Contents

13

On the Implementation of LEDA

This chapter deals with the implementation of LEDA. It gives the details of the implementation of parameterized data types, implementation parameters, handle types, the memory management, and iteration macros. We close the chapter with a comprehensive example that illustrates all concepts discussed.

13.1 Parameterized Data Types

The definition of parameterized data types of LEDA has been discussed in Chapter 2. In the next sections we describe how they are implemented. We first describe the C++ template approach to parameterized data types using a simple list data type. Then we use the same example to explain the basic idea of the LEDA solution for implementing parameterized data types and discuss the reasons for choosing this solution. Finally, we extend the basic solution and apply it to more advanced data types and develop optimizations for the case where the actual type parameters are small (fit into one memory word) or are basic built-in types.

13.2 A Simple List Data Type

We start this section by giving a very simple implementation for a data type *list* of singly linked lists of integers. It offers about the same set of operations as the LEDA *stack* data type. There is a *push* operation that inserts a given integer at the front of the list and a *pop* operation that removes the first element from the list and returns it. Operation *head* returns the first element without changing the list, and finally, operation *size* returns the number of elements of the list. Of course, we also have to provide a constructor, destructor, copy constructor and an assignment operator in order to make *list* a fully equipped C++ data type.

Note that we use this simple type only as a first example for introducing some aspects of the LEDA mechanism for implementing parameterized data types. Of course, LEDA contains much more powerful and useful list types, see Section 3.2.

As usual, the declaration (or specification) of our list class is contained in a header file called *_list.h* and the implementations of its operations are contained in a separate source code file *_list.c*. We let the file names start with _, because we want to use the file names without the underscore later in the section.

The header file *_list.h* might look as follows:

⟨*_list.h*⟩≡

```
class list {

  struct list_elem
  { // a local structure for representing the elements of the list
    int entry;
    list_elem* succ;
    list_elem(const int& x, list_elem* s) : entry(x), succ(s) {}
    friend class list;
  };

  list_elem* hd;  // head of list
  int        sz;  // size of list

public:

  void push(int);
  void pop(int&);
  int  head() const;
  int  size() const;

  list();
 ~list();
  list(const list&);
};
```

The corresponding source code file *_list.c* is as follows:

⟨*_list.c*⟩≡

```
#include "_list.h"
#define NULL 0

int list::head() const { return hd->entry; }

void list::push(int x)
{ hd = new list_elem(x,hd);
  sz++;
}

void list::pop(int& y)
{ y = hd->entry;
  list_elem* p = hd;
  hd = p->succ;
  delete p;
  sz--;
}
list::list()
{ // construct an empty list
  hd = NULL;
  sz = 0;
}
list::list(const list& L)
{ // construct a copy of L
  hd = NULL;
  sz = L.sz;
  if (sz > 0)
  { hd = new list_elem(L.hd->entry,0); // first element
    list_elem* q = hd;
    // subsequent elements
    for (list_elem* p = L.hd->succ; p != NULL; p = p->succ)
    { q->succ = new list_elem(p->entry,NULL);
      q = q->succ;
    }
  }
}
list::~list()
{ // destroy the list
  while (hd)
  { list_elem* p = hd->succ;
    delete hd;
    hd = p;
  }
}
```

13.3 The Template Approach

Most data types in LEDA are parameterized. LEDA does not only offer lists of integers but lists of an arbitrary element type E. In this section we discuss the C++ standard approach to parameterized data types. We explain the approach and discuss why we have not taken it in LEDA. The solution which we adopted in LEDA is described in the next section.

C++ supports parameterized classes by means of its *template feature*. How can one obtain lists of *char* from our implementation of lists of *int*? It seems to be very simple. Replace in files list.h and list.c all occurrences of *int* by *char*. Well, that's not quite true. Actually, we should replace only those occurrences of *int* that refer to the element type of the list. So the declarations of variable *sz* and the return type of *size*( ) stay unchanged. Since it is completely mechanical to derive list of characters from lists of integers we might as well ask the compiler to do it. All we have to do is to mark those occurrences of *int* that are to be replaced. The template feature of C++ is an elegant way to automate this transformation. The following simple textual transformation changes the definition of our list class into the definition of a parameterized list class:

- Replace in list.h all occurrences of *int* that refer to the element type of our lists by a new class name, say *E*.
- Prefix the definition of class *list* in the file list.h and the definition of each member function in the file list.c by *template <class E>*. This informs the compiler that *E* is the name of a type parameter and not the name of a concrete type.
- Replace in list.c all occurrences of *list* that refer to the name of the list class by *t_list<E>*. This replacement is not really necessary. We make it so that we can later contrast classes *list* and *t_list*.

For concreteness, we include excerpts from the modified files t_list.h and t_list.c.

⟨*t_list.h*⟩≡

```
template <class E>
class t_list {

  struct list_elem
  { E entry;
    list_elem* succ;
    list_elem(const E& x, list_elem* s) : entry(x), succ(s) {}
  };

  list_elem* hd;  // head of list
  int        sz;  // size of list

public:

  void push(const E&);
  void pop(E&);
  const E& head() const;
  int  size() const;

  t_list();
  t_list(const t_list<E>&);
 ~t_list();
};
```

and file t_list.c is as follows:

⟨*t_list.c*⟩≡

```
#include "t_list.h"
#define NULL 0

template <class E> t_list<E>::t_list()
{ hd = NULL;
  sz = 0;
}

template <class E> const E& t_list<E>::head() const
{ return hd->entry; }

template <class E> void t_list<E>::push(const E& x)
{ hd = new list_elem(x,hd);
  sz++;
}
```

```
template <class E> void t_list<E>::pop(E& y)
{ y = hd->entry;
  list_elem* p = hd;
  hd = p->succ;
  delete p;
  sz--;
}
```

In an application program we can now write

```
t_list<char> L1;
t_list<segment> L2;
```

to define a list *L1* of *char* and a list *L2* of line segments, respectively. When the compiler encounters these definitions it constructs two versions of files list.c and list.h by substituting *E* by *char* and by *segment*, respectively, which it can then process in the standard way. Let us summarize:

- The template feature is powerful and elegant. The implementer of a data type simply prefixes his code by *template<class E>* and otherwise writes his code as usual, and the user of a parameterized data type only needs to specify the actual type parameter in angular brackets.
- The template feature duplicates code. This increases code length and compilation time. It has to duplicate code because the layout of the elements of a list in memory (type *list_elem*) depends on the size of the objects of type *E* and hence the code generating new list elements depends on the size of the objects of the actual type parameter.
- Separate compilation is impossible. Since the code to be generated depends on the actual type parameter one cannot precompile t_list.c to obtain an object file t_list.o. Rather both files t_list.h and t_list.c have to be included in an application and have to be compiled with the application. For an application, that uses many parameterized data types from the library, this leads to a large source and therefore large compilation times. Moreover, it forces the library designer to make his .c-files public.
- When we started this project, most C++ compilers did not support templates and, even today, many do not support them fully. Some compilers use repositories of precompiled object code to avoid multiple instantiations of the same template code. However, there is no standard way for solving this problem.

We found in particular the drawback of large compilation times unacceptable and therefore decided against the strategy of implementing parameterized data types directly by the template feature of C++.

The LEDA solution uses templates in a very restricted form. It allows separate compilation, it allows us to keep the .c-files private, and it does not over-strain existing C++ compilers. We discuss it in the next section.

Let us summarize. The template feature is an elegant method to realize parameterized data types (from a user's as well as an implementor's point of view). However, it also has a certain weakness. It duplicates code, it does not allow us to precompile the data types, and it is only partially supported by compilers.

13.4 The LEDA Solution

In LEDA every parameterized data type is realized by a pair of classes: a *class for the abstract data type* and a *class for the data structure*, e.g., we have a dictionary class (= the data type class) and a binary tree class (= the data structure class). Only the data type classes use the template mechanism. All data type classes are specified in the header file directory *LEDAROOT/incl/LEDA* and only their header files are to be included in application programs. All data structures are precompiled into the object code libraries (*libL, libG, libP, libW, ...)* and are linked to application programs by the C**++** linker. Instead of abstract data type class we will also say data type class or data type template or abstract class and instead of data structure class we will also say implementation class or concrete class.

Precompilation of a data structure is only possible if its implementation does not depend on the actual type parameters of the corresponding parameterized data type. In particular:

- the layout of the data structure in memory must not depend on the size of the objects stored in it. We achieve this (in a first step) by always storing pointers to objects instead of the objects themselves in our data structures. Observe that the space requirement of a pointer is independent of the type of the object pointed to. In a second step (cf. Section 13.5.1) we show how to avoid this level of indirection in the case of small types (types whose size in memory is at most the size of a pointer).

- all functions used in the implementation whose meaning depends on the actual type parameters use the dynamic binding mechanism of C**++**, i.e., are realized as virtual functions. A prime example is the comparison function in comparison based data structures. The comparison function is defined as a virtual member function of the implementation class, usually called *cmp_key*. In the definition of the abstract data type template we bind *cmp_key* to a function *compare* that defines the linear order on the actual type parameter.

The remainder of this section is structured as follows. We first give the basic idea for parameterized data types in LEDA. Then we discuss the use of virtual functions and dynamic binding for the implementation of assignment, copy constructor, default constructor, and destruction. In the sections to follow we describe an improvement for so-called one-word or small types, and show how implementation parameters are realized. Finally, we give the full implementation of priority queues by Fibonacci heaps and illustrate all features in one comprehensive example.

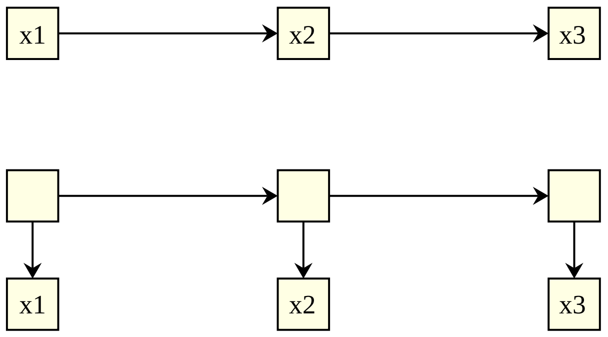

Figure 13.1 A *t_list* and a *list*: The top part shows a *t_list*<*E*> with three elements x_1, x_2, x_3. The bottom part shows the corresponding *list* data structure in the LEDA approach.

13.4.1 ***The Basic Idea***

We introduce the basic idea for realizing parameterized data types in LEDA, the idea will be refined in later sections:

- The data fields in the containers of all data structures are of type *void*∗, the generic pointer type of C++. They contain pointers to objects of the actual type parameters. Consider a data structure whose containers have a slot for storing objects of a type T, e.g., the type *t_list*<T>.
- In the LEDA approach the objects of type T are not stored directly in the containers of the data structure but on the heap. The data slots of the containers have type *void*∗, the generic pointer type of C++, and contain pointers to the objects on the heap. More precisely, if a container has a slot of type T in the template solution and t is the object stored in it (at a particular time) then the corresponding container in the LEDA solution will have a field of type *void*∗ and this field will contain a pointer to t. See Figure 13.1 for an illustration.
- The abstract data type class uses the template mechanism and is derived from the implementation class.
- Type casting is used to bridge the gap between the untyped world of the implementation class (all data is *void*∗) and the typed world of the abstract class.

We use our singly linked list data type as a first example to illustrate our approach. We saw an implementation of lists, called *t_list*, using the template approach in the preceding section.

Our goal is to realize the parameterized data type *list*<T> by a concrete data structure *list_impl* that stores pointers of type *void*∗. The definition of *list_impl* is straightforward. It is essentially a list of type *t_list*<*void* ∗>

⟨*list_impl.h*⟩≡

```
class list_impl {

  struct list_impl_elem
  { void* entry;
    list_impl_elem* succ;
    list_impl_elem(void* x,list_impl_elem* s):entry(x),succ(s) {}
    friend class list_impl;
  };

  list_impl_elem* hd;
  int sz;

 protected:

  list_impl();
 ~list_impl();

  void* head() const;
  void* pop();
  void  push(void* x);
  void  clear();
  int   size() const;
};
```

and

⟨*list_impl.c*⟩≡

```
#include "list_impl.h"

list_impl::list_impl() : hd(0), sz(0) {}

list_impl::~list_impl() { clear(); }

void* list_impl::head() const { return  hd->entry; }

void list_impl::push(void* x)
{ hd = new list_impl_elem(x,hd);
  sz++;
}

void* list_impl::pop()
{ void* x =  hd->entry;
  list_impl_elem* p = hd;
  hd = p->succ;
  delete p;
  sz--;
  return x;
}

void list_impl::clear() { while (hd) pop(); }

int  list_impl::size() const { return sz; }
```

We declared the member functions of *list_impl* protected so that they can only be used in derived classes. We can now easily derive the data type template *list*<*T*> for arbitrary types *T* from *list_impl*. We make *list_impl* a private base class of *list*<*T*> and implement the member functions of *list*<*T*> in terms of the member functions of the implementation class.

Making the implementation class a private base class makes it invisible to the users of the *list*<*T*> class. This guarantees type safety as we argue at the end of the section.

A member function of *list*<*T*> with an argument of type *T* first copies the argument into the dynamic memory (also called heap), then casts a pointer to the copy to *void*∗, and finally passes the pointer to the corresponding function of the implementation class.

All member functions of *list*<*T*> that return a result of type *T* call the corresponding function of the implementation class (which returns a result of type *void*∗), cast the pointer to *T*∗, and return the dereferenced pointer.

We next give the details.

```
template<class T>
class list : private list_impl {
public:
  list() : list_impl() {}
```

The constructor of *list*<*T*> constructs an empty *list_impl*.

```
    void push(const T& x) { list_impl::push((void*) new T(x)); }
```

L.push(x) makes a copy of x in dynamic memory (by calling the copy constructor of T in the context of the *new* operator) and passes a pointer to this copy (after casting it to *void*∗) to *list_impl*::*push*. The conversion from $T*$ to *void*∗ is a built-in conversion of C++ and hence we may equivalently write

```
    void push(const T& x) { list_impl::push(new T(x)); }
```

Let us relate *list*<*T*>::*push*(x) to *t_list*<*T*>::*push*(x). The latter operation stores a copy of x directly in the entry-field of a new list element and the former makes a copy of x on the heap and stores a pointer to the copy in entry-field.

```
    const T&  head() const { return *(T*)list_impl::head(); }
```

L.head( ) casts the *void*∗ result of *list_impl*::*head*( ) to a $T*$ pointer, dereferences the result, and returns the object obtained as a const-reference. It thus returns the element of the list that was pushed last.

```
    T pop()
    { T* p = (T*)list_impl::pop();
      T x = *p;
      delete p;
      return x;
    }
```

L.pop(x) casts and dereferences the *void*∗ result of *list_impl*::*pop*, assigns it to a local variable x, deletes the copy (made by *list*<*T*>::*push*), and returns x. Observe that the assignment to x makes a copy, and it is therefore OK to delete the copy made by *push*. It is also necessary to delete it, as we would have a memory leak otherwise.

```
    int size() const { return list_impl::size(); }
    void clear()
    { while (size() > 0) delete (T*)list_impl::pop();
    list_impl::clear();
  }
 ~list() { clear(); }
};
```

The implementations of *clear* and of the destructor are subtle. *Clear* first empties the list and then calls *list_impl*::*clear*. The latter call is unnecessary as popping all elements from the list already has the effect of clearing the list. We make the call for reasons of uniformity (all *clear* functions of abstract classes in LEDA first destroy all objects contained in the data structure and then call the *clear* function of the implementation). It is, however, absolutely vital to destroy the objects stored in the list before calling *list_impl*::*clear*. An implementation

```
void clear()      { list_impl::clear(); }
```

has a memory leak as it leaves the elements contained in the list as orphans on the heap.

