Contents

0.1 Linked Lists

Linked lists are a simple and yet powerful data type for representing sequences. They allow us to manipulate sequences in many ways in constant time. In particular, single elements and even entire subsequences can be inserted and deleted in constant time. Access to list elements is through pointers or handles.

There are many versions of linked lists: singly linked, doubly linked, circular, etc. We discuss singly linked lists in depth in Sections 0.1.1 to **??** and briefly touch upon other other variants in Section 0.1.6 and in the exercises. The orthogonal issue of how to manage memory for list data structures is treated in Section 0.1.7.

0.1.1 Singly Linked lists

In singly linked lists, every element knows its successor; in doubly linked lists elements also know their predecessor. Thus singly linked lists can be traversed from front to rear and double linked lists can be traversed in both directions. This makes doubly linked lists more flexible but also more space consuming.

We will discuss implementations of singly linked lists; we discuss two implementations called *Sslist* and *Mslist*, respectively, as the two authors prefer different implementations of the singly linked list type. We will introduce the functionality of lists as we go along. We summarize the definitions in Sections 0.1.3 and 0.1.4 and compare the implementations in Section **??**.

The first version of these notes were written by PS. He defined *Sslists*. KM read the notes and remarked that he did not like the implementation of lists through *Sslists*. He took over and added *Mslists*. He also tried to compare the two implementations. Of course, this comparison is biased towards *Mslists*. PS did not have a chance yet to look over the revised version of the notes. KM is sure that he will add comments which are biased towards *Sslists*.

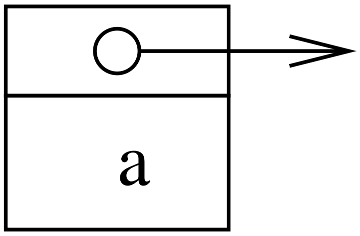

Figure 1: A list node is drawn as a box with two compartments. The upper compartment contains a pointer to another list node (drawn as an arrow) and the lower compartment contains a element of type T.

The key ingredient for the implementation is the concept of a list node. A list node (for singly linked lists representing sequences of elements of type T) is a structure consisting of two fields (see Figure 1):

- a pointer *next* to a list node, and
- a data field *inf* of type T.

In C++ notation the definition is as follows:

⟨*class node*⟩≡

```
class node {
public:
  node* next;
  T inf;
  node() { }
  node(node* x, T i) : next(x), inf(i) {}
};
```

The default constructor *node*( ) defines a node whose initial contents is unspecified and the constructor *node*(x, i) defines a node whose next-field has value x and whose *inf*-field has value i. We will use nodes within the definitions of *Sslist* and *Mslist*.

At this point it is appropriate to review the operators of C++ pertinent to structures and pointers.

```
node v;                  // declaration of a node v
&v;                      // address of v
v.inf;                   // the inf-component of v (unspecified value)
node w(&v,4);            // declaration of a node w;
                         // w.next points to v
                         // w.inf is 4
w.next;                  // the next-component of w
                         // = the address of v
node* p;                 // declaration of a pointer to node
p = &w;                  // p is made to point to w
p;                       // the value of p is the address of w
*p;                      // the node p points to, i.e., w
```

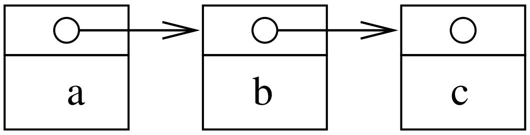

Figure 2: A list of three elements. Who points to the first element and where does the last element point to?

```
p->inf;                   // short hand for (*p).inf, i.e., the
                          // inf-field of the node p points to.
(p == &v);
                          // false, since p points to w and not to v
(p == &w);
                          // true, since p points to w
(p->next == &v);
                          // true, since w.next points to v
(p->next == w.next);      // true, since p points to w
p.next;                   // syntax error
                          // p is not a node, p points to a node
(*p) == v;                // syntax error
                          // no equality operator defined for nodes
```

It is convenient to have a pointer value which is not the address of any object. We will denote this value by *nil*; the notations *null* and *NULL* are also used in the literature. In C++ the constant 0 is not the address of any object.

⟨*structures and pointers*⟩≡

```
const int nil = 0;         // definition of nil (predefined in LEDA)
p = nil;                   // p does not point to any object
(p == nil);                // true, since p has value nil
(nil == &v);               // false, since nil is not the address of v
(p->inf);                  // runtime error (segmentation fault)
                           // p does not point to a node
```

We come back to lists. Figure 2 shows a list consisting of three elements. The figure rises two immediate questions:

1. Who points to the first element?
2. Where does the last element point to?

The answer is, this depends on the implementation of lists. We give two implementations in the text. The implementation *Mslist* is the one prefered by the first author and the implementation *Sslist* is the one prefered by the second author.

