced by `"Kurt"`, and a string v with value `"StefanKurt"`, i.e., the second call of *replace* returns a string in which the second occurrence of s in t is replaced by `"Kurt"`.

The operator $<$ realizes the lexicographic ordering of strings. So

```
(t < (s + s + s ));
```

evaluates to true since `"StefanKurt"` precedes `"StefanStefanStefan"` in the lexicographic ordering of strings. Many other operations on strings can be found in the manual.

Strings are implemented by C++ character vectors. All operations on strings that do not involve pattern matching take linear time. Pattern matching takes quadratic time. More precisely, it takes time $O(nm)$ in the worst case to search for a string of length m in a string of length n. There are $O(n+m)$ pattern matching algorithms, see for example [CLR90].

3.8 Making Simple Demos and Tables

This book contains many tables. For many of these tables there is also a corresponding demo which allows the reader to perform experiments on his or her own. We wanted to have a single program that handles both cases. In this section we describe the IO-interface used in these programs.

The program below serves as the random_variate_demo and also produces Table 3.6. It makes all its input and output through *IO_interface I*. The program can be executed in two modes: in book-mode it produces a table[2] and in demo-mode it realizes the random_variate_demo. The demo-mode is the default and the book-mode is selected at compile-time by compiling with the flag `-DBOOK`[3].

[2] This book is typeset using LaTeX and hence the program generates a sequence of LaTeX-commands that produce a table.

[3] An alternative design would be to use an integer variable to distinguish between the cases and set the variable through a command line argument.

⟨*random_variate_demo.c*⟩≡
```
#include <LEDA/random_variate.h>
#include <LEDA/IO_interface.h>
main()
{ IO_interface I("Random Variates");
  I.write_demo("This demo illustrates the speed of classes \
random variate and dynamic random variate. \nYou will be asked \
to input integers n and N. We set up the weight vector w with \
w[2] = 2, w[3] = 3, ..., w[n+1] = n + 1 and generate N random \
variates according to this weight vector.");
  int n, N;
  n = I.read_int("n = ",100);
  N = I.read_int("N = ",100000);
  if ( n < 1 ) error_handler(1,"n must be at least one");
#ifdef BOOK
N = 10000000;
for (n = 100; n <= 10000; n = n*n)
{ I.write_table("\n ", n);
#endif
  array<int> w(2, 1 + n);
  array<double> Rfreq(2,n+1), Qfreq(2,n+1);
  int W = 0; int i;
  for (i = 2; i < n + 2; i++) { W += w[i] = i; Qfreq[i] = Rfreq[i] = 0; }
  dynamic_random_variate R(w);
  random_variate Q(w);
  float T = used_time(); float UT;
  for (i = 0; i < N; i++) Qfreq[Q.generate()]++;
  UT = used_time(T);
  I.write_demo("static random variate, time = ",UT);
  I.write_table(" & ",UT);
  for (i = 0; i < N; i++) Rfreq[R.generate()]++;
  UT = used_time(T);
  I.write_demo("dynamic random variate, time = ",UT);
  I.write_table(" & ",UT, " \\\\ \\hline");
  I.write_demo("We report some frequencies.");
  for (i = n + 1; i >= Max(2,n - 3); i--)
  { I.write_demo("relative frequency, i = ",i);
    I.write_demo(0,", w[i]/W = ",((double)w[i])/W);
    I.write_demo(1,"generated freq, static variate = ",  Qfreq[i]/N);
    I.write_demo(1,"generated freq, dynamic variate = ", Rfreq[i]/N);
  }
#ifdef BOOK
}
#endif
}
```

The output statements come in two kinds: *write_table* and *write_demo*. The output statement *write_xxx* produces output when executed in *xxx*-mode and produces no output otherwise. Thus, the introductory text that explains the demo is output in demo-mode, but is suppressed in book-mode. The output statements come in different forms:

```
I.write_xxx(string mes);
I.write_xxx(string mes, double T, string mes2 = "");
I.write_xxx(string mes, int T,    string mes2 = "");

I.write_xxx(int k, string mes);
I.write_xxx(int k, string mes, double T, string mes2 = "");
I.write_xxx(int k, string mes, int T,    string mes2 = "");
```

The first form outputs the string *mes* and the second and the third form output the string *mes*, followed by the number T, followed by the optional string *mes2*. The output is preceded by an empty line. The last three forms allow a finer control over the positioning of the output; the output is preceded by k line feeds, i.e., with $k = 0$ the output is printed on the same line as the previous output, with $k = 1$ the output is printed on a new line, and with $k = 2$ the output is preceded by an empty line.

The input statement

```
int I.read_int(string mes, int n = 0);
```

returns n in book-mode and asks for an integer input with prompt *mes* in demo-mode.

The precision of the output of double-values is controlled by a precision parameter p. It is set to 4 by default and can be changed by

```
I.set_precision(int prec);
```

We come to the implementation. It is quite simple. We define classes *IO_interface_book* and *IO_interface_demo* in the obvious way (see LEDAROOT/incl/LEDA/IO_interface.h) and define *IO_interface* as one of them depending on the compile-time flag.

⟨*IO_interface*⟩≡

```
#ifdef BOOK
#define IO_interface IO_interface_book
⟨definition of IO_interface_book⟩
#else
#define IO_interface IO_interface_demo
⟨definition of IO_interface_demo⟩
#endif
```

Bibliography

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

[HP90] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[Knu81] D.E. Knuth. *The Art of Computer Programming (Volume II): Seminumerical Algorithms*. Addison-Wesley, 1981.

[KSK76] G. Kemeny, L. Snell, and A.W. Knapp. *Denumerable Markov Chains*. Springer, 1976.

[Meh84] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer, 1984.

[Moo61] E.F. Moore. U.S. Patent 2983904, 1961.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[Wal77] A.J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Transaction on Mathematical Software*, 3:253–256, 1977.

Index