The destructor first calls *clear* and then the destructor of the base class (the latter call being automatically inserted by the compiler). The base class destructor ∼*list_impl* deletes all list elements. Observe that it does not suffice to call this destructor as this will leave all entries contained in the elements of the list on the heap.

If our list implementation class would support iteration in the LEDA *forall* style an alternative implementation of the clear function would be

```
void clear()
{ void* p;
  forall(p,*this) delete (T*)p;
  list_impl::clear();
}
```

Let us assess our construction:

- The construction is non-trivial. Please read it several times to make sure that you understand it and try to mimic the approach for other data types (see the exercises). The construction is certainly more complicated than the pure template approach presented in the preceding section.
- The data type *list*<*T*> simulates the data type *t_list*<*T*>. Suppose that we perform the same sequence of operations on a *list*<*T*> *S* and a *t_list*<*T*> *TS*. Assume that $x_0, \ldots, x_{t-1}$ are the entries of *TS* after performing the sequence. Then *S* also has t elements and the corresponding entries contain pointers to copies of x_0 to x_{t-1} in dynamic memory, see Figure 13.1.
- All operations of *list*<*T*> are implemented by very simple inline member functions. Except for *pop*, *clear*, and ∼*list*( ) they do not produce any additional code. We will show in the next section how the code for *pop* and *clear* can also be moved into the data structure by the use of virtual functions. This will make the definition of the abstract class cleaner.

- The implementation class can be precompiled; see below.
- The above implementation of lists is incomplete. In particular, the definitions of the copy constructor and of the assignment operator are missing. We will discuss them in Section **??**.

The abstract data type *list*$<T>$ can be used in the usual way.

```
list<string> L;
L.push("fun");
L.push("is");
L.push("LEDA");
while (L.size() > 0) cout << L.pop() << endl;
```

Separate Compilation: We defined two classes, the implementation class *list_impl* and the abstract data type class *list*$<T>$, in three files: the file *list_impl.h* contains the skeleton of the class definition of *list_impl*, namely the definition of the private data of the class and the declarations of the member functions, the file *list_impl.c* contains the implementation of all member functions of *list_impl* and the file *list.h* contains the definition of the abstract data type and its implementation in terms of the implementation class. We have shown all three files above. It is still worthwile to repeat their global structure.

⟨*list_impl.h* ⟩≡
```
class list_impl {
   ⟨definition of private data⟩
protected:
   ⟨declaration of member functions⟩
};
```

The file *list_impl.c* contains the implementations of all member functions. It must include *list_impl.h*

⟨*list_impl.c* ⟩≡
```
#include <list_impl.h>
⟨implementation of all member functions⟩
```

The abstract class template *list*$<T>$ is defined in file *list.h*. It is derived from class *list_impl* and all member functions of the abstract class are realized by calling the corresponding member function of the implementation class as described above. The calls also do the appropriate type conversions from type T to *void*$*$ and vice versa. Since class *list_impl* is only used to implement its derived classes *list*$<T>$ it is qualified as a private base class of the list template. Of course, we have to include *list_impl.h* before using *list_impl* as a base class.

⟨*list.h*⟩≡

```
#include "list_impl.h"
template<class T>
class list : private list_impl {
public:
  void push(const T& x)  { list_impl::push(new T(x)); }
  const T&  head() const { return *(T*)list_impl::head(); }
  void pop(T& x)
  { T* p = (T*)list_impl::pop();
    x = *p;
    delete p;
  }
  int size() const       { return list_impl::size(); }
  void clear()
  { while (size() > 0) delete (T*)list_impl::pop();
    list_impl::clear();
  }
  list() : list_impl() {}
 ~list() { clear(); }
};
```

The file *list_impl.c* can be compiled into the object code file *list_impl.o*. An application program, say *list_prog.c*, using lists needs to include *list.h* and can be compiled separately into file *list_prog.o*. Finally, *list_prog.o* and *list_impl.o* can be linked to an executable program.

In the LEDA system the header files of implementation classes are collected in the directory *LEDAROOT/incl/LEDA/impl* and the header files of abstract classes are collected in *LEDAROOT/incl/LEDA*. All .c-files are contained in the various subdirectories of *LEDAROOT/src*.

Type Safety: We next comment on the type safety of the construction described above. The implementation class *list_impl* is untyped in the sense that anything can be pushed onto a list of type *list_impl*, the class *list* is typed in the sense that only objects of type T can be pushed onto a list of type *list*$<T>$. In the definition of class *list* we make the transition from the safer (typed) world to a potentially unsafer (untyped) world. Since we declared all operations of *list_impl* protected and made *list_impl* a private base class of *list*, the untyped world is completely encapsulated inside class *list* and invisible to any application program. Only the implementation class *list_impl* works in the untyped world; we designed it carefully so as to avoid the dangers of the untyped world. We conclude that the construction is type safe.

Efficiency: The construction is also efficient. Note that no code needs to be generated for the type conversions; the casts simply tell the compiler how the entries of the list are to be interpreted. Also all member functions of the abstract class are trivial inline functions and their calls can be eliminated by optimizing compilers, i.e., there is, for example, no need first to call the abstract function *list*::*push* which in turn calls the concrete function *list_impl*::*push*. The compiler will directly call the concrete function.

Genericness: Finally, the construction is elegant, although not as elegant as the solution relying completely on templates. The definition of the implementation class is completely natural[1], it is essentially *t_list*<*void*∗>. The definition of the abstract class in terms of the implementation class is somewhat inelegant because of the required type conversions. However, these type conversions follow a very simple rule. In-going values are converted to *void*∗ and return-values are converted back to type T.

13.4.2 *Virtual Functions and Dynamic Binding*

In the example of the preceding section the implementation class *list_impl* required no knowledge about the actual type argument of the data type template *list*<T>. This is an exceptional situation; in most situations the implementation needs to have some knowledge about the actual type argument. We give two examples.

The first example is a print operation for our list type that prints all elements to the standard output. We want to realize this operation by a *print* member function in the implementation class *list_impl*. Of course, this function needs to know how to print an object of the actual type parameter. The second example is comparison-based implementations of dictionaries, e.g., binary search trees. Any comparison-based implementation of the parameterized data type *dictionary*<K, I> (cf. Section 5.3) needs to know how to compare keys. In LEDA, the linear order on a key type K is defined by a global function *int compare*(*const* K&, *const* K&) (cf. Section 2.10) and hence the implementation class must be able to call this function.

In both examples we need a mechanism to transfer functionality of the actual type parameters from the abstract data type template to the implementation class. The appropriate C++ feature is dynamic binding and virtual functions. Detailed discussions of this concept can be found in [Str91, ES90]. The following should be clear even without prior knowledge of the concept.

In the first example, the class *list_impl* uses a virtual function *print_elem*(*void* ∗ p) to print elements to standard output. This function is declared in the implementation class but its implementation is left undefined by labeling it as pure virtual. Syntactically, pure virtual functions are designated by the key word *virtual* and the assignment "=0" which replaces the body. The implementation class may use the virtual function in its other member functions, e.g., *list_impl* uses *print_elem* in a function *print* that prints the entire list.

```
class list_impl {
   ...
virtual void print_elem(void*) const = 0;
   ...
void print() const
```

[1] You may want to include a *typedef void*∗ T; to make it look even more natural.

```
{ for(list_impl_elem* p = hd; p; p = p->succ) print_elem(p->entry); }
   ...
};
```

The implementation of *print_elem* is provided in the derived class *list*<T>. It converts its argument from *void*∗ to $T*$ (observe that this conversion makes sense on the level of the data type template) and then hands the object pointed to to the output operator ($\ll$) of type T (assuming that this operator is defined for T).

```
template<class T>
class list : private list_impl {
   ...
 void print_elem(void* p) const { cout << *(T*)p << endl; }
 void print() const { list_impl::print(); }
   ...
};
```

When a list is created, say through the declaration *list*<*char* ∗ > L; the definition of *print_elem* in terms of *operator* $\ll$ (*ostream*&, *char*∗) is associated with L. In a call L.*print*( ) which leads via *list_impl*::*print*( ) to a call of *list_impl*::*print_elem* the implementation of *print_elem* bound to L is used. In this way, information about the actual type parameter is transported into the implementation class.

We turn to our second example. All implementations of *dictionary*<K, I> use a virtual member function *int cmp_key*(*void*∗, *void*∗) for comparing keys. We discuss the implementation class *bin_tree*. As in the previous example, *cmp_key* is declared as pure virtual in the implementation class. In the derived class template *dictionary*<K, I> we define *cmp_key* in terms of the compare function of type K. We have

```
class bin_tree {
  ...
virtual int cmp_key(void*,void*) const = 0;
  ...
};
```

in the implementation class and

```
template<class K, class I>
class dictionary: private bin_tree {
  ...
int cmp_key(void* x, void* y) const { return compare(*(K*)x,*(K*)y); }
  ...
};
```

in the data type template (note the conversion from *void*∗ to $K*$ in the implementation of *cmp_key*).

The construction associates the appropriate compare function with every dictionary, e.g., *compare*(*const int*&, *const int*&) with *dictionary*<*int*, *int*>. Furthermore, the compare function is available in the implementation class *bin_tree* and can be called by its member functions (e.g. lookup).

In the remainder of this section and in the next section we give more details of the *bin_tree* class. This will allow us to discuss further aspects of the LEDA approach to parameterized data types.

The nodes of a *bin_tree* are realized by a class *bin_tree_node*. Each node contains a key and an information, both of type *void*∗, and additional data members for building the actual tree. For unbalanced trees the pointers to the two children suffice. For balanced trees additional information needs to be maintained. All implementations of balanced trees in LEDA are derived from the *bin_tree* class.

In the remainder of this chapter we will use the type name *GenPtr* for the generic pointer type *void*∗.

```
typedef void* GenPtr;
class bin_tree_node {
  GenPtr key;
  GenPtr inf;
  bin_tree_node* left_child;
  bin_tree_node* right_child;
  // allow bin_tree to access all members
  friend class bin_tree;
};
```

The class *bin_tree* contains some private data, such as a pointer to the root of the tree. The member functions realizing the usual dictionary operations are declared protected to make them accessible for derived classes (e.g., *dictionary*<K, I>) and the *cmp_key* function is declared a private pure virtual function. Finally, we define the item type (cf. Section 2.2.2) for class *bin_tree* (*bin_tree*::*item*) to be equal to type *bin_tree_node*∗.

```
class bin_tree {
private:
   bin_tree_node*  root;
   int cmp_key(GenPtr,GenPtr) const = 0;
protected:
   typedef bin_tree_node* item;
   item  insert(GenPtr,GenPtr);
   item  lookup(GenPtr) const;
   void  del_item(item);
   GenPtr key(item p) const { return p->key; }
   GenPtr inf(item p) const { return p->inf; }
   bin_tree();
  ~bin_tree();
};
```

The virtual *cmp_key* function is used to compare keys, e.g., in the *lookup* member function that returns a pointer to the node storing a given key k or *nil* if k is not present in the tree.

```
bin_tree_node* bin_tree::lookup(GenPtr k) const
{ bin_tree_node* p = root;
  while (p)
```

```
    { int c = cmp_key(k,p->key);
      if (c == 0) break;
      p = (c > 0) ? p->right_child : p->left_child;
    }
    return p;
  }
```

In the definition of the data type template *dictionary*$<K, I>$ we define *cmp_key* in terms of the compare function for type K. The dictionary operations are realized by calling the corresponding member functions of *bin_tree*. As in the list example, we also need to perform the necessary type conversions. The item type of dictionaries (*dic_item*) is defined to be equal to the item type of the implementation class *bin_tree*::*item*.

```
  typedef bin_tree::item dic_item;

  template<class K, class I>
  class dictionary : private bin_tree {

    int cmp_key(GenPtr x, GenPtr y) const
    { return compare(*(K*)x,*(K*)y); }

  public:

    const K& key(dic_item it) const { return *(K*)bin_tree::key(it); }

    const I& inf(dic_item it) const { return *(I*)bin_tree::inf(it); }

    dic_item insert(const K& k const I& i)
    { return bin_tree::insert(new K(k), new I(i)); }

    dic_item bin_tree::lookup(const K& k) const
    { return bin_tree:lookup(&k); }
  };
```

Observe that *bin_tree*::*lookup* expects a *GenPtr* and hence we pass the address of k to it.

The code for classes *bin_tree* and *dictionary*$<K, I>$ is distributed over the files *bin_tree.h*, *bin_tree.c*, and *dictionary.h* as described in the previous section: classes *bin_tree_node* and *bin_tree* are defined in *LEDA*/*impl*/*bin_tree.h*, the implementation of *bin_tree* is contained in *LEDAROOT*/*src*/*dict*/*bin_tree.c*, and *dictionary*$<K, I>$ is defined in *LEDA*/*dictionary.h*.

The above implementation of dictionaries has a weakness (which we will overcome in the next section). Consider the insert operation. According to the specification of dictionaries (see Section 5.3) a call *D.insert*(k, i) adds a new item $\langle k, i\rangle$ to D when there is no item with key k in D yet and otherwise replaces the information of the item with key k by i. However, in the implementation given above *dictionary*$<K, I>$::*insert*(k, i) makes a copy of k and then passes a pointer to this copy to *bin_tree*::*insert*. If k is already in the tree *bin_tree*::*insert* must destroy the copy again (otherwise, there would be a memory leak). It would be better to generate the copy of k only when needed.

In the next section we show how to shift the responsibility for copying and deleting data objects to the implementation class by means of virtual functions. We will also show how to implement the missing copy constructor, assignment operator, and destructor.