Mslists: An *Mslist*

1. contains a variable *head* of type *node∗* which points to the first element and

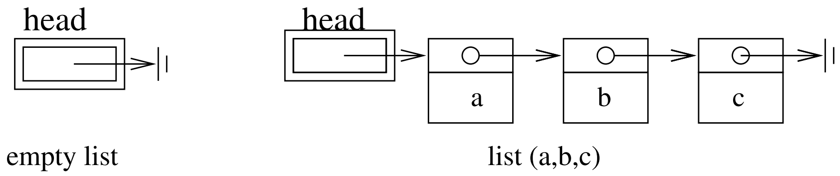

Figure 3: *Mslists*: The representations of an empty list and of the list $\langle a, b, c\rangle$; *head* is a variable of type pointer to node; the *next*-field of the last element points to *nil*. In an empty list, *head* points to *nil*.

2. the last element of the list points nowhere (its *next*-field has value nil),

see Figure 3.

⟨*Mslist*⟩≡
```
template<class T>
class Mslist {
  ⟨class node⟩
  node* head;
public:
  typedef node* handle;
  Mslist() { head = 0; }
  ~Mslist() { make_empty(); }
  ⟨operations on Mslists⟩
  ⟨operations with common implementations⟩
private:
  Mslist(const Mslist& other);            // hide copy constructor
  Mslist& operator=(const Mslist& other); // hide assignment
};
```

An *Mslist* has a single data member *head* of type pointer to node. In order to create an empty list, we simply set *head* to *nil* and in order to destroy a list we make the list empty. The operation *make_empty* will be discussed below.

The type pointer to node plays an important role in the interface to lists and in programs working with lists. It therefore deserves a shorter name. We introduce the type name *handle* for the type pointer to node. Our concept of a handle is about the same as the concept of *item* and *iterator* in LEDA and the STL, respectively.

C++ uses default definitions of the copy-constructor (component-wise copy) and the assignment operator (component-wise assignment), if the author of a class does not give own definitions. The default implementations do not have the correct semantics in our case. In order to copy a list, it does not suffice to make a copy of *head*. We leave it to the exercises to give correct definitions of the copy constructor and the assignment operator (exercise 4). We declared the copy constructor and the assignment operator private; this is the C++ way of stating that there is no copy constructor and no assignment operator.

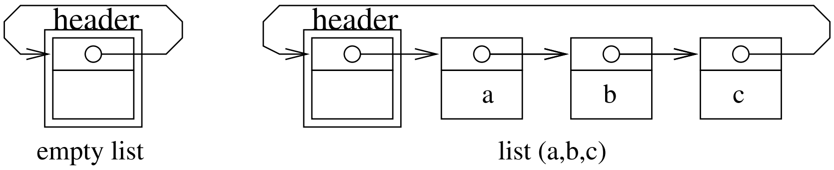

Figure 4: *Sslists*: The representations of an empty list and of the list $\langle a, b, c\rangle$; *hdr* is a list node which points to the first element and to which the last elements points to. In an empty list, *hdr* points to itself.

Sslists: An *Sslist* contains a list node *hdr*

1. which points to the first elements, and
2. to which the last elements points to,

see Figure 4. The *inf*-field of the hdr may contain an arbitrary value. We will exploit this fact in the search operation on lists.

⟨*Sslist*⟩≡
```
template<class T>
class Sslist {
  ⟨class node⟩
  node hdr;
public:
  typedef node* handle;
  Sslist() { hdr.next = &hdr; }
  ~Sslist() { make_empty(); }

  ⟨operations on Sslists⟩
  ⟨operations with common implementations⟩
private:
  Sslist(const Sslist& other);            // hide copy constructor
  Sslist& operator=(const Sslist& other); // hide assignment
};
```

An *Sslist* has a single data member *header* of type node. The default constructor makes the *next*-field of *hdr* point to *hdr*.

Simple Operations on Lists: We come to operations on lists and list elements. For each operations we will give the implementation for both list implementations.

In order to test whether a list is empty, we inspect the value of *head* in the Mslist-implementation and the value of *hdr.next* in the Sslist-implementation.