13.4.3 ***Copy Constructor, Assignment, and Destruction***

Copying, assignment, and destruction are fundamental operations of every data type. In C++ they are implemented by copy constructors, assignment operators, and destructors. Let us see how they are realized in LEDA. As an example, consider the assignment operation $D1 = D2$ for the data type *dictionary*$<K, I>$. A first approach would be to implement this operation on the level of abstract types, i.e., in the data type template *dictionary*$<K, I>$. We could simply first clear *D1* by a call of *D1.clear*( ) and then insert the key/information pairs for all items *it* of *D2* by calling *D1.insert*(*D2.key*(*it*), *D2.inf*(*it*)) for every one of them. This solution is inflexible and inefficient; the assignment would take time $O(n \log n)$ instead of time $O(n)$.

A second approach is to realize the operation on the level of the implementation class *bin_tree*. This requires that *bin_tree* knows how to copy a key and an information. In the destructor it also needs to know how to destroy them. There are also many other reasons why the implementation class should have these abilities, as we will see. In LEDA, we use virtual functions and dynamic binding to provide this knowledge.

In the dictionary example, we have the following virtual member functions in addition to *cmp_key*:

void copy_key(*GenPtr*& x) and *void copy_inf*(*GenPtr*& x) that make a copy of the object pointed to by x and assign a pointer to this copy to x,

void clear_key(*GenPtr* x) and *void clear_inf*(*GenPtr* x) that destroy the object pointed to by x, and finally

void assign_key(*GenPtr* x, *GenPtr* y) and *void assign_inf*(*GenPtr* x, *GenPtr* y) that assign the object pointed to by y to the object pointed to by x.

We exemplify the use of the virtual copy and clear function in two recursive member functions *copy_subtree* and *clear_subtree* of *bin_tree* that perform the actual copy and clear operations for binary trees. The copy constructor, the assignment operator, the destructor, and the *clear* function of class *bin_tree* are then realized in terms of *copy_subtree* and *clear_subtree*. The use of *assign_inf* will be demonstrated later in the realization of the operation *change_inf*.

In the header file *bin_tree.h* we extend class *bin_tree* as follows.

```
class bin_tree {
private:
  ...
  virtual void copy_key(GenPtr&) const = 0;
  virtual void clear_key(GenPtr) const = 0;
  virtual void assign_key(GenPtr x, GenPtr y) const =0;

  virtual void copy_inf(GenPtr&) const = 0;
  virtual void clear_inf(GenPtr) const = 0;
  virtual void assign_inf(GenPtr x, GenPtr y) const =0;

  void clear_subtree(bin_tree_node* p);
  // deletes subtree rooted at p

  bin_tree_node* copy_subtree(bin_tree_node* p);
  // copies subtree rooted at p, returns copy of p
protected:
  void clear();
  bin_tree(const bin_tree& T);
  bin_tree& operator=(const bin_tree& T);
 ~bin_tree() { clear(); }
};
```

In the data type template *dictionary*<K, I> we realize the virtual copy, assign, and clear functions by type casting, dereferencing, and calling the new, assignment, or delete operators of the corresponding parameter types K and I. Copy constructor, assignment operator, and destructor of type *dictionary* are implemented by calling the corresponding operations of the base class *bin_tree*.

```
template<class K, class I>
class dictionary: private bin_tree {
  ...
  void copy_key(GenPtr& x) const { x = new K(*(K*)x); }
  void copy_inf(GenPtr& x) const { x = new I(*(I*)x); }
  void clear_key(GenPtr x) const { delete (K*)x); }
  void clear_inf(GenPtr x) const { delete (I*)x); }
  void assign_key(GenPtr x, GenPtr y) const { *(K*)x = *(K*)y; }
  void assign_inf(GenPtr x, GenPtr y) const { *(I*)x = *(I*)y; }
  ...
public:
  ...
  dictionary(const dictionary<K,I>& D) : bin_tree(D) {}

  dictionary<K,I>& operator=(const dictionary<K,I>& D)
  { bin_tree::operator=(D); return *this; }

 ~dictionary() { bin_tree::clear(); }
};
```

The functions *bin_tree*::*copy_subtree*, *bin_tree*::*clear_subtree*, *bin_tree*::*clear*, the copy constructor, the destructor, and the assignment operator are implemented in bin_tree.c.

```
bin_tree_node*  bin_tree::copy_subtree(bin_tree_node* p) {
  if (p == nil) return nil;
  bin_tree_node* q = new bin_tree_node;
  q->l_child = copy_subtree(p->l_child);
  q->r_child = copy_subtree(p->r_child);
  q->key = p->key;
  q->inf = p->inf;
  copy_key(q->key);
  copy_inf(q->inf);
  return q;
}

void  bin_tree::clear_subtree(bin_tree_node* p) {
  if (p == nil) return;
  clear_subtree(p->l_child);
  clear_subtree(p->r_child);
  clear_key(p->key);
  clear_inf(p->inf);
```

```
    delete p;
  }
  void bin_tree::clear() {
    clear_subtree(root);
    root = nil;
  }
  bin_tree& bin_tree::operator=(const bin_tree& T) {
    if (this != &T)
    { clear();
      root = copy_subtree(T.root);
    }
    return *this;
  }
```

The implementation of the copy constructor is subtle. It is tempting to write (as in *operator* $=$)

```
  bin_tree::bin_tree(const bin_tree& T) { root = copy_subtree(T.root); }
```

This will not work. The correct implementation is

```
  bin_tree::bin_tree(const bin_tree& T) { root = T.copy_subtree(T.root); }
```

What is the difference? In the first case we call *copy_subtree* for the object under construction, and in the second case we call *copy_subtree* for the existing tree T. The body of *copy_subtree* seems to make no reference to either T or the object under construction. But note that all member functions of a class have an implicit argument, namely the instance to which they are applied. In particular, the functions *copy_key* and *copy_inf* are either T's versions of these functions or the new object's versions. The point is that these versions are different.

Object T belongs to class *dictionary*$<K, I>$ and hence knows the correct interpretation of *copy_key* and *copy_inf*. The object under construction does not know them yet. It knows them only when the construction is completed. As long as it is under construction the functions *copy_key* and *copy_inf* are as defined in class *bin_tree* and not as defined in the derived class *dictionary*$<K, I>$. In other words, when an object of type *dictionary*$<K, I>$ is constructed we first construct a *bin_tree* and then turn the *bin_tree* into a *dictionary*$<K, I>$. The definitions of the virtual functions are overwritten when the *bin_tree* is turned into a *dictionary*$<K, I>$.

What will happen when the wrong definition of the *copy_subtree* function is used, i.e., when the copy constructor of *bin_tree* is defined as

```
  bin_tree::bin_tree(const bin_tree& T) { root = copy_subtree(T.root); }
```

In this situation, the original definition of *copy_key* is used. According to the specification of C++ the effect of calling a virtual function directly or indirectly for the object being constructed is undefined. The compilers that we use interpret a pure virtual function as a function with an empty body and hence the program above will compile but no copies will be made. One may guard against the inadvertent call of a pure virtual function by using a virtual function whose call rises an error instead, e.g., one may define

```
virtual void copy_key(GenPtr&) { assert(false); return 0; }
```

Destructors give rise to the same problem as constructors. In a destructor of a base class virtual member functions also have the meaning defined in the base class and not the meaning given in a derived class. What does this mean for the destructor of class *dictionary*<K, I>? It first calls *bin_tree*::*clear* and then the destructor of the base class *bin_tree* (the latter call is generated by the compiler). The destructor of *bin_tree* again calls *bin_tree*::*clear*. So why do we need the first call at all? We need it because the second call uses the "wrong" definitions of the virtual functions *clear_key* and *clear_inf*. When *bin_tree*::*clear* is called for the second time the object to be destroyed does not know anymore that it was a *dictionary*<K, I>. The second call of the *clear* is actually unnecessary. We put it for reasons of uniformity; it incurs only very small additional cost.

Since *bin_tree* now knows how to copy and destroy the objects of type K and I, respectively, we can write correct implementations of the operations *del_item* and *insert* on the level of the implementation class, i.e., use precompiled versions of these functions, too.

```
void bin_tree::del_item(bin_tree_node* p) {
  // remove p from the tree
    ...
  clear_key(p->key);
  clear_inf(p->inf);
  delete p;
}

bin_tree_node* bin_tree::insert(GenPtr k, GenPtr i) {
  bin_tree_node* p = lookup(k);

  if (p != nil) { // k already present
    change_inf(p,i);
    return p;
  }

  copy_key(k);
  copy_inf(i);

  p = new bin_tree_node();
  p->key = k;
  p->inf = i;

  // insert p into tree
  ...
  return p;
}
```

By using the virtual *assign_inf* function we can realize the *change_inf* operation on the level of the implementation class, too.

```
void bin_tree::change_inf(bin_tree_node* p, GenPtr i) {
  assign_inf(p->inf,i);
}
```

With this modification the corresponding operations in the *dictionary*<K, I> template do not need to copy or destroy a key or an information anymore. They just pass the addresses of their arguments of type K and I to the member functions of class *bin_tree*.

```
template<class K, class I>
class dictionary: private bin_tree {
  ...
public:
  ...
  void del_item(dic_item it) { bin_tree::del_item(it); }

  void change_inf(dic_item it, const I& i)
  { bin_tree::change_inf(it,&i); }

  dic_item insert(const K& k, const I& i) { bin_tree::insert(&k,&i); }
```

13.4.4 *Arrays and Default Construction*

Some parameterized data types require that the actual element type has a default constructor, i.e., a constructor taking no arguments, that initializes the object under construction to some default value of the data type. The LEDA data types *array* and *map* are examples for such types.

The declaration

```
array<string> A(1,100);
```

creates an array of 100 variables of type *string* and initializes every variable with the empty string (using the default constructor of type *string*).

The declaration

```
map<int,vector>  M;
```

creates a map with index type *int* and element type *vector*, i.e., a mapping from the set of all integers of type *int* to the set of variables of type *vector*. All variables are initialized with the vector of dimension zero (the default value of type *vector*).

Note that a default constructor does not necessarily need to initialize the object under construction to a unique default value. There are data types that have no natural default value (for example, a line segment) and there are others where initialization to a default value is not done for efficiency reasons. In these cases, the default constructor simply constructs some arbitrary object of the data type. Examples for such types are the built-in types of C++. The declaration

```
int x;
```

declares x as a variable of type *int* initialized to some unspecified integer, and the declaration

```
array<int> A(1,100);
```

creates an array of 100 variables of type *int* each holding some arbitrary integer.

As for copying, assignment, and destruction, LEDA implements default initialization of parameterized data types in the corresponding implementation class by virtual functions and dynamic binding. We use the array data type as an example.

The parameterized data type *array*<T> is derived from the implementation class *gen_array* of arrays for generic pointers. The class *gen_array* provides two operations *init_all_entries* and *clear_all_entries* which can be called to initialize or to destroy all entries of the array, respectively. They use the virtual member functions *void init_entry*(*GenPtr*&) and *void clear_entry*(*GenPtr*) to do the actual work, i.e., they use the first function to initialize an array entry and the second function to destroy one.

```
class gen_array {
  GenPtr* first;
  GenPtr* last;
  ...
  virtual void init_entry(GenPtr& x) = 0;
  virtual void clear_entry(GenPtr x) = 0;
  ...
protected:
  ...
  void init_all_entries()
  { for(GenPtr* p = first; p <= last; p++) init_entry(*p); }
  void clear_all_entries()
  { for(GenPtr* p = first; p <= last; p++) clear_entry(*p); }
};
```

In the data type class *array*<T> we define *init_entry* and *clear_entry* by calling the new and delete operator of type T, respectively. The constructor of *array*<T> uses *init_all_entries* to initialize all elements of the array and the destructor uses *clear_all_entries* to destroy all objects stored in the array.

```
template <class T>
class array : private gen_array {
  void init_entry(GenPtr& x) { x = new T; }
  void clear_entry(GenPtr x) { delete (T*)x; }
public:
  ...
  array(int l, int h) : gen_array(l,h) { init_all_entries(); }
 ~array() { clear_all_entries(); }
};
```

We give one more example of default construction, the *new_node* and *new_edge* operations of parameterized graphs *GRAPH*<*vtype*, *etype*>. There are two variants of these operations: the first one takes an argument that is used to initialize the information associated with the new object (node or edge).

```
node G.new_node(const vtype&)
edge G.new_edge(node, node, const etype&)
```

The second one does not take such an argument. Here the information associated with the object is initialized by the default constructor of the corresponding type (*vtype* or *etype*).

```
node G.new_node()
edge G.new_edge(node v, node w)
```

The following piece of code constructs a graph with two nodes v and w connected by an edge $e = (v, w)$. The nodes are labeled with the default value of type *string*, i.e., the empty string, and edge e is labeled with a vector of dimension zero, the default value of type *vector*.

```
GRAPH<string,vector> G;
node v = G.new_node();
node w = G.new_node();
edge e = G.new_edge(v,w);
```

Default initialization for nodes and edges is also used by LEDA's various graph generators. If G is a parameterized graph of type *GRAPH<vtype, etype>*, a call *random_graph*(G, n, m) constructs a random graph with n nodes and m edges where each node information is initialized by the default constructor of type *vtype* and each edge information is initialized by the default constructor of type *etype*.

13.4.5 *Some Useful Function Templates*

In *<LEDA/param_types.h>* we define five function templates that are useful to define the virtual functions required in the LEDA approach.

```
template <class T>
inline T& leda_access(const T*, const GenPtr& p) { return *(T*)p; }
```

returns a reference to the object of type T pointed to by p. The first argument of this function template is a dummy pointer argument of type $T*$ that is used for selecting the correct instantiation. For instance, to access an object of type T through a generic pointer p we write *leda_access*$((T*)0, p)$. As an abbreviation LEDA provides the macro.

```
#define LEDA_ACCESS(T,p)       leda_access((T*)0,p)
```

The function template

```
template <class T>
inline GenPtr leda_create(const T*) { return new T; }
```

returns a generic pointer to an object of type T initialized with the default value of type T. Again, there is a dummy pointer argument of type $T*$.

The function template

```
template<class T>
inline GenPtr leda_copy(const T& x) { return new T(x); }
```

returns a generic pointer to an object of type T initialized with a copy of x.

The function template

```
template <class T>
inline void leda_clear(T& x) { T* p = &x; delete p; }
```

destroys the object stored at x and the function template

```
template <class T>
inline GenPtr leda_cast(const T& x) { return (GenPtr)&x; }
```

returns the address of *x* casted to a generic pointer.

Given these function pointers it is easy to define the virtual function required in the LEDA approach in a generic way for every type parameter *T*.

```
void create_T(GenPtr& p)  { p = leda_create((T*)0); }
void copy_T  (GenPtr& p)  { p = leda_copy(LEDA_ACCESS(T,p)); }
void clear_T (GenPtr  p)  { leda_clear(LEDA_ACCESS(T,p)); }
void assign_T(GenPtr& p, GenPtr q)
                          { LEDA_ACCESS(T,p) = LEDA_ACCESS(T,q); }
```

We return to the dictionary and array data type templates to demonstrate the use of the above defined function templates and macros. We have

```
class dictionary : public bin_tree {
  int  cmp(GenPtr x, GenPtr y) const
             { return compare(LEDA_ACCESS(K,x), LEDA_ACCESS(K,x,y); }
  void clear_key(GenPtr& x) const { leda_clear(LEDA_ACCESS(K,x)); }
  void clear_inf(GenPtr& x) const { leda_clear(LEDA_ACCESS(I,x)); }
  void copy_key(GenPtr& x)  const { x = leda_copy(LEDA_ACCESS(K,x)); }
  void copy_inf(GenPtr& x)  const { x = leda_copy(LEDA_ACCESS(I,x)); }
  void assign_inf(GenPtr& x, GenPtr y) const
                              { LEDA_ACCESS(I,x) = LEDA_ACCESS(I,y); }
public:
  ...
  K key(dic_item it) const
  { return LEDA_ACCESS(K,bin_tree::key(it)); }
  I inf(dic_item it) const
  { return LEDA_ACCESS(I,bin_tree::inf(it)); }
  dic_item insert(const K& k, const I& i)
  { return bin_tree::insert(leda_cast(k),leda_cast(i)); }
  dic_item lookup(const K& k) const
  { return bin_tree::lookup(leda_cast(k)); }
  void change_inf(dic_item it, const I& i)
  { bin_tree::change_inf(it,leda_cast(i)); }
  ...
};
```

and

```
template <class T>
class array : private gen_array {
  void init_entry(GenPtr& x) { x = leda_create((T*)0); }
  void clear_entry(GenPtr x) { leda_clear(LEDA_ACCESS(T,x)); }
  ...
};
```

13.4.6 *Further Uses of Virtual Functions*

There are many other situations where LEDA uses virtual functions for transferring functionality of actual type arguments from the data type class to the implementation class. Examples are:

- Printing and Reading
- Hashing
- Id-Numbers
- Type Information (see the next section)
- Rebalancing of binary trees

We touched upon printing and reading in Section 5.7.3, an example of the use of id-numbers can be found in Section 5.1.2, and we will see type information in Section 13.5.3.

Exercises for 13.4

1 Write a template implementation of the LEDA data type *queue*.

2 Is it correct to change the interface of *pop* to `const T& pop()`?

3 The implementation of *list*<T>::*clear* which simply calls *list_impl*::*clear* has a memory leak, as it leaves the entries contained in the elements of the list as orphans on the heap. Why does *t_list*<T>::*clear* not have a memory leak?

4 Define a class *dlist*<T> that implements doubly linked lists for elements of type T. Use the template approach and convert the solution to the LEDA approach.

5 Add an operation *pop*(T& x) to the list data type that returns the result of the pop operation in the reference parameter x.

6 In the text we established a relationship between corresponding states of *t_list*<T> and *list*<T>. Argue that the implementations of the various functions of the list data type leave this correspondence invariant.

7 Consider the following skeleton for the function *bin_tree*::*insert*.

```
bin_tree_node* insert(void* k, void* i)
{ bin_tree_node *p = root, *q = nil;  // q is always the parent of p
  int c;
  while (p)
  { c = cmp_key(k,p->key);
    if (c == 0)
    { // something is missing here
      return p;
    }
    q = p;
    p = (c > 0) ? p->right_child : p->left_child;
  }
  if ( c > 0 ) return q->right_child = new bin_tree_node(k,i);
  else return q->left_child) = new bin_tree_node(k,i);
}
```

Complete the code. Make sure that your implementation has no memory leak.

13.5 Optimizations

In this section we describe some optimizations that can be applied to special type arguments of parameterized data types.

13.5.1 *Small Types*

The LEDA solution for parameterized data types presented in the preceding sections uses one additional (generic) pointer field for every value or object that is stored in the data type. The method incurs overhead in space and time, in space for the additional pointer and in time for the additional indirection. We show how to avoid the overhead for types whose values are no larger than a pointer. In C++ the space requirement of a type is easily detemined: $\mathit{sizeof}(T)$ returns the size of the objects of type T in bytes. We call a type T *small* if $\mathit{sizeof}(T) \le \mathit{sizeof}(\mathit{GenPtr})$ and *large* otherwise. By definition, all pointer types are small. On 32 bit systems the built-in types *char*, *short*, *int*, *long*, *float* are small as well, and type *double* is big. On 64 bit systems even the type *double* is small. Note that class types can be small too, e.g., a class containing a single pointer data member. An example for small class types are the LEDA *handle types* that will be discussed in Section 13.7.

Values of any small type T can be stored directly in a data field of type *void*∗ or *GenPtr* by using the *in-place new operator* of C++. If p is a pointer of type *void*∗

```
new(p) T(x);
```

calls the copy constructor of type T to construct a copy of x at the address in memory that p points to, in other words with $\mathit{this} = p$. Similarly,

```
new(p) T;
```

calls the default constructor of type T (if defined) to construct the default value of type T at the location that p points to.

We use the in-place new operator as follows. If y is a variable corresponding to a data field of some container and T is a small type then

```
new(&y) T(x);
new(&y) T;
```

constuct a copy of x and the default value of T directly in y.

Of course, small objects have to be destroyed too. For this purpose we will use the *explicit destructor call* of C++. If z is a variable of some type T,

```
z.~T()
```

calls the destructor of T for the object stored in z. Destructor calls for named objects are constructed automatically in C++ when the scope of the object ends, and therefore few C++ programmers ever need to make an explicit destructor call.

We have to. Observe that we construct objects of type T in variables of type *void*∗ and therefore cannot rely on the compiler to generate the destructor call. We destroy an object of type T stored in a variable y of type *void*∗ by casting the address of y to a pointer of type $T*$ and calling the destructor explicitly as in

```
((T*)&y)->~T();
```

To access the value of a small type T stored in a *void*∗ data field y we take the address of y, cast it into a $T*$ pointer, and dereference this pointer.

```
*((T*)&y)
```

13.5.2 *Summary of LEDA Approach to Parameterized Data Types*

We summarize the LEDA approach to parameterized data types. We store values of arbitrary types T in data fields of type *void*∗ (also called *GenPtr*). We distinguish between small and large types.

For objects of a large type T ($\mathit{sizeof}(T) > \mathit{sizeof}(\mathit{GenPtr})$) we make copies in the dynamic memory using the *new* operator and store pointers to the copies.

For objects of a small type T ($\mathit{sizeof}(T) \leq \mathit{sizeof}(\mathit{GenPtr})$) we avoid the overhead of an extra level of indirection by copying the value directly into the *void*∗ data field using the "in-place" variant of the *new* operator.

We next give versions of *leda_copy*, *leda_create*, *leda_clear*, *leda_access*, and *leda_cast* that can handle small and large types. The functions are defined in LEDA/param_types.h.

GenPtr leda_copy(*const T*& x) makes a copy of x and returns it as a generic pointer of type *GenPtr*. If T is a small type, the copy of x is constructed directly in a *GenPtr* variable using the in-place new operator of T, and if T is a big type, the copy of x is constructed in the dynamic memory (using the default new operator) and a pointer to this copy is returned.

```
template<class T>
inline GenPtr leda_copy(const T& x)
{ GenPtr p;
  if (sizeof(T) <= sizeof(GenPtr)) new(&p) T(x);
  if (sizeof(T) >  sizeof(GenPtr)) p = new T(x);
  return p;
}
```

GenPtr leda_create(*const T*∗) constructs the default value of type T by a call of either the in-place new or the normal new operator of T.

```
template <class T>
inline GenPtr leda_create(const T*)
{ GenPtr p;
  if (sizeof(T) <= sizeof(GenPtr)) new(&p) T;
  if (sizeof(T) >  sizeof(GenPtr)) p = new T;
  return p;
}
```

void leda_clear(*T*& x) destroys the object stored in x either by calling the destructor of T explicitly or by calling the *delete* operator on the address of x.

```
template <class T>
inline void leda_clear(T& x)
{ T* p = &x;
```

```
    if (sizeof(T) <= sizeof(GenPtr)) p->~T();
    if (sizeof(T) >  sizeof(GenPtr)) delete p;
  }
```

T& leda_access(const T∗, const GenPtr& p) returns a reference to the object of type *T* stored in *p* or pointed to by *p* respectively.

```
  template <class T>
  inline T& leda_access(const T*, const GenPtr& p)
  { if (sizeof(T) <= sizeof(GenPtr)) return *(T*)&p;
    if (sizeof(T) >  sizeof(GenPtr)) return *(T*)p;
  }
```

GenPtr leda_cast(const T& x) either returns the value of *x* or the address of *x* casted to a generic pointer.

```
  template <class T>
  inline GenPtr leda_cast(const T& x)
  { GenPtr p;
    if (sizeof(T) <= sizeof(GenPtr)) *(T*)&p = x;
    if (sizeof(T) >  sizeof(GenPtr)) p = (GenPtr)&x;
    return p;
  }
```

The functions above incur no overhead at run time. Note that all comparisons between the size of *T* and the size of a pointer can be evaluated at compile-time when instantiating the corresponding function template and therefore do not cause any overhead at run time.

13.5.3 *Optimizations for Built-in Types*

Our method of implementing parameterized data types stores the objects of the data type in *void∗* data fields and uses virtual member functions for passing type-specific functionality from the data type template to the implementation class.

In a previous section we already showed how to avoid the space overhead of an additional pointer for small types. However, there is also an overhead in time. Every type-dependent operation, such as comparing two keys in a dictionary, is realized by a virtual member function. Calling such a function, e.g., in the inner loop when searching down a tree, can be very expensive compared to the cost of applying a built-in comparison operator.

LEDA has a mechanism for telling the implementation class that an actual type parameter is one of the built-in types in order to avoid this overhead. For the identification of these types we use an enumeration. For every built-in type *xyz* this enumeration contains an element *XYZ_TYPE_ID*. There is also an *UNKNOWN_TYPE_ID* member used for indicating that the corresponding type is unknown, i.e., is not one of the built-in types.

```
  enum { UNKNOWN_TYPE_ID, CHAR_TYPE_ID, SHORT_TYPE_ID, INT_TYPE_ID,
         LONG_TYPE_ID, FLOAT_TYPE_ID, DOUBLE_TYPE_ID };
```

To compute the type identification for a given type we use a global function *leda_type_id*. Given a pointer to some type *T* this function returns the corresponding type identification, e.g., if *T* = *int*, it will return *INT_TYPE_ID*, if *T* is not one of the recognized types, the result is *UNKNOWN_TYPE_ID*. We first define a default function template returning the special value *UNKNOWN_TYPE_ID* and then define specializations for all built-in types.

```
template <class T>
inline int leda_type_id(const T*)     { return UNKNOWN_TYPE_ID; }

inline int leda_type_id(const char*)  { return CHAR_TYPE_ID; }
inline int leda_type_id(const int*)   { return INT_TYPE_ID; }
inline int leda_type_id(const long*)  { return LONG_TYPE_ID; }
inline int leda_type_id(const double*){ return DOUBLE_TYPE_ID; }
...
```

Now we can add a virtual function *key_type_id* to the dictionary implementation and define it in the corresponding data type template by calling the *leda_type_id* function with an appropriate pointer value.

```
class bin_tree {
  ...
  virtual int key_type_id() = 0;
  ...
};

template <class K, class I>
class dictionary {
  ...
  int key_type_id() { return leda_type_id((K*)0); }
  ...
};
```

In the implementation of the various dictionary operations (in *bin_tree.c*) we can now determine whether the actual key type is one of the basic types and choose between different optimizations. We use the *bin_tree*::*search* member function as an example. Let us assume we want to write a special version of this function for the built-in type *int* that does not call the expensive *cmp_key* function but compares keys directly. First we call *type_id*( ) to get the actual key type id and in the case of *INT_TYPE_ID* we use a special searching loop that compares keys using the *LEDA_ACCESS* macro and the built-in comparison operators for type *int*.

```
bin_tree_node* bin_tree::search(GenPtr x) const
{
  bin_tree_node* p = root;

  switch ( type_id() ) {

   case INT_TYPE_ID: {
         int x_int =  LEDA_ACCESS(int,x);
         while (p)
         { int p_int =  LEDA_ACCESS(int,p->k);
           if (x_int == p_int) break;
           p = (x_int < p_int) ? p->left_child : p->right_child;
         }
         break;
       }

   default: {
```

n	*myint*	*int*
1000000	6.74	0.68

Table 13.1 The effect of the optimization for built-in types. The time to sort an array of *n* random elements is shown. The table was generated with the program built_in_types_optimization in directory LEDAROOT/demo/book/Impl.

```
        while (p)
        { int c = cmp(x,p->k);
          if (c == 0) break;
          p = (c < 0) ? p->left_child : p->right_child;
        }
        break;
      }
  }
  return p;
}
```

The above piece of code is easily extended to other built-in types.

Table 13.1 shows the effect of the optimization. We defined a class *myint* that encapsulates an *int*

⟨*class myint*⟩≡

```
class myint {
  int x;
public:
  myint() {}
  myint(const int _x): x(_x) {}
  myint(const myint& p)  {  x = p.x; }
  friend void operator>>(istream& is, myint& p) { is >> p.x; };
  friend ostream& operator<<(ostream& os, myint& p)
         { os << p.x; return os; };
  friend int compare(const myint&,const myint&);
};
int compare(const myint& p,const myint& q)
{
  if (p.x == q.x) return 0;
  if (p.x < q.x) return -1; else return +1;
}
```

and then built two arrays of size *n*, one filled with random *int*s and the other one filled with the same *myint*s. We then sorted both arrays. Table 13.1 shows that the optimization leads to a considerable reduction in running time.

Exercise for 13.5

1 Extend the search procedure for binary trees such that it uses the optimization also for *double*s.

13.6 **Implementation Parameters**

There are many implementations of dictionaries: binary trees, skiplists, hashing, sorted arrays, self-adjusting lists, ... Which implementation should be included in a library?

If one provides only one implementation, then this implementation should clearly be the "best" possible. This was the direction taken in the first versions of LEDA. In the case of the dictionary data type, we included red-black trees because they are asymptotically as efficient as any other implementation. But, of course, only asymptotically. Also, there are better implementations for special cases, e.g., for integer keys from a bounded universe. For other data types, e.g., range trees, there are implementations with vastly differing performance parameters (time-space tradeoff) and so there is not even an asymptotically best implementation. All of this implies that providing only one implementation for each data type is not satisfactory.