⟨*operations on Mslists*⟩≡
```
bool is_empty() const { return ( head == nil ); }
```

⟨*operations on Sslists*⟩≡

```
bool is_empty() const { return ( hdr.next == &hdr ); }
```

The next set of operations allows us to navigate through a list. We have an operations that gives us a handle to the first list element, an operation which allows us to test whether a handle points to the last element of a list, an operation which returns a handle to the successor of a list element, and an operation which gives access to the information field of a list element. All functions (except the access operator) give only read-access and hence are declared const-functions[1]

⟨*operations on Mslists*⟩+≡

```
handle first()          const { return head; }
bool is_last(handle p)  const { return ( p->next == nil ); }
handle next(handle p)   const { return p->next; }
T& operator[](handle p)       { return p->inf; }
```

If L is a list, *L.first*( ) returns a handle to the first element in the list. If the list is empty, the call will return *nil*. The call *L.next*(p) returns a handle to the successor element of p, it returns *nil*, if p points to the last element. A call *L.is_last*(p) returns true iff p points to the last element and $L[p]$ returns a reference to the information of the node handled by p.

In *Sslists*, we also have an operation which returns a handle to the header and an operation which test whether a given handle is a handle to the header[2].

⟨*operations on Sslists*⟩+≡

```
handle header()          const { return const_cast<handle>(&hdr); }
// handle header()       const { return &hdr; }  does not compile

bool is_header(handle p) const { return ( p == &hdr); }
handle first()           const { return hdr.next; }
```

[1]The const-qualifier states that the function does not change the value of any data member of the class. Const-member functions can be applied to objects that are declared `const`, non-const member functions cannot be applied to such objects. The most frequent kind of const-objects are const-parameters of functions. The use of the const-qualifier guards against the common programming mistake that a member functions is intended to leave the state unchanged, but does not do so.

[2]The function *header* returns the address of *hdr* which is a private data member of *Sslist*. This is potentially unsafe, as it might allow a user of class *Sslist* to change private data of the class in an arbitrary way. For example, it would be a disaster, if the handle to *hdr* would allow a user of *Sslists* to change the next-field of *hdr*. It is the idea of classes that changes of state can only be affected by calls of member functions. What can a user do with a pointer to *hdr*? He/she can only change the *inf*-field of *hdr*, but its value is arbitrary, he/she cannot change the *next*-field since the member function *next* gives only read-access to it. Thus it is safe to return a handle to *hdr* and to nevertheless claim that the function *header* does not open the possibility to change the state of an *Sslist* in an uncontrolled way. Of course, a C++ compiler has no way to know the reasoning above. Indeed, the C++-compiler used by the authors does not compile the function *header* if its body is given as `return &hdr`. It complains that there is a conflict with the fact that the function is declared const. The cast *const_cast*<*handle*> inserted into the body tells the compiler to accept the code anyway; in other words, it tells the compiler that we are aware of the danger and believe we know what we are doing. The argument above shows that we indeed do.

```
bool is_last(handle p)   const { return (p->next == &hdr); }
handle next(handle p)    const { return p->next; }
T& operator[](handle p)        { return p->inf; }
```

The implementations are very similar, in the last two cases even identical. However, the semantic differs slightly. If L is a list, *L.first*( ) returns a handle to the first element in the list. If the list is empty, a handle to the *header* will be returned[3]. The call *L.next*(p) returns a handle to the successor element of p; it returns a handle to the *header*, if p points to the last element. A call *L.is_last*(p) returns true iff p points to the last element and $L[p]$ returns the information of the node to which p points to. It is illegal to apply this operation to an invalid handle or to a handle to the *header*.

KM feels that the preceding paragraph is the strongest argument for the *Mslist*-implementation. A user of the *Sslist*-implementation always has to keep in mind that there is a special node in every list. KM admits that the special element simplifies the implementation of some functions on lists (see below). However, he believes that this advantage is more than out-weight by the fact, that *Sslists* make it easier to make common mistakes. Consider the following program.

```
p = L.first();
while ( true ) { p = L.next(p); }
```

It will cause a run-time error in the case of *Mslists* and it will run forever in the case of *Sslists*. Or consider

```
cout << L.[L.first()];
```

when applied to an empty list. Or consider a program to print all elements of a list L. In the case of *Mslists*, we can write[4]

```
for (handle p = L.first(); p != nil; p = L.next(p)) cout << L[p];
```

Observe, that the code also works for empty lists. In the case of *Sslists*, we have to write either

```
for (handle p = L.header(); !L.is_last(p); p = L.next(p)) cout << L[L.next(p)];
```

or

```
for (handle p = L.first(); !L.is_header(p); p = L.next(p)) cout << L[p];
```

KM feels that both versions are awkward. We could make the second version look better by adding a function

```
bool is_beyond_last(handle p) { return is_header(p); }
```

which is synonymous to *is_header*, but reflects the fact that *header* is also past the last element.

Insertions: Much of the flexibility and usefulness of lists comes from the fact that they are easily modified by insertions and deletions.

We start with a function for inserting a new element e. For singly linked lists, this is easy if we know the intended predecessor for p. We create a new node x, let $p \rightarrow next$ point to it, and set $x \rightarrow next$ to the old successor of p, see Figure 5. In *Mslists*, the

[3]KM does not like this.