So, one has to provide many and allow for the possibility of adding more. What properties should a mechanism for choosing between different implementations have?

(1) There should be a simple syntax for choosing between different implementations. In LEDA, the declaration

```
_dictionary<K,I,rb_tree> D;
```

creates an empty dictionary with key type K and information type I and selects red-black trees as the implementation variant, *_dictionary*<K, I, *impl*> selects the implementation *impl*. The actual type parameter for *impl* has to be a dictionary implementation, i.e., must be a class that provides a certain set of operations and uses virtual functions for type dependent operations. This will be discussed below. The declaration

```
dictionary<K,I> D;
```

selects the default implementation (skiplists in the current version).

Remark: Because templates cannot be overloaded in C++ we have to use different names *dictionary* and *_dictionary*. The general rule is that the data type variant with implementation parameter starts with an underscore.

(2) Applications can be written that work with any implementation. For example, applications that use a dictionary are written as functions with an additional parameter of the abstract dictionary type. Then the function can be called with any implementation of the dictionary type. We illustrate this feature with the word-count example.

```
void WORD_COUNT(const list<string>& L, dictionary<string,int>& D)
{ string s;
  forall(s,L)
```

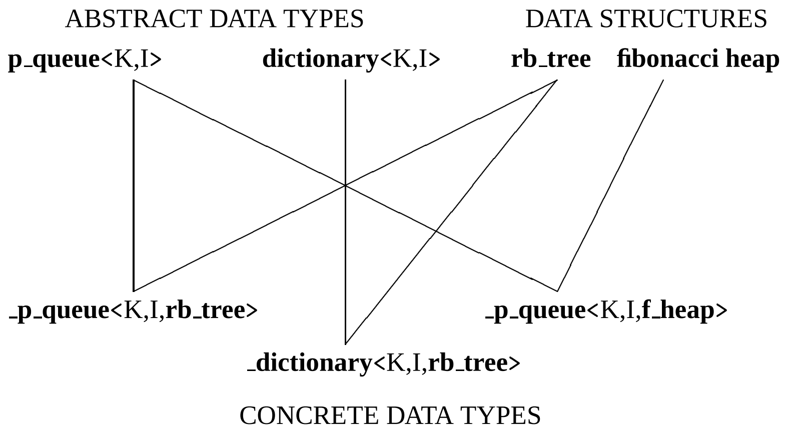

Figure 13.2 Multiple inheritance combines abstract data types and data structures to concrete data types.

```
    { dic_item it = D.lookup(s);
      if (it == nil)
         D.insert(s,1);
      else
         D.change_inf(it,D.inf(it)+1);
    }
    dic_item it;
    forall_items(it,D)
      cout << D.key(it) << " appeared " << D.inf(it) << " times.";
}
```

In the context of the declarations

```
dictionary<string, int>            SL_D;  // skiplists
_dictionary<string, int, rb_tree> RB_D;  // red-black trees
_dictionary<string, int, my_impl> MY_D;  // user implementation
```

the calls

```
WORD_COUNT(L,SL_D);
WORD_COUNT(L,RB_D);
WORD_COUNT(L,MY_D);
```

are now possible.

The realization of the implementation parameter mechanism makes use of multiple inheritance, cf. Figure 13.2. Every concrete data type, say dictionary with the rb_tree implementation, is derived from the abstract data type and the data structure used to implement it. In the abstract data type class, all functions are virtual, i.e., have unspecified implementations. In the data structure class the details of the implementation are given and the classes in the bottom line of Figure 13.2 are used to match the abstract functions with the concrete implementations.

```
template<class K,class I> class dictionary : private default_impl
{
  int cmp_key(GenPtr x, GenPtr y)
  { return compare(LEDA_ACCESS(K,x), LEDA_ACCESS(K,y)); }
  void clear_key(GenPtr x) { leda_clear(LEDA_ACCESS(K,x)); }
public:
  virtual K        key(dic_item it) = 0;
  virtual dic_item lookup(K y)      = 0;
  virtual dic_item insert(K x, I y) = 0;
  virtual void     del(K y)         = 0;
  ...
};
```

Dictionaries with implementation parameter can now be derived from the abstract dictionary class.

```
template<class K, class I,class IMPL>
class _dictionary : private IMPL, public dictionary<K,I>
{
  public:
  K        key(dic_item it) { return LEAD_ACCESS(K,IMPL::key(it)); }
  dic_item lookup(K y)      { return IMPL::lookup(leda_cast(y)); }
  dic_item insert(K x, I y)
           { return IMPL::insert(leda_cast(x),leda_cast(y)); }
  void     del(K y)         { IMPL::del(leda_cast(y)); }
  ...
};
```

Of course, an implementation class *IMPL* can be used as actual implementation parameter of a parameterized data type only if it provides all necessary operations and definitions and calls type-dependent functions through the appropriate virtual member functions. For item-based types, it must in addition define a local type *item* representing the items of the data type. In the case of dictionaries, any class *dic_impl* with the following definitions and declarations can be used as implementation class.

```
class dic_impl {
  // type dependent functions
  virtual int  cmp(GenPtr, GenPtr) const = 0;
  virtual int  type_id()           const = 0;
  virtual void clear_key(GenPtr&)  const = 0;
  virtual void clear_inf(GenPtr&)  const = 0;
  virtual void copy_key(GenPtr&)   const = 0;
  virtual void copy_inf(GenPtr&)   const = 0;
  virtual void assign_inf(GenPtr&, GenPtr) const = 0;
public:
  // definition of the item type
  typedef ... item;
  // construction, destruction, copying
  dic_impl();
```

```
    dic_impl(const dic_impl&);
   ~dic_impl();
    dic_impl& operator=(const dic_impl&);

    // dictionary operations
    GenPtr key(item)  const;
    GenPtr inf(item)   const;

    item insert(GenPtr,GenPtr);
    item lookup(GenPtr)  const;

    void change_inf(item,GenPtr);
    void del_item(item);
    void del(GenPtr);
    void clear();

    int size() const;

    // iteration
    item first_item() const;
    item next_item(item) const;
  };
```

For most of its parameterized data types LEDA provides several implementation classes. Before using an implementation class *xyz* the corresponding header file *<LEDA/impl/xyz.h>* has to be included. The following dictionary implementations are currently available: AVL-Trees (*avl_tree*), (a,b)-Trees (*ab_tree*), BB[α]-Trees (*bb_tree*), Skiplists (*skiplist*), Red-Black-Trees (*rb_tree*), Randomized Search Trees (*rs_tree*), Dynamic Perfect Hashing (*dp_hashing*), and Hashing with Chaining (*ch_hashing*).

Section "Available Implementations" of the LEDA user manual gives the complete list of all available implementations.

Exercises for 13.6

1 Write an implementation class for dictionaries based on the *so_set* class of Section 3.2.

2 Write an implementation class for priority queues.

13.7 Independent Item Types (Handle Types)

All independent item types of LEDA (cf. Section 2.2.2) are implemented by so-called *handle types*. Basically, a handle type H is a pointer (or handle) to some representation class *H_rep* that contains all data members used for the representation of objects of type H. Assignment and copy operations translate to simple pointer assignments and the test for identity translates to the equality test for pointers. Thus assignment, copy operations, and identity functions are easily handled, but destruction of representation objects causes a problem.

A representation object has to be destroyed as soon as no handle is pointing to it anymore. To detect this situation we use a technique called *reference counting*. Every representation object has a reference counter *ref_count* that contains the number of handles which are still in scope and point to the object. The counters are updated in the copy constructor, assignment operator, and destructor of the corresponding handle class.

We use a two-dimensional point class *point* as an example. The representation class *point_rep* has three data members, a pair of floating-point coordinates (x, y) and a reference counter *ref_count*. A constructor initializing the coordinates to two given values and setting the reference counter to one is the only member function.

```
class point_rep {
  double x, y;
  int    ref_count;
  point_rep(double a, double b) :x(a),y(b),ref_count(1) {}
};
```

Now we could implement points by pointers to the representation class *point_rep*. However, just using the type *point_rep*$*$ for representing points, as in

```
typedef point_rep* point;
```

would not make reference counting work automatically when variables of type *point* are created, assigned to each other, or destroyed. Therefore *point* has to be implemented by a real C++ class with constructors, destructor, and assignment operator.

The only data member of class *point* is a pointer to the corresponding representation class *point_rep*.

```
class point {
  point_rep* ptr;
public:
  point(double,double);
  point(const point&);
  point& operator=(const point&);
 ~point();
  double xcoord()    const;
  double ycoord()    const;
  point  translate() const;
  friend bool identical(const point& x, const point& y);
};
```

The constructor of class *point* creates a new representation object (with *ref_count* equal to one) in the dynamic memory and assigns the pointer to *ptr*. The copy constructor copies the corresponding pointer and increases the reference counter of the representation object by one. The destructor decreases the corresponding reference counter by one and deletes the representation object if the new value of the counter is zero.

```
point::point(double x, double y) { ptr = new point_rep(x,y); }
point::point(const point& p)
{ ptr = p.ptr;
  ptr->count++;
}
point::~point() { if (--ptr->ref_count == 0) delete ptr; }
```

In an assignment operation $q = p$ we first increase the reference counter of the representation object pointed to by p and then decrease the counter of the representation object pointed to by q. If the counter of the representation object pointed to by q is zero afterwards then q was the only handle pointing to the representation object and we have to delete it. Note that in the case that p and q are identical the same reference counter is first increased and then decreased and hence is unchanged in the end.

```
point& point::operator=(const point& p)
{ p.ptr->count++;
  if (--ptr->count == 0) delete ptr;
  ptr = x.ptr;
  return *this;
}
```

Two handles are identical if they share a common representation object, i.e., the *identical* function reduces to pointer equality.

```
bool identical(const point& x, const point& y)
{ return x.ptr == y.ptr; }
```

The above defined member functions and operators are common to all handle types. We will show how to put them in a common base class for all handle types below.

In order to complete the definition of *points*, we still have to implement the individual operations specific to them. For example,

```
double point::xcoord() const { return ptr->x; }
double point::ycoord() const { return ptr->y; }

point  point::translate(double dx, double dy) const
{ return point(ptr->x+dx, ptr->y+dy); }
```

Classes handle_rep and handle_base: As mentioned above, there is a group of operations that is the same for all handle types (copy constructor, assignment, destructor, identity). LEDA encapsulates these operations in two classes *handle_rep* and *handle_base* (see <*LEDA*/*handle_types.h*>). Concrete handle types and their representation classes are derived from them. This will be demonstrated for the *point* type at the end of this section.

The *handle_rep* base class contains a reference counter of type *int* as its only data member, a constructor initializing the counter to 1, and a trivial destructor. Later we will derive representation classes of particular handle types (e.g., *point_rep*) from this base class adding type specific individual data members (e.g., x- and y-coordinates of type *double*).

```
class handle_rep  {
  int ref_count;

  handle_rep() : ref_count(1) {}
  virtual ~handle_rep() {}

  friend class handle_base;
};
```

The *handle_base* class has a data member *PTR* of type *handle_rep*$*$, a copy constructor, an assignment operator, and a destructor. Furthermore, it defines a friend function *identical* that declares two *handle_base* objects identical if and only if their *PTR* fields point to the same representation object. Specific handle types (e.g., *point*) derived from *handle_base* use the *PTR* field for storing pointers to the corresponding representation objects (e.g., *point_rep*) derived from *handle_rep*.

```
class handle_base {
  handle_rep* PTR;
  handle_base(const handle_base& x)
  { PTR = x.PTR;
    PTR->ref_count++;
  }
  handle_base& operator=(const handle_base& x)
  { x.PTR->ref_count++;
    if (--PTR->ref_count == 0)  delete PTR;
    PTR = x.PTR;
    return *this;
  }
 ~handle_base() { if (--PTR->ref_count == 0)  delete PTR; }
  friend bool identical(const handle_base& x, const handle_base& y)
  { return x.PTR == y.PTR; }
};
```

This completes the definition of classes *handle_base* and *handle_rep*. We can now derive an independent item type T from *handle_base* and the corresponding representation class *T_rep* from *handle_rep*. We demonstrate the technique using the point example.

point_rep is derived from *handle_rep* adding two data members for the x- and y-coordinates and a constructor initializing these members.

```
class point_rep  : public handle_rep {
  double x, y;
  point_rep(double a, double b) x(a), y(b) { }
 ~point_rep() {}
};
```

We will next derive class *point* from *handle_base*. The class *point* uses the inherited *PTR* field for storing *pointer_rep*$*$ pointers. The constructor constructs a new object of type *point_rep* in the dynamic memory and stores a pointer to it in the *PTR* field, and copy constructor and assignment reduce to the corresponding function of the base class. In order to access the representation object we cast *PTR* to *point_rep*$*$. This is safe since *PTR* always points to a *point_rep*. For convenience, we add an inline member function *ptr*( ) that performs this casting. Now we can write *ptr*( ) wherever we used *ptr* in the original *point* class at the beginning of this section. The full class definition is as follows:

```
class point  : public handle_base
{
  point_rep* ptr() const { return (point_rep*)PTR; }
public:
  point(double x=0, double y=0) { PTR = new point_rep(x,y); }
  point(const point& p) : handle_base(p) {}
 ~point() {}
  point& operator=(const point& p)
  { handle_base::operator=(p); return *this; }
  double xcoord()  const   { return ptr()->x; }
  double ycoord()  const   { return ptr()->y; }
  point  translate(double dx, double dy) const
  { return point(xcoord() + dx, ycoord() + dy); }
};
```

Note that all the “routine work” (copy construction, assignment, destruction) is done by the corresponding functions of the base class *handle_base*.

Exercises for 13.7

1 Explain why the destructor *handle_rep*::∼*handle_rep*( ) is declared *virtual*.
2 How would the above code have to be changed if it were not *virtual*?
3 Implement a *string* handle type using the mechanism described above.
4 Add an array subscript operator *char*& *string*::*operator*[](*int i*) to your string class. What kind of problem is caused by this operator and how can you solve it?

13.8 **Memory Management**

Many LEDA data types are implemented by collections of small objects or nodes in the dynamic memory, e.g., lists consist of list elements, graphs consist of nodes and edges, and handle types are realized by pointers to small representation objects.

Most of these data types are dynamic and thus spend considerable time for the creation and destruction of these small objects by calling the *new* and *delete* operators.

Typically, the C++ default *new* operator is implemented by calling the *malloc* function of the *C* standard library

```
void* operator new(size_t bytes) { return malloc(bytes) }
```

and the default *delete* operator by calling the *free* library function

```
void operator delete(void* p) { free(p); }
```

Unfortunately, *malloc* and *free* are rather expensive system calls on most systems.