[4]This chunk assumes that the output operator << is defined for objects of type T.

TODO

Figure 5: Insertion of a new element into a list.

code for inserting an element at the front of a list is slightly different, since there is no predecessor of the first element. We make both functions return a handle to the new element.

⟨*operations on Mslists*⟩+≡
```
handle insert_after(handle p, const T& e)
{ p->next = new node(p->next, e);
  return p->next;
}
handle add_at_front(const T& e)
{ head = new node(head, e); return head; }
```

The implementation of *insert_after* is the same for *Sslists*. In fact, the code can also used to insert a new first element into the list, since inserting an element in the beginning of the list since this is equivalent to inserting it after *hdr*. KM feels that this advantage is minor, since we probably want to give the operation its own name, anyway.

⟨*operations on Sslists*⟩+≡
```
handle insert_after(handle p, const T& e)
{ p->next = new node(p->next, e);
  return p->next;
}
handle add_at_front(const T& e)
{ return insert_after(&hdr,e); }
```

Deletions: Deleting the successor of a list element p is also easy. We simply make the successor of the successor the new successor of p. We also delete the old successor of p. Of course, we need to guard against deleting after the last element. Deletion of the first element is a special case for *Mslists*. In order to delete all elements, we repeatedly remove the first element until the list becomes empty.

⟨*operations on Mslists*⟩+≡
```
void delete_after(handle p)
{ assert( !is_last(p) );
  handle x = p->next;
  p->next = x->next;
  delete x;
}
void delete_first()
{ assert( !is_empty() );
  handle x = head;
  head = head->next;
```

```
    delete x;
  }
  void make_empty() { while (head != nil) { delete_first(); } }
```

The implementations for *Sslists* are similar, but more elegant. In *delete_after*, we assert that the successor of p is different from the header of the list. This ensures that p is not the last element of the list, and, if p points to the header, that the list is non-empty. The code for deleting the first element becomes trivial.

⟨*operations on Sslists*⟩+≡

```
  void delete_after(handle p)
  { handle x = p->next;
    assert(x != &hdr);
    p->next = x->next;
    delete x;
  }
  void delete_first() { delete_after(&hdr); }

  void make_empty() { while (hdr.next != &hdr) { delete_first(); } }
```

What happens to handles to deleted elements? They become *invalid*. It is illegal to pass an invalid handle to a list function. What does this mean? Deleting an element from a list is supposed to make the element inaccessible. Of course, deletion of a list node does not make the memory allocated for it disappear from the memory of our computer and hence we may still attempt to access the list node after its deletion from the list. The result of such an access is undefined. It may lead to a run time error or it may return an arbitrary value. We summarize this behavior in the statement that any handle to a deleted node becomes invalid.

Splice: Whereas it is maybe not so astonishing that simple operations involving a single element can be performed in constant time, a more remarkable feature of lists is that we can move around entire sublists using just four assignments. The procedure *splice* operates on two lists: *this* list and some other list L; L may be identical to *this* list. The function is given three handles: A handle *insert_after_me* into *this* list and handles *from_after_me* and *until* into L. It must be the case that *from_after_me* precedes or is equal to *until* and that, if L is equal to *this* list, *insert_after_me* is not between *next*(*from_after_me*) and *until* (both inclusive). The function removes the elements following *from_after_me* through *until* from L and inserts the entire sublist into *this* following the handle *insert_after_me*. Figure 6 illustrates the effect of splice. If *until* is identical to *from_after_me* no elements are moved.

⟨*operations with common implementations*⟩≡

```
  void splice(handle insert_after_me, handle from_after_me, handle until)
  { if (from_after_me == until) return;
    handle        x         =   insert_after_me->next;
    insert_after_me->next   = from_after_me->next;
    from_after_me->next     =     until->next;
    until->next             = x;
  }
```

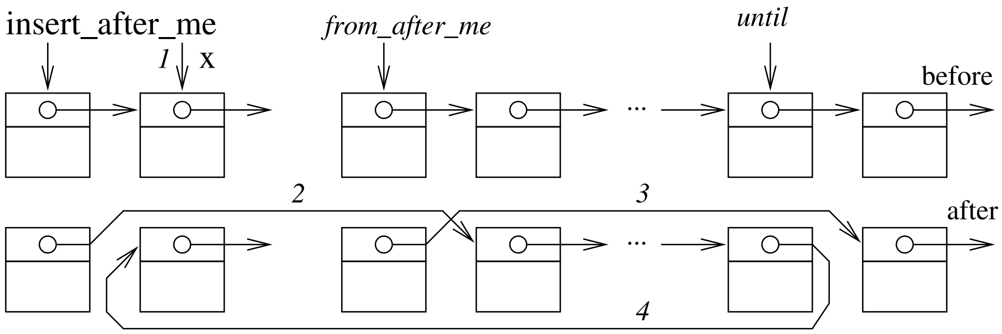

Figure 6: Splicing. The numbers indicate the order in which new pointers are established.

splice can be used to define many more specialized functions. For example, the two elements following a handle p can be swapped by calling `L.splice(p, L.next(p), L.next(L.next(p)))`.