LEDA offers an efficient memory manager that is used for all node, edge and item types. The manager can easily be applied to a user defined class *T* by adding the macro call

"LEDA_MEMORY(T)" to the declaration of the class T. This redefines the new and delete operators for type T, such that they allocate and deallocate memory using LEDA's internal memory manager.

The basic idea in the implementation of the memory manager is to amortize the expensive system calls to *malloc* and *free* over a large sequence of requests (calls of *new* and *delete*) for small pieces of memory. For this purpose, LEDA uses *malloc* only for the allocation of large memory blocks of a fixed size (e.g., 4 kbytes). These blocks are sliced into chunks of the requested size and the chunks are maintained in a singly linked list. The strategy just outlined is efficient if the size of the chunks is small compared to the size of a block. Therefore the memory manager applies this strategy only to requests for memory pieces up to a certain size. Requests for larger pieces of memory (often called vectors) are directly mapped to *malloc* calls. The maximal size of memory chunks handled by the manager can be specified in the constructor. For the standard memory manager used in the *LEDA_MEMORY* macros this upper bound is set to 255 bytes.

The heads of all lists of free memory chunks are stored in a table *free_list*[256]. Whenever an application asks for a piece of memory of size $sz < 256$ the manager first checks whether the corresponding list *free_list*[*sz*] is empty. If the list is non-empty, the first element of the list is returned, and if the list is empty, it is filled by allocating a new block and slicing it as described above. Freeing a piece of memory of size $sz < 256$ in a call of the *delete* operator is realized by inserting it at the front of list *free_list*[*sz*].

Applications can call the global function *print_statistics* to get a summary of the current state of the standard memory manager. It prints for every chunk size that has been used in the program the number of free and still used memory chunks.

The following example illustrates the effect of the memory manager. We defined a class *pair* and a class *dumb_pair*. The definitions of the two classes are identical except that *dumb_pair* does not use the LEDA memory manager.

⟨*class pair*⟩≡
```
class pair {
  double x, y;
public:
  pair(double a=0, double b=0) : x(a), y(b) { }
  pair(const pair& p) : x(p.x), y(p.y) { }
  friend ostream&  operator<<(ostream& ostr, const pair&) {return  ostr;}
  friend istream&  operator>>(istream& istr, pair&) { return istr; }
  LEDA_MEMORY(pair)  // not present in dumb_pair
};
```

We then built a list of n pairs or dumb pairs, respectively, and cleared them again. Table 13.2 shows the difference in running time. We also printed the memory statistics before and after the *clear* operation.

n	LEDA memory	C++ memory
1000000	0.94	2.77

Table 13.2 The effect of the memory manager. We built and destroyed a list of n pairs or dumb pairs, respectively. Pairs use the LEDA memory manager and dumb pairs do not. The table was generated with program memmgr_test.c in LEDAROOT/demo/book/Impl.

⟨*timing for dumb pair*⟩≡

```
list<dumb_pair> DL;
for (i = 0; i < n; i++ ) DL.append(dumb_pair());
print_statistics();
DL.clear();
print_statistics();
UT = used_time(T);
```

13.9 Iteration

For most of its item-based data types LEDA provides iteration macros . These macros can be used to iterate over the items or elements of lists, arrays, sets, dictionaries, and priority queues or over the nodes and edges of graphs. Iteration macros can be used similarly to the C++ *for*-statement. We give some examples.

For all item-based data types:

```
forall_items(it,D) { ... }
```

iterates over the items *it* of D and

```
forall_rev_items(it,D) { ... }
```

iterates over the items *it* of D in reverse order.

For sets, lists and arrays:

```
forall(x,D) { ... }
```

iterates over the elements x of D and

```
forall_rev(x,D) { ... }
```

iterates over the elements x of D in reverse order.

For graphs:

```
forall_nodes(v,G) { ... }
```

iterates over the nodes v of G,

```
STD_MEMORY_MGR (memory status)
+--------------------------------------------------+
|   size     used     free     blocks     bytes    |
+--------------------------------------------------+
|    12    1000001      388      1469   12004668    |
|    16    1000000      110      1961   16001760    |
|    20         29      379         1       8160   |
|    28          1      290         1       8148   |
|    40          2      201         1       8120   |
| > 255          -        -         1        300   |
+--------------------------------------------------+
|   time:  0.64 sec             space:27450.88 kb |
+--------------------------------------------------+


STD_MEMORY_MGR (memory status)
+--------------------------------------------------+
|   size     used     free     blocks     bytes    |
+--------------------------------------------------+
|    12          1  1000388      1469   12004668    |
|    16          0  1000110      1961   16001760    |
|    20         29      379         1       8160   |
|    28          1      290         1       8148   |
|    40          2      201         1       8120   |
| > 255          -        -         1        300   |
+--------------------------------------------------+
|   time:  0.98 sec             space:27450.88 kb |
+--------------------------------------------------+
```

Figure 13.3 Statistic of memory usage. We built a list of $n = 10^6$ pairs of doubles. A list of n pairs requires n list items of 12 bytes each and n pairs of 16 bytes each. The upper statistic shows the situation before the clear operations and the lower statistic shows the situation after the clear operations. The figure was generated with program memmgr_test.c in LEDAROOT/demo/book/Impl.

```
forall_edges(e,G) { ... }
```

iterates over the edges e of G,

```
forall_adj_edges(e,v) { ... }
```

iterates over all edges e adjacent to v, and

```
forall_adj_nodes(u,v) { ... }
```

iterates over all nodes e adjacent to v.

Inside the body of a forall loop insertions into or deletions from the collection iterated over are not allowed, with one exception, the current item or object of the iteration may be removed, as in

```
// remove self-loops
forall_edges(e,G) { if (G.source(e) == G.target(e)) G.del_edge(e); }
```

The *forall_item*(it, S) iteration macro can be applied to instances S of all item-based data types T that define $T::item$ as the corresponding item type and that provide the following member functions:

```
T::item  S.first_item()
```

returns the first item of S and *nil* if S is empty

```
T::item  S.next_item(T::item it)
```

returns the successor of item *it* in S (*nil* if $it = S.last\_item()$ or $it = nil$).

The *forall_rev_items*(it, S) macro can be used if the following member functions are defined:

```
T::item  S.last_item()
```

returns the last item of S and *nil* if S is empty, and

```
T::item  S.pred_item(T::item it)
```

returns the predecessor of item *it* in S (*nil* if $it = S.first\_item()$ or $it = nil$).

The *forall*(x, S) and *forall_rev*(x, S) iteration macros in addition require that the operation *S.inf*$(T::item\ it)$ is defined and returns the information associated with item *it*.

A first try of an implementation of the *forall_items* macro could be

```
#define forall_items(it,S)\
for(it = S.first_item(); it != nil; it = S.next_item(it))
```

However, with this implementation the current item of the iteration cannot be removed from S. To allow this operation we use a temporary variable p always containing the successor item of the current item *it*. Since our macro has to work for all item-based LEDA data types, the item type (e.g., *dic_item* for dictionaries) is not known explicitly, but is given implicitly by the type of the variable *it*. We therefore use a temporary iterator p of type $void*$ and a function template *LoopAssign*$(item\_type\&\ it, void * p)$ to copy the contents of p to *it* before each execution of the for-loop body. The details are given by the following piece of code.

```
template <class T>
inline bool LoopAssign(T& it, void* p) { it = (T)p; }

#define forall_items(it,S)\
for( void* p = S.first_item(); \
    LoopAssign(it,p), p = S.next_item(it), it != nil; )

#define forall__rev_items(it,S)\
for( void* p = S.last_item();  \
    LoopAssign(it,p), p = S.pred_item(it), it != nil; )
```

With the above implementation of the *forall_items* loop the current item (but not its successor) may be deleted. There are many situations where this is desirable.

The following piece of code deletes all occurrences of a given number x from a list L of integers:

```
list_item it;
forall_items(it,L) if (L[it] == x) L.del_item(it);
```

The following piece of code removes self-loops from a graph G:

```
edge e;
forall_adj_edges(e,G) if (source(e) == target(e)) G.del_edge(e);
```

Exercises for 13.9

1 Design a forall macro allowing insertions at the end of the collection.

2 Implement an iteration macro for the binary tree class *bin_tree* traversing the nodes in in-order.

13.10 **Priority Queues by Fibonacci Heaps (A Complete Example)**

We give a comprehensive example that illustrates most of the concepts introduced in this chapter, the implementation of the priority queue data type *p_queue*$<P, I>$ by Fibonacci heaps. The data type *p_queue*$<P, I>$ was discussed in Section 5.4 and is defined in the header file <*LEDA*/*p_queue.h*>. We show the header file below, but without the manual comments that generate the manual page.

We call the implementation class *PRIO_IMPL*. There is one slight anomaly in the derivation of *p_queue*$<P, I>$ from *PRIO_IMPL*: What is called *prio*rity in the data type template is called *key* in the implementation class, since in the first version of LEDA priorities were called keys and this still shows in the implementation class.

13.10.1 ***The Data Type Template***

We start with the data type template.

⟨*p_queue.h*⟩≡

```
#define PRIO_IMPL f_heap

typedef PRIO_IMPL::item pq_item;

template<class P, class I>
class p_queue: private PRIO_IMPL
{
  int  key_type_id()          const { return leda_type_id((P*)0); }
  int  cmp(GenPtr x, GenPtr y) const
       { return compare(LEDA_ACCESS(P,x),LEDA_ACCESS(P,y)); }

  void clear_key(GenPtr& x)    const { leda_clear(LEDA_ACCESS(P,x)); }
  void clear_inf(GenPtr& x)    const { leda_clear(LEDA_ACCESS(I,x)); }
  void copy_key(GenPtr& x)     const { x = leda_copy(LEDA_ACCESS(P,x)); }
  void copy_inf(GenPtr& x)     const { x = leda_copy(LEDA_ACCESS(I,x)); }

public:
  p_queue()  {}
  p_queue(const p_queue<P,I>& Q):PRIO_IMPL(Q) {}
 ~p_queue()  { PRIO_IMPL::clear(); }

  p_queue<P,I>& operator=(const p_queue<P,I>& Q)
          { PRIO_IMPL::operator=(Q); return *this; }

  P       prio(pq_item it) const
          { return LEDA_CONST_ACCESS(P,PRIO_IMPL::key(it)); }
  I       inf(pq_item it)  const
          { return LEDA_CONST_ACCESS(I,PRIO_IMPL::inf(it)); }
  pq_item find_min()       const { return PRIO_IMPL::find_min(); }
  void    del_min()              { PRIO_IMPL::del_min(); }
  void    del_item(pq_item it)   { PRIO_IMPL::del_item(it); }

  pq_item insert(const P& x, const I& i)
          { return PRIO_IMPL::insert(leda_cast(x),leda_cast(i)); }

  void    change_inf(pq_item it, const I& i)
          { PRIO_IMPL::change_inf(it,leda_cast(i)); }

  void    decrease_p(pq_item it, const P& x)
          { PRIO_IMPL::decrease_key(it,leda_cast(x)); }

  int     size()  const { return PRIO_IMPL::size(); }
  bool    empty() const { return (size()==0) ? true : false; }
  void    clear()       { PRIO_IMPL::clear(); }

  pq_item first_item()          const { return PRIO_IMPL::first_item(); }
  pq_item next_item(pq_item it) const { return PRIO_IMPL::next_item(it); }
};
```

Every implementation class *PRIO_IMPL* for *p_queue*<*P*, *I*> has to provide the following operations and definitions.

```
class PRIO_IMPL
{
  virtual int  key_type_id()       const = 0;
  virtual int  cmp(GenPtr, GenPtr) const = 0;
  virtual void clear_key(GenPtr&)  const = 0;
  virtual void clear_inf(GenPtr&)  const = 0;
  virtual void copy_key(GenPtr&)   const = 0;
  virtual void copy_inf(GenPtr&)   const = 0;

public:

  typedef ... item;

protected:

  PRIO_IMPL();
  PRIO_IMPL(const PRIO_IMPL&);

  virtual ~PRIO_IMPL();

  PRIO_IMPL& operator=(const PRIO_IMPL&);

  item insert(GenPtr,GenPtr);
  item find_min() const;

  GenPtr key(item) const;
  GenPtr inf(item) const;

  void del_min();
  void del_item(item);
```

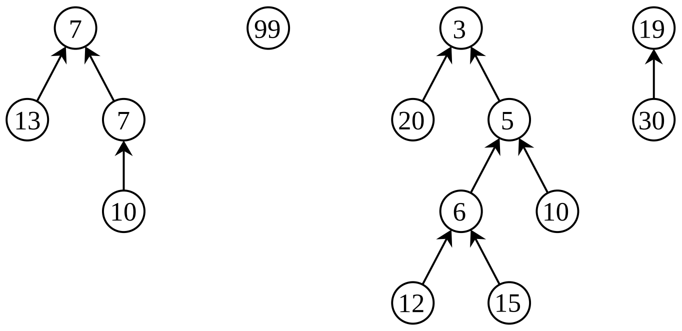

Figure 13.4 A heap-ordered forest.

```
    void decrease_key(item,GenPtr);
    void change_inf(item,GenPtr);
    void clear();
    int  size()  const;
    //iteration
    item first_item() const;
    item next_item(item) const;
  };
```

13.10.2 ***Fibonacci Heaps***

In the remainder of this section we give the Fibonacci heap realization of *PRIO_IMPL*.

Definition and Header File: Fibonacci heaps (class *f_heap*) are one of the best realizations of priority queues [FT87]. They represent priority queues as heap-ordered forests. The items of the priority queue are in one-to-one correspondence to the *nodes* of the forest; so it makes sense to talk about the key and the information of a node. A forest is *heap-ordered* if each tree in the forest is *heap-ordered*, and a tree is heap-ordered if the key of every non-root node is no less than the key of the parent of the node. In other words, the sequence of keys along any root to leaf path is non-decreasing. Figure 13.4 shows a heap-ordered forest.

In the storage representation of *f_heap*s every node contains a pointer to its parent (the parent pointer of a root is *nil*) and to one of its children. The child-pointer is *nil* if a node has no children. The children of each node and also the roots of the trees in a *f_heap* form a doubly-linked circular list (pointers *left* and *right*). In addition, every node contains the four fields *rank*, *marked*, *next*, and *pred*. The *rank* field of each node contains the number of children of the node and the *marked* field is a boolean flag whose purpose will be made clear below. The *next* and *pred* fields are used to keep all nodes of a Fibonacci heap in a doubly-linked linear list. This list is needed for the *forall_items*-iteration. An *f_heap*-item (type *F_heap*::*item*) is a pointer to a node. Figure 13.5 shows the storage representation of the heap-ordered forest of Figure 13.4.