Searching: We finish with a function searching for the first occurrence of an element e. We give two versions of the function in order to illustrate a difference between *Mslists* and *Sslists*.

The first version starts searching at a handle p and looks for the first occurrence at or after p. The second version starts searching at the position after p and looks for the first occurrence after p. It returns a handle to the predecessor of the occurrence. Having a predecessor to the occurrence makes it easier to operate on the occurrence later-on, e.g., to delete it.

In *Mslists*, an unsuccessful search returns *nil* in the first version and a handle to the last element in the second version.

⟨*operations on Mslists*⟩+≡

```
handle find_next_occurrence_starting_at(handle p, const T& e)
{ while ( true )
  { if ( p == nil || p->inf == e ) return p;
    p = p->next;
  }
}
handle find_pred_of_next_occurrence_after(handle p, const T& e)
{ while ( true )
  { handle x = p->next;
    if ( x == nil || x->inf == e ) return p;
    p = x;
  }
}
```

Observe, that the second variant of our function cannot be used to search an entire list, since we do not have a handle for the predecessor of the first element. This is a major inconvenience in the use of *Mslists*.

We come to *Sslists*. The design of *Sslists* offers two advantages when it comes to searching. First, we have a predecessor of the first element and hence can use the second variant for the entire list. Second, the last element in the list has a successor and this can be exploited to speed up the search.

We use *hdr* as a *sentinel* guarding the data structure so that the search cannot run out of its specified range. Sentinels are a quite general technique and can be applied in many circumstances. More specifically, we initialize the search by storing *e*, the element we are searching for, in the *inf*-field of *hdr*. This makes sure that there is an element after *p*, whose *inf*-field is equal to *e* and, in this way, turns every search into a successful one. The design obviates the need of testing for the end of the list in each iteration.

⟨*operations on Sslists*⟩+≡

```
handle find_next_occurrence_starting_at(handle p, const T& e)
{ hdr.inf = e; // build sentinel
  while ( true )
  { if ( p->inf == e ) return p;
    p = p->next;
  }
}
handle find_pred_of_next_occurrence_after(handle p, const T& e)
{ hdr.inf = e; // build sentinel
  while ( true )
  { handle x = p->next;
    if ( x->inf == e ) return p;
    p = x;
  }
}
```

This concludes the definitions of our two list data implementations. Sections 0.1.3 and 0.1.4 summarize the definitions.

0.1.2 A Short Program Using Lists

We give a short program that illustrates our list implementations.

⟨*list_test.c*⟩≡

```
#include "Sslist.h"
#include "Mslist.h"
#include <iostream.h>
#include <LEDA/misc.h>
template <class T>
void print(const Sslist<T>& L)
{
  cout << "< ";
  for (Sslist<T>::handle p = L.header(); !L.is_last(p);  p = L.next(p)) {
    cout << L[L.next(p)] << " ";
  }
  cout << ">" << endl;
}
template <class T>
void print(const Mslist<T>& L)
{
```

```
  cout << "<";
  for (Mslist<T>::handle p  = L.first(); p != nil;  p = L.next(p)) {
    cout << L[p] << " ";
  }
  cout << ">" << endl;
}
template <class listimpl>
void experiment(listimpl& L)
{ cout << "\n\n";
  for (int i = 1;  i <= 10;  i++) {
    L.add_at_front(i);
    print(L);
  }
  L.delete_after(L.next(L.first()));
  print(L);
  L.splice(              L.first(),
                  L.next(L.first()),
           L.next(L.next(L.first())));
  print(L);
  L[L.first()] = 7;
  L.make_empty();
  int n = 4000000; // 4 Million (32 MegaBytes)
  float T = used_time();
  for (int i = 0; i < n; i++) L.add_at_front(0);
  cout << "construction took " << used_time(T) << " seconds.\n";
  L.find_next_occurrence_starting_at(L.first(),n);
  cout << "search took       " << used_time(T) << " seconds.\n";
}
int main()
{ { cout << "Sslists:"; Sslist<int> SL; experiment(SL); }
  { cout << "Mslists:"; Mslist<int> ML; experiment(ML); }
}
```

The program produced.

```
Sslists:
< 1 >
< 2 1 >
< 3 2 1 >
< 4 3 2 1 >
< 5 4 3 2 1 >
< 6 5 4 3 2 1 >
< 7 6 5 4 3 2 1 >
< 8 7 6 5 4 3 2 1 >
< 9 8 7 6 5 4 3 2 1 >
< 10 9 8 7 6 5 4 3 2 1 >
< 10 9 7 6 5 4 3 2 1 >
< 10 7 9 6 5 4 3 2 1 >
construction took 4.3 seconds.
search took       0.45 seconds.
Mslists:
```

```
<1 >
<2 1 >
<3 2 1 >
<4 3 2 1 >
<5 4 3 2 1 >
<6 5 4 3 2 1 >
<7 6 5 4 3 2 1 >
<8 7 6 5 4 3 2 1 >
<9 8 7 6 5 4 3 2 1 >
<10 9 8 7 6 5 4 3 2 1 >
<10 9 7 6 5 4 3 2 1 >
<10 7 9 6 5 4 3 2 1 >
construction took 3.65 seconds.
search took       0.42 seconds.
```

0.1.3 Mslists: Summary

1. Definition

An instance L of the parameterized data type *Mslist*<T> represents a sequence of elements of type T. Every list element knows its successor element. Element access and navigation in the sequence is done via handles. A handle is the abstraction of a pointer to an element. Deletions invalidate handles. Invalid handles may not be passed to any function of *Sslist*.

2. Types

Mslist<T>::*handle*	the handle type.

3. Creation

Mslist<T> L;	creates an instance L of type *Mslist*<T> representing an empty list. This is the only constructor for *Mslist*s. In particular, there is no copy constructor.

4. Operations

bool	L.is_empty( )	returns true if L is empty.
handle	L.first( )	returns a handle to the first element (*nil* if the list is empty)
bool	L.is_last(*handle* p)	returns whether p is the last element of the list.
handle	L.next(*handle* p)	returns a handle for the element following p (*nil* is p points to the last element of the list).
T&	L[*handle* p]	returns a reference to the data element pointed to by p.
handle	L.insert_after(*handle* p, T e)	inserts e after p and returns a handle to the new element.
handle	L.add_at_front(T e)	inserts e at the front of L and returns a handle to the new element.
void	L.delete_after(*handle* p)	deletes the element following element p and makes any handle to it invalid. *Precondition*: p is not the last element.
void	L.delete_first( )	deletes the first element of L and invalidates any handle to it. *Precondition*: L is not empty
void	L.make_empty( )	deletes all elements of L.
handle	L.find_next_occurrence_starting_at(*handle* p, T e)	returns and handle to the first element e' after or at p such that $e' = e$. If no such item exists, *nil* is returned.
handle	L.find_pred_of_next_occurrence_after(*handle* p, T e)	returns the predecessor of the first element e' following p such that $e' = e$. If no such item exists, the handle of the last element of L is returned.
void	L.splice(*handle insert_after_me*, *handle from_after_me*, *handle until*)	deletes the sublist following handle *from_after_me* and ending with handle *until* from the list there are currently stored in and inserts it after handle *insert_after_me*. *Precondition*: *from_after_me* precedes or is equal to *until* and *insert_after_me* is not between *next*(*from_after_me*) and *until*.

5. Implementation

Operations *make_empty* and destruction take linear time. *find_pred_of_next_occurrence* takes time proportional to the number of elements between p and the returned result. All other operations take constant time.

0.1.4 Sslists: Summary

1. Definition

An instance L of the parameterized data type *Sslist*<T> is a sequence of list nodes each holding an element of type T. Every list contains a special node, its *header*. Every list element knows its successor element. The header precedes the first element and is the successor of the last element in the list. Element access and navigation in the sequence is done via handles. A handle is the abstraction of a pointer to an element. Deletions invalidate handles. Invalid handles may not be passed to any function of *Sslist*.

2. Types

Sslist<T>::*handle*	the handle type.

3. Creation

Sslist<T> L;	creates an instance L of type *Sslist*<T> representing an empty list. This is the only constructor for *Sslist*s. In particular, there is no copy constructor.