The constructor of class *f_heap_node* creates a new node $\langle k, i \rangle$ and initializes some of the

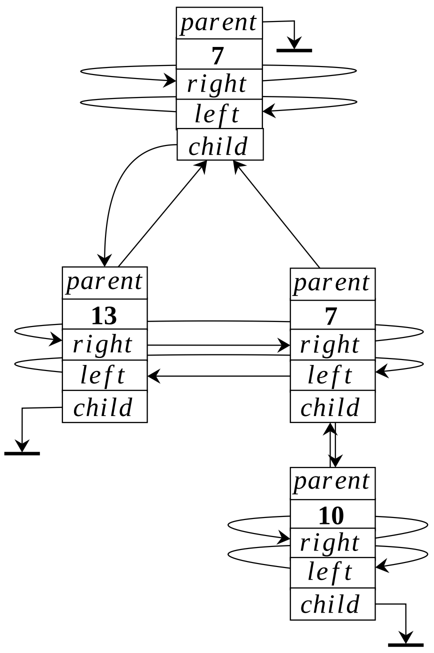

Figure 13.5 The storage representation of the heap-ordered forest of Figure 13.4. The *key*, *rank*, *marked*, *next*, and *pred* fields are not shown, informations are integers and nil-pointers are shown as pointing to "ground".

fields to their obvious values. It also adds the new item to the front of the list of all items of the heap. The LEDA memory management is used for *f_heap_node*s (cf. Section 13.8).

⟨*f_heap.h*⟩≡

```
#include <LEDA/basic.h>
class f_heap_node;
typedef f_heap_node* f_heap_item;
class f_heap_node {
  friend class f_heap;
  f_heap_item left;     // left and right siblings (circular list)
  f_heap_item right;
  f_heap_item parent;   // parent node
  f_heap_item child;    // a child
  f_heap_item next;     // list of all items
  f_heap_item pred;
  int  rank;            // number of children
  bool marked;          // mark bit
  GenPtr key;           // key
  GenPtr inf;           // information
f_heap_node(GenPtr k, GenPtr info, f_heap_item n)
{
    // the third argument n is always the first item in the list
    // of all items of a Fibonacci heap. The new item is added
    // at the front of the list

    key = k;
    inf = info;
    rank = 0;
    marked = false;
    parent = child = nil;
    next = n;
    if (n) n->pred = this;
  }

    LEDA_MEMORY(f_heap_node)
  };
```

The storage representation of an *f_heap* consists of five fields:

number_of_nodes	the number of nodes in the heap
power	the smallest power of two greater than or equal to *number_of_nodes*
logp	the binary logarithm of power
minptr	a pointer to a root with minimum key
node_list	first element in the list of all nodes

⟨*f_heap.h*⟩+≡

```
  class f_heap  {

    int number_of_nodes;
    int power;
    int logp;

    f_heap_item minptr;
    f_heap_item node_list;

    ⟨virtual functions related to keys and infs⟩
    ⟨auxiliary functions⟩

  public:

    typedef f_heap_item item;

  protected:

    // constructors, destructor, assignment
    f_heap();
    f_heap(const f_heap&);
    f_heap& operator=(const f_heap&);
    virtual ~f_heap();

    // priority queue operations
    f_heap_item insert(GenPtr, GenPtr);
    f_heap_item find_min()  const;

    void   del_min();
    void   decrease_key(f_heap_item,GenPtr);
    void   change_inf(f_heap_item,GenPtr);
    void   del_item(f_heap_item);
    void   clear();

    GenPtr key(f_heap_item) const;
    GenPtr inf(f_heap_item) const;
    int    size() const;
    bool   empty() const;
    // iteration
    f_heap_item first_item() const;
    f_heap_item next_item(f_heap_item) const;
  };
```

We turn to the implementation of the member functions. The file _f_heap.c contains the implementations of all operations on *f_heap*s.

Construction: To create an empty *f_heap* set *number_of_nodes* to zero, *power* to one, *logp* to zero, and *minptr* and *node_list* to *nil*.

⟨*_f_heap.c*⟩≡

```
  #include <LEDA/basic.h>
  #include "f_heap.h"
  f_heap::f_heap()
  { number_of_nodes = 0;
    power = 1;
    logp = 0;
    minptr = nil;
    node_list = nil;
  }
```

Simple Operations on Heaps: We discuss create, findmin, size, empty, key, inf, and change_key. A *find_min* operation simply returns the item pointed to by *minptr*. The empty operation compares *number_of_nodes* to zero, and the *size* operation returns *number_of_nodes*. Both operations take constant time.

The *key* and *inf* operations apply to an item and return the appropriate component of the item.

The *change_inf* operations applies to an item *x* and an information *inf* and changes the information associated with *x* to a copy of *inf*. It also clears the memory used for the old information.

⟨*_f_heap.c*⟩+≡

```
  f_heap_item f_heap::find_min()          const { return minptr; }
  int         f_heap::size()              const { return number_of_nodes; }
  bool        f_heap::empty()             const
                                           { return number_of_nodes == 0; }
  GenPtr      f_heap::key(f_heap_item x) const { return x->key; }
  GenPtr      f_heap::inf(f_heap_item x) const { return x->inf; }
  void f_heap::change_inf(f_heap_item x, GenPtr i)
  { clear_inf(x->inf);
    copy_inf(i);
    x->inf = i;
  }
```

We have used functions *clear_key* and *copy_key* without defining them. Both functions belong to the set of virtual functions of class *f_heap* which we need to make *f_heap* a parameterized data structure. We declare these functions as pure virtual and define them in the definition of the class *p_queue*$<K, I>$ as discussed in Section 13.4.

The six virtual functions are: *cmp* compares two keys (of type P), *clear_key* and *clear_inf* deallocate a key and an information, respectively, *copy_key* and *copy_inf* return a copy of their argument, and *key_type_id*( ) determines whether its argument belongs to a built-in type as discussed in Section 13.5. It is used to bypass the calls to compare function for such types.

⟨*virtual functions related to keys and infs*⟩≡
```
  virtual int    cmp(GenPtr,GenPtr)     const = 0;
  virtual void   clear_key(GenPtr&)     const = 0;
  virtual void   clear_inf(GenPtr&)     const = 0;
  virtual GenPtr copy_key(GenPtr&)      const = 0;
  virtual GenPtr copy_inf(GenPtr&)      const = 0;
  virtual int    key_type_id()          const = 0;
```

Some Theory: The non-trivial operations are *insert*, *decrease_inf* and *del_min*. We discuss them in some detail now. The discussion will be on the level of heap-ordered forests. All implementation details will be given later.

An insert adds a new single node tree to the Fibonacci heap and, if necessary, adjusts the *minptr*. So a sequence of n inserts into an initially empty heap will simply create n single node trees. The cost of an insert is clearly $O(1)$.

A *del_min* operation removes the node indicated by *minptr*. This turns all children of the removed node into roots. We then scan the set of roots (old and new) to find the new minimum. To find the new minimum we need to inspect all roots (old and new), a potentially very costly process. We make the process even more expensive (by a constant factor) by doing some useful work on the side, namely combining trees of equal rank into larger trees. A simple method to combine trees of equal rank is as follows. Let *max_rank* be the maximal rank of any node. Maintain a set of buckets, initially empty and numbered from 0 to *max_rank*. Then step through the list of old and new roots. When a root of rank i is considered inspect the i-th bucket. If the i-th bucket is empty then put the root there. If the bucket is non-empty then combine the two trees into one (by making the root with the larger information a child of the other root). This empties the i-th bucket and creates a root of rank $i + 1$. Try to throw the new tree into the $i + 1$st bucket. If it is occupied, combine .... When all roots have been processed in this way, we have a collection of trees whose roots have pairwise distinct ranks. What is the running time of the *del_min* operation?

Let K denote the number of roots before the call of *del_min*. The cost of the operation is $O(K + \mathit{max\_rank})$ (since the deleted node has at most *max_rank* children and hence there are at most $K + \mathit{max\_rank}$ roots to start with. Moreover, every combine reduces the number of roots by one). After the call there will be at most *max_rank* roots (since they all have

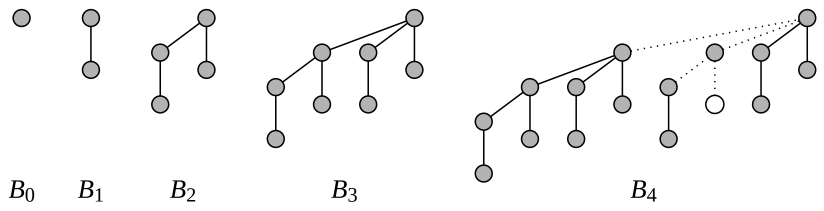

Figure 13.6 Binomial trees. Deletion of the high-lighted node and all high-lighted edges decomposes B_4 into binomial trees.

different ranks) and hence the number of roots decreases by at least $K - \mathit{max\_rank}$. Thus, if we use the potential function Φ_1 with

$$\Phi_1 = \text{number of roots}$$

then the amortized cost of a *delete_min* operation is $O(\mathit{max\_rank})$. The amortized cost of an insert is $O(1)$; note that n inserts increase the potential Φ_1 by one. We will extend the potential by a second term Φ_2 below.

What can we say about the maximal rank of a node in a Fibonacci heap? Let us consider a very simple situation first. Suppose that we perform a sequence of inserts followed by a single *del_min*. In this situation, we start with a certain number of single node trees and all trees formed by combining are so-called *binomial trees* as shown in Figure 13.6. The binomial tree B_0 consists of a single node and the binomial tree B_{i+1} is obtained by joining two copies of the tree B_i. This implies that the root of the tree B_i has rank i and that the tree B_i contains exactly 2^i nodes. We conclude that the maximal rank in a binomial tree is logarithmic in the size of the tree. If we could guarantee in general that the maximal rank of any node is logarithmic in the size of the tree then the amortized cost of the *del_min* operation would be logarithmic.

We turn to the *decrease_key* operation next. It is given a node v and a new information *newkey* and decreases the information of v to *newkey*. Of course, *newkey* must not be larger than the old information associated with v. Decreasing the information associated with v will in general destroy the heap property. In order to maintain the heap property we delete the edge connecting v to its parent and turn v into a root. This has the side effect that for any ancestor w of v different from v's parent the size of w's subtree decreases by one but w's rank is unchanged. Thus, if we want to maintain the property that the maximal rank of any node is logarithmic in the size of the subtree rooted at the node, we need to do more than just cutting v's link to its parent.

An old solution suggested by Vuillemin [Vui78] is to keep all trees in the heap binomial. This can be done as follows: for any proper ancestor z of v delete the edge into z on the path from v to z, call it e, and all edges into z that were created later than e. In Figure 13.6 a node and a set of edges is high-lighted in the tree B_4. If all high-lighted edges are removed then B_4 decomposes into two copies of B_0 and one copy each of B_1, B_2, and B_3. It is not too hard to see that at most k edges are removed when a B_k is disassembled (since a B_k

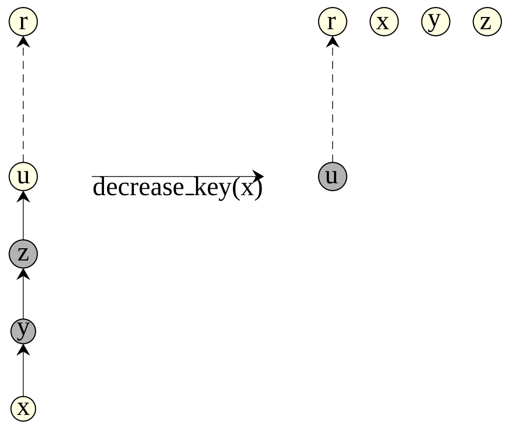

Figure 13.7 A decrease key on x is performed and y and z are marked but u is not; x, y, and z become roots, roots are unmarked, and u becomes marked. Marked nodes are shown shaded. A dashed edge stands for a path of edges.

decomposes into two B_j's and one each of $B_{j+1}, \ldots, B_{k-1}$ for some j, with $0 \leq j \leq k-1$) and hence this strategy gives a logarithmic time bound for the *decrease_key* operation.

In some graph algorithms the *decrease_key* operation is executed far more often than the other priority queue operations, e.g., Dijkstra's shortest-path algorithm (cf. Section 6.6) executes m *decrease_key*s and only n *insert*s and *del_min*s, where m and n are the number of edges and nodes of the graph, respectively. Since m might be as large as n^2 it is desirable to make the *decrease_key* operation cheaper than the other operations. Fredman and Tarjan showed how to decrease its cost to $O(1)$ without increasing the cost of the other operations. Their solution is surprisingly simple and we describe it next.

When a node x loses a child because *decrease_key* is applied to the child the node x is marked; this assumes that x has not already been marked. When a marked node x loses a child, we turn x into a root, remove the mark from x and attempt to mark x's parent. If x's parent is marked already then .... In other words, suppose that we apply *decrease_key* to a node v and that the k-nearest ancestors of v are marked, then turn v and the k-nearest ancestors of v into roots and mark the $k+1$st-nearest ancestor of v (if it is not a root). Also unmark all the nodes that were turned into roots, cf. Figure 13.7. Why is this a good strategy?

First, a *decrease_key* marks at most one node and unmarks some number k of nodes. No other operation marks a node and hence in an amortized sense k can be at most one (we cannot unmark more nodes than we mark). However, we also increase the number of roots by k which in turn increases the potential Φ_1 by k and therefore we have to argue more carefully. Let

$$\Phi_2 = 2 \cdot \text{number of marked nodes}$$

and let $\Phi = \Phi_1 + \Phi_2$. A *decrease_key* operation where the node v has k marked ancestors has actual cost $O(k+1)$ and decreases the potential by at least $2(k-1)-(k+1)=k-3$. Note that the number of marked nodes is decreased by at least $k-1$ (at least k nodes are unmarked and at most one node is marked) and that the number of roots is increased by $k+1$. The amortized cost of a *decrease_key* is therefore $O(1)$. *insert*s do not change Φ_2 and *del_min*s do not increase Φ_2 (it may decrease it because the marked children of the removed node become unmarked roots) and hence their amortized cost does not increase by the introduction of Φ_2.

How does the strategy affect the maximal rank. We show that it stays logarithmic. In order to do so we need some notation. Let $F_0=0$, $F_1=1$, and $F_i=F_{i-1}+F_{i-2}$ for $i\geq 2$ be the sequence of Fibonacci numbers. It is well-known that $F_{i+1}\geq(1+\sqrt{5}/2)^i\geq 1.618^i$ for all $i\geq 0$.