4. Operations

bool	L.is_empty( )	returns true if L is empty.
handle	L.header( )	returns a handle to the *header*.
bool	L.is_header(*handle* p)	returns true if p points to the *header*.
handle	L.first( )	returns a handle to the first element (an handle to the header if the list is empty or should it return nil??).
bool	L.is_last(*handle* p)	returns whether p is the last element of the list (the *header* is the last element of the empty list).
handle	L.next(*handle* p)	returns a handle for the element following p if it exists. If *is_last*(p), a handle to *header* is returned.
$T\&$	L[*handle* p]	returns a reference to the data element pointed to by p. *Precondition*: $p \neq L.header(\,)$ (should we test it??)
handle	L.insert_after(*handle* p, T e)	inserts e after p and returns a handle to the new element.
handle	L.add_at_front(T e)	inserts e at the front of L and returns a handle to the new element..
void	L.delete_after(*handle* p)	deletes the element following the element handled by p; any handle to the deleted element becomes invalid. *Precondition*: p is not the last element and L is non-empty
void	L.delete_first( )	deletes the first element of L and invalidates any handle to it. *Precondition*: L is not empty
void	L.make_empty( )	deletes all elements of L. All handles except $L.head(\,)$ become invalid.
handle	L.find_next_occurrence_starting_at(*handle* p, T e)	returns a handle to the first element e' after or at p such that $e' = e$. If no such item exists, a handle to *header* is returned.
handle	L.find_pred_of_next_occurrence_after(*handle* p, T e)	returns the predecessor of the first element e' following p such that $e' = e$. If no such node exists, the handle of the last element of L is returned (*header* is returned if the list is empty).
handle	L.succ(*handle* p)	returns a handle to the element after p and returns *nil* if p is the last element (also returns *nil* if p is the header and the list is empty)
handle	L.cyclic_succ(*handle* p)	returns a handle to the cyclic successor p (the cyclic successor of the last element is the first element). *Precondition*: $p \neq L.header(\,)$.

void *L*.splice(*handle insert_after_me*, *handle from_after_me*, *handle until*)

deletes the sublist following handle *from_after_me* and ending with handle *until* from the list there are currently stored in and inserts it after handle *insert_after_me*.
Precondition: *from_after_me* precedes or is equal to *until* and *insert_after_me* is not between *next*(*from_after_me*) and *until*.

5. Implementation

Operations *make_empty* and destruction take linear time. *find_pred_of_next_occurrence* takes time proportional to the number of elements between p and the returned result. All other operations take constant time.

0.1.5 Discussion and Conclusion

At this point we have two implementations of lists. Which one is better?

Sslists have the advantage that the existence of the special node *hdr* makes many programs more elegant. The fact that the first element of the list has a predecessor allows to treat the position before the first element in the same way as the position after a list element. In *Mslists*, the position before the first element has to be treated differently. This makes the use of *Mslists* more difficult. Consider, for example, the task of removing all occurrences of an element e from a list L. In *Sslists* we can write

```
// Sslists
handle p = L.header();
while ( true )
{ handle x = L.find_pred_of_next_occurrence_after(p,e);
  if ( L.is_last(x) ) break;
  L.delete_after(x);
  p = x;
}
```

or even shorter (but not necessarily clearer)

```
handle p = L.header();
while ( !L.is_last(p = L.find_pred_of_next_occurrence_after(p,e)) )
{ L.delete_after(p); }
```

Observe that the code even works if the list is empty, since *header* is the last element of an empty list. KM had not thought of the empty list, when he wrote this code. He was pleasantly surprised to learn that the code also works for an empty list, a definite plus for *Sslists*. The power of definitions which lead to correct results even when used carelessly cannot be overestimated.

In *Mslists* we have to work much harder, because we need to treat the case of an empty list and the case that the first element is equal to e separately.

```
// Mslists
handle p = L.first();
while ( p != nil && L[p] == e )
{ L.delete_first();
  p = L.first();
}
if ( p != nil )
{ while ( true )
  { handle x = L.find_pred_of_next_occurrence_after(p,e);
    if ( L.is_last(x) ) break;
    L.delete_after(x);
```

```
      p = x;
    }
  }
```

KM concedes defeat on this one.

Another advantage of *Sslists* is the use of *hdr* as a sentinel in searches. It simplifies code and saves one instruction in the inner loop of the search. The gain in speed is minor. On an ULTRA-SPARC an unsuccessful search in a list with 4 million elements took 0.41 seconds with both implementations. This is surprising. Shouldn't *Sslists* be faster. An explanation is as follows. The running time of the search is dominated by memory access time. The additional instruction of comparing a pointer to *nil* is negligible compared to the cost of bringing a list node into cache-memory. KM concedes that the code is simpler in the case of *Sslists* and that there is a potential for greater speed.

KM's main arguments for *Mslists* are that he was taught *Mslists* and that he feels that they are safer. The fact that the loop

```
p = L.first();
while ( true ) { p = L.next(p); }
```

does not lead to a runtime error in the case of *Sslists* disturbs him a lot. Similarly, the fact that `L[L.first()]` does not crash when L is the empty list.

Can we have the best of both worlds? KM is almost convinced that *Sslists* are the better design. If there were only a way around the problems mentioned in the previous paragraph.

0.1.6 Doubly Linked Lists

In a doubly linked list every node contains two pointers, one to its predecessor and one to its successor. The details are left for the exercises.