Lemma 1 *Let v be any node in a Fibonacci heap and let i be the rank of v. Then the subtree rooted at v contains at least F_{i+2} nodes. In a Fibonacci heap with n nodes all ranks are bounded by* $1.4404\log n$.

Proof Consider an arbitrary node v of rank i. Order the children of v by the time at which they were made children of v. Let w_j be the j-th child, $1\leq j\leq i$. When w_j was made child of v both nodes had the same rank. Also, since at least the nodes $w_1,\ldots,w_{j-1}$ were nodes of v at that time, the rank of v was at least $j-1$ at the time when w_j was made a child of v. The rank of w_j has decreased by at most 1 since then because otherwise w_j would be a root. Thus the current rank of w_j is at least $j-2$.

We can now set up a recurrence for the minimal number S_i of nodes in a tree whose root has rank i. Clearly $S_0=1$, $S_1=2$, and $S_i\geq 2+S_0+S_1+\ldots+S_{i-2}$. The last inequality follows from the fact that for $j\geq 2$, the number of nodes in the subtree with root w_j is at least S_{j-2}, and that we can also count the nodes v and w_1. The recurrence above (with = instead of $\geq$) generates the sequence 1, 2, 3, 5, 8,... which is identical to the Fibonacci sequence (minus its first two elements).

Let's verify this by induction. Let $T_0=1$, $T_1=2$, and $T_i=2+T_0+\ldots+T_{i-2}$ for $i\geq 2$. Then, for $i\geq 2$, $T_{i+1}-T_i=2+T_0+\ldots+T_{i-1}-2-T_0-\ldots-T_{i-2}=T_{i-1}$, i.e., $T_{i+1}=T_i+T_{i-1}$. This proves $T_i=F_{i+2}$.

For the second claim, we only have to observe that $F_{i+2}\leq n$ implies $i\cdot\log(1+\sqrt{5}/2)\leq\log n$ which in turn implies $i\leq 1.4404\log n$. $\square$

This concludes our theoretical treatment of Fibonacci heaps. We have shown the following time bounds: an *insert* and a *decrease_key* take constant amortized time and a *del_min* takes logarithmic amortized time. The operations *size*, *empty*, and *findmin* take constant time.

We now return to the implementation.

Insertions: An *insert* operation takes a key k and an information i and creates a new heap-ordered tree consisting of a single node $\langle k,i\rangle$. In order to maintain the representation invariant it must also add the new node to the circular list of roots, increment *number_of_nodes*, and may be *power* and *logp*, and change *minptr* if k is smaller than the current minimum key in the queue.

⟨*_f_heap.c*⟩+≡

```
f_heap_item f_heap::insert(GenPtr k, GenPtr i)
{
  k = copy_key(k);
  i = copy_inf(i);
  f_heap_item new_item = new f_heap_node(k,i,node_list);
  if ( number_of_nodes == 0 )
  { // insertion into empty queue
    minptr = new_item;
    // build trivial circular list
    new_item->right = new_item;
    new_item->left = new_item;
    // power and logp have already the correct value
  }
  else
  { // insertion into non-empty queue;
    // we first add to the list of roots
    new_item->left = minptr;
    new_item->right = minptr->right;
    minptr->right->left = new_item;
    minptr->right = new_item;
    if ( cmp(k,minptr->key) < 0 ) minptr = new_item; // new minimum
    if ( number_of_nodes >= power) // log number_of_nodes grows by one
    { power = power * 2;
      logp = logp + 1;
    }
  }
  number_of_nodes++;
  return new_item;
}
```

Delete_min: A *del_min* operation removes the item pointed to by *minptr*, i.e., an item of minimum *key*. This turns all children of the removed node into roots. We then scan the set of roots (old and new) to find the new minimum.

⟨*_f_heap.c*⟩+≡

```
void f_heap::del_min()
{ // removes the item pointed to by minptr
  if ( minptr == nil )
    error_handler(1,"f_heap: deletion from empty heap");
  number_of_nodes--;
  if ( number_of_nodes==0 )
  { // removal of the only node
    // power and logp do not have to be changed.
```

```
      clear_key(minptr->key);
      clear_inf(minptr->inf);
      delete minptr;
      minptr = nil;
      node_list = nil;
      return;
    }
    /* removal from a queue with more than one item. */
    ⟨turn children of minptr into roots⟩;
    ⟨combine trees of equal rank and compute new minimum⟩;
    ⟨remove old minimum⟩;
  }
```

We now discuss the removal of a node of minimum *key* from an *f_heap* with more than one item. Recall that *number_of_nodes* already has its new value. We first update *power* and *logp* (if necessary) and then turn all children of *minptr* into roots (by setting their parent pointer to nil and their mark bit to false and combining the list of children of *minptr* with the list of roots). We do not delete *minptr* yet. It is convenient to keep it as a sentinel.

The cost of turning the children of the *minptr* into roots is $O(maxrank)$;

Note that the body of the loop is executed for each child of the node *minptr* and that, in addition, to the children of *minptr* we access *minptr* and its right sibling.

⟨*turn children of minptr into roots*⟩≡

```
  if ( 2 * number_of_nodes <= power )
  { power = power / 2;
    logp = logp - 1;
  }
  f_heap_item r1 = minptr->right;
  f_heap_item r2 = minptr->child;
  if ( r2 )
  { // minptr has children
    while ( r2->parent )
    { //  visit them all and make them roots
      r2->parent = nil;
      r2->marked = false;
      r2 = r2->right;
    }
    // combine the lists, i.e. cut r2's list between r2 and its left
    // neighbor and splice r2 to minptr and its left neighbor to r1
    r2->left->right = r1;
    r1->left = r2->left;
    minptr->right = r2;
    r2->left = minptr;
  }
```

The task of the combining phase is to combine roots of equal rank into larger trees. The combining phase uses a procedure *link* which combines two trees of equal rank and returns the resulting tree.

⟨*_f_heap.c*⟩+≡

```
f_heap_item f_heap::link(f_heap_item r1, f_heap_item r2)
{
  // r1 and r2 are roots of equal rank, both different from minptr;
  // the two trees are combined and the resulting tree is returned.

  f_heap_item h1;
  f_heap_item h2;

  if (cmp(r1->inf,r2->inf) <= 0)
   { // r2 becomes a child of r1
     h1 = r1;
     h2 = r2;
   }
   else
   { // r1 becomes a child of r2
     h1 = r2;
     h2 = r1;
   }

  // we now make h2 a child of h1. We first remove h2 from
  // the list of roots.

  h2->left->right = h2->right;
  h2->right->left = h2->left;

  /* we next add h2 into the circular list of children of h1 */

  if ( h1->child == nil )
  { // h1 has no children yet; so we make h2 its only child
    h1->child = h2;
    h2->left = h2;
    h2->right = h2;
  }
  else
  { // add h2 to the list of children of h1
    h2->left = h1->child;
    h2->right = h1->child->right;
    h1->child->right->left = h2;
    h1->child->right = h2;
  }

  h2->parent = h1;
  h1->rank++;

  return h1;
}
```

Let's not forget to add the declaration of link to the set of auxiliary functions of *class f_heap*.

⟨*auxiliary functions*⟩≡

```
f_heap_item link(f_heap_item, f_heap_item);
```

Next comes the code to combine trees of equal rank. The task is to step through the list of old and new roots, to combine roots of equal rank, and to determine the node of minimum key. We solve this task iteratively. We maintain an array *rank_array* of length *maxrank* of pointers to roots: *rank_array*$[i]$ points to a root of rank i, if any and to *nil* otherwise.

Initially all entries point to *nil*. When a root of rank r is inspected and *rank_array*$[r]$ is *nil*, store r there. If it is non-empty, combine r with the array entry and replace r by the combined tree. The combined tree has rank one higher. We declare *rank_array* as an array of length $12 * sizeof(int)$. This is a save choice since the number of nodes in a heap is certainly bounded by $MAXINT = 2^{8*sizeof(int)}$. Hence $maxrank \leq 1.5 * \log(MAXINT) = 12 * sizeof(int)$.

There is a small subtlety in the following piece of code. We are running over the list of roots and simultaneously modifying it. This is potentially dangerous, but our strategy is safe. Imagine the list of roots drawn with the *minptr* at the far right. Then *current* points to the leftmost element initially. At a general step of the iteration *current* points at some arbitrary list element. All modifications of the list by calls of *link* take place strictly to the left of *current*. For this reason it is important to advance *current* at the beginning of the loop.

⟨*combine trees of equal rank and compute new minimum*⟩≡

```
f_heap_item rank_array[12*sizeof(int)];
for (int i = (int)1.5*logp; i >= 0; i--) rank_array[i] = nil;

f_heap_item new_min = minptr->right;
f_heap_item current = new_min;

while (current != minptr)
{ // old min is used as a sentinel
  r1 = current;
  int rank = r1->rank;
  // it's important to advance current already here
  current = current->right;

  while (r2 = rank_array[rank])
  { rank_array[rank] = nil;
    // link combines trees r1 and r2 into a tree of rank one higher
    r1 = link(r1,r2);
    rank++;
  }
  rank_array[rank] = r1;
  if ( cmp(r1->inf,new_min->inf) <= 0 ) new_min = r1;
}
```

We complete the operation by actually deleting the old minimum and setting *minptr* to its new value.

⟨*remove old minimum*⟩≡

```
minptr->left->right = minptr->right;
minptr->right->left = minptr->left;

clear_key(minptr->key);
clear_inf(minptr->inf);

r1 = minptr->pred;
r2 = minptr->next;
if (r2) r2->pred = r1;
```

```
  if (r1) r1->next = r2; else node_list = r2;
  delete minptr;
  minptr = new_min;
```

Decrease_key, Clear, and Del_item: *decrease_key* makes use of an auxiliary function *cut*(x) that turns a non-root node x into a root and returns its old parent.

⟨*auxiliary functions*⟩+≡

```
  f_heap_item cut(f_heap_item);
```

⟨*_f_heap.c*⟩+≡

```
  f_heap_item f_heap::cut(f_heap_item x)
  {
    f_heap_item y = x->parent;
    if ( y->rank == 1 ) y->child = nil;  // only child
    else
    { /* y has more than one child. We first make sure that its childptr
         does not point to x and then delete x from the list of children */
      if ( y->child == x ) y->child = x->right;
      x->left->right = x->right;
      x->right->left = x->left;
    }
    y->rank--;
    x->parent = nil;
    x->marked = false;
    // add to circular list of roots
    x->left = minptr;
    x->right = minptr->right;
    minptr->right->left = x;
    minptr->right = x;
    return y;
  }
```

Now we can give the implementation of *decrease_key*.

⟨*_f_heap.c*⟩+≡

```
  void f_heap::decrease_key(f_heap_item v, GenPtr newkey)
  {
    /* changes the key of f_heap_item v to newkey;
       newkey must be no larger than the old key;
       if newkey is no larger than the minimum key
       then v becomes the target of the minptr  */
    if (cmp(newkey,v->key) > 0)
      error_handler(1,"f_heap: key too large in decrease_key.");
    // change v's key
    clear_key(v->key);
    v->key = copy_key(newkey);
    if ( v->parent )
    { f_heap_item x = cut(v);           // make v a root
      while (x->marked) x = cut(x);     // a marked f_heap_node
                                        // is a non-root
      if (x->parent) x->marked = true; // mark x if it not a root
    }
    // update minptr (if necessary)
    if (cmp(newkey,minptr->key) <= 0) minptr = v;
  }
```

To clear a heap simply remove the minimum until the heap is empty. The cost of *clear* is bounded by n times the cost of *del_min*. We can also use *clear* as the destructor of class *f_heap*.

⟨*_f_heap.c*⟩+≡

```
  void   f_heap::clear()  { while (number_of_nodes > 0) del_min(); }
  f_heap::~f_heap() { clear(); }
```

To remove an arbitrary item from a heap, we first decrease its *key* to the minimum key (this makes the item the target of the *minptr*) and then remove the minimum. The cost of removing an item is therefore bounded by $O(1)$ plus the cost of *decrease_key* plus the cost of *del_min*.

⟨*_f_heap.c*⟩+≡

```
  void f_heap::del_item(f_heap_item x)
  { decrease_key(x,minptr->key);     // the minptr now points to x
    del_min();
  }
```

Assignment, Iteration, and Copy Constructor: Next comes the assignment operator. In order to execute $S = H$ we simply step through all the items of H and insert their key and information into S. We must guard against the trivial assignment $H = H$.

⟨*_f_heap.c*⟩+≡

```
  f_heap& f_heap::operator=(const f_heap& H)
  { if (this != &H)
    { clear();
      for (f_heap_item p = H.first_item(); p; p = H.next_item(p))
        insert(p->key,p->inf);
    }
    return *this;
  }
```

The assignment operator makes use of the two functions *first_item* and *next_item*. They allow us to iterate over all items of a heap. We use these functions in the assignment operator, the copy constructor, and the *forall_items*-iteration. The last use forces us to make both functions public members of the class. However, we do not list them in the manual and so they are only semi-public. For this reason *next_item* does not check whether its argument is distinct from *nil*.

⟨*_f_heap.c*⟩+≡

```
f_heap_item f_heap::first_item() const { return node_list; }

f_heap_node* f_heap::next_item(f_heap_node* p) const
                                  { return p ? p->next : 0; }
```

The last operation to implement is the copy constructor. It makes a copy of its argument H. The strategy is simple. For each item of H we create a single node tree with the same key and information.

There is a subtle point in the implementation. When a virtual function is applied to an object under construction then the default implementation of the function is used and not the overriding definition in the derived class. It is therefore important in the code below to call the virtual functions *copy_key*, *copy_inf* and *cmp* through the already existing object H; leaving out the prefix H. would select the default definitions (which do not do anything).

⟨*_f_heap.c*⟩+≡

```
f_heap::f_heap(const f_heap& H)
{ number_of_nodes = H.size();
  minptr = nil;
  node_list = nil;

  f_heap_item first_node = nil;

  for(f_heap_item p = H.first_item(); p; p = H.next_item(p))
  { GenPtr k = H.copy_key(p->key);
    GenPtr i = H.copy_inf(p->inf);
    f_heap_item q = new f_heap_node(k,i,node_list);
    q->right = node_list->next;
    if (node_list->next) node_list->next->left = q;
    if (minptr == nil) { minptr = q; first_node = q; }
    else if ( H.cmp(k,minptr->key) < 0 ) minptr = q;
  }
  first_node->right = node_list;
  node_list->left = first_node;
}
```

Bibliography

[ES90] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[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.

[Str91] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 1991.

[Vui78] J. Vuillemin. A data structure for manipulating priority queues. *Communication of the ACM*, 21:309–314, 1978.

Index