0.1.7 Memory Management

Our implementations of lists rely on the built-in operators `new` and `delete` for the constructions and destruction of list nodes in insertions and deletions. This is wasteful, except if the elements of type T are very large, since every invocation of `new` and `delete` involves a system call (= a call to the operationg system). We are wasting space because the memory manager of the operating system associates bookkeeping information with every piece of memory handed out to the application. We are wasting time because system calls are much more time consuming than simple operations like *splice*. This can be seen from the output of our example program in Section 0.1.2.

```
construction took 4.3 seconds.
search took       0.45 seconds.
```

The construction of the list took about 10 times as long as the search, although the number of instructions executed differs by much less. LEDA's list implementations do not rely on the default implementations of the operators `new` and `delete`. The same test with LEDA's singly linked lists

⟨*LEDA_list_test*⟩≡

```
int n = 4000000; // 4 Million (32 MegaBytes)
slist<int> L;

float T = used_time();
for (int i = 0; i < n; i++) L.push(0);
cout << "construction took " << used_time(T) << " seconds.\n";

slist_item it = L.first();
while (it != nil && L[it] != n ) it = L.succ(it);
cout << "search took       " << used_time(T) << " seconds.\n";
```

gives the following times:

```
construction took 0.71 seconds.
search took       0.23 seconds.
```

We see that construction can be improved considerably[5].

We outline next how memory management can be done more efficiently. The main idea is to allocate list nodes in large chunks in order to avoid the large overhead of frequent system calls.

For each list class, we keep a global pool P of unused nodes. Insertions take nodes from the pool and deletions return nodes to the pool. The destructor for lists returns the non-header nodes of the list by a single splice operation and then returns the header. In this way construction takes constant time. If the pool runs out of nodes it makes a system call to allocate a large chunk of new memory, say eight kilobytes at a time. This is the virtual memory page size used by many current operating systems.[6]

C++ offers the possibility to redefine the `new` and `delete` operators. For example, we could redefine `new` and `delete` such that `new` takes elements from the pool and `delete` returns elements to the pool. In LEDA this is done by inserting the macro LEDA_MEMORY(class name) at the end of a class definition. The STL offers a similar scheme.

Exercises:

1. Implement operations `front`, `push_front` and `pop_front` which return the first element, insert a new elements at the beginning of the list and remove the first element respectively.

2. Prove (or disprove) that *Sslist* and *Mslist* have the following class invariants[7]:

    - Every node is the target of exactly one *next*-Pointer.
    - Every node lies in a cycle containing exactly one *header*-node.

    What would happen to these invariants if we would lift the preconditions of *splice*?

[5] I do not understand why search is also faster.

[6] Actually we should reserve slighly less than eigth kilobytes, because the operating system requires some additional space for bookkeeping information.

[7] A *class invariant* is a predicate that is true before and after any call to a member function of the class.

3. Explain why we cannot simply allow the compiler generated defaults for the copy constructor and the assignment operator to be used.

4. Implement copy constructor and assignment operator with the *deep copy* semantics, i.e., all elements of the list are copied.

5. How can `Sslists` implement iterators in the sense of the C++ standard library.

6. Add some syntactic sugar to *Sslists* by defining non-member functions $Sslist{<}T{>}::handle\ next(Sslist{<}T{>}::handle\ p)$ and $T\&\ inf(Sslist{<}T{>}::handle\ p)$.

7. Write a procedure *reverse* that given an *Sslist* $\langle i_1, i_2, \ldots, i_n \rangle$ tranforms it into the reverse list $\langle i_n, \ldots, i_2, i_1 \rangle$. It should run in time $\mathcal{O}(n)$ using only $\mathcal{O}(1)$ additional memory.

8. Design a extension of *Sslists* with an additional member *last* that points to the last node. Go through all member functions and make the required changes. Use this additional information to implement the following additional functionality using only constant execution time:

   - *back* (*tail*) returns the last element.
   - *pushBack* (*append*) inserts an node at the end of the list.
   - *concat* concatenates two lists.
   - *insert_list_after* inserts one list into another list after a specified node.

9. Design a subclass of *Sslist* with an additional member *size* that keeps track of the number of elements stored. The goal is to support a function *size* working in constant time. Can you achieve this goal and still have the function *splice*? Can you find a solution which implements *size* and runs in constant time as long as *splice* is not used by an application. The function *splice* is still supported and *size* becomes more expensive, if *splice* is also used. The solution requires a small change in the interface of *splice*.

10. Design a doubly linked list data structure based on nodes that store both predecessor pointers and successor pointers. Implement a function *delete* that deletes any specified node. Choose a representation where no special case treatments with respect to the ends of the list are needed. Additionally, there should be no need for a separate member *last*.