Data Structures

(Feodor F. Dragan)
Department of Mathematics and Computer Science
Kent State University

Fall, 2000

CHAPTER 17
Sequences and Linked Lists: Implementations

Contents

We will learn:

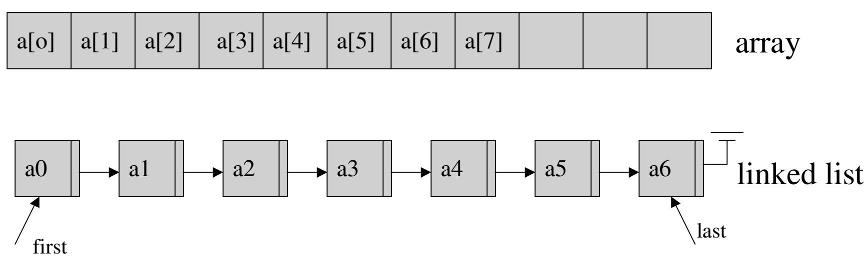
- How to allow access to any item by using a general linked list;
- The general algorithms for the linked list operations;
- How the iterator class provides a safe mechanism for traversing and accessing linked lists;
- List variations, such as doubly linked lists and circularly linked lists;
- How to use inheritance to derive a sorted linked list class; and
- How to implement the STL *list* class

Sequences and Linked Lists

- *Sequences* vs. *stacks* and *queues* (allow insertions and deletions in the middle of the container)
- Access is performed by *iterator* class
- Representation

Sequences → vectors, arrays

Sequences → linked lists

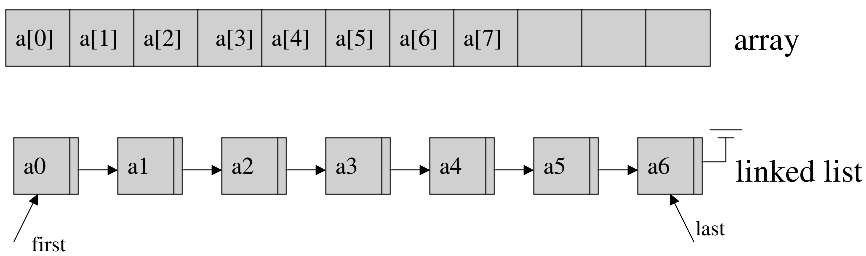


Operations on sequences include

• ***void push_back(const Object & element)***	appends an element at the end
• ***void push_front(const Object & element)***	prepends an element to the front
• ***Object & front( )***	returns the first element
• ***Object & back( )***	returns the last element
• ***void pop_front( )***	removes the first element
• ***void pop_back( )***	removes the last element
• ***iterator insert(iterator pos, const Object & obj)***	inserts ***obj*** after the element in the position referred to by ***pos***, returns the position of the newly inserted element (time for list vs vector)
• ***iterator erase(iterator pos)***	removes the object at the position referred to by ***pos***, returns the position of the element that followed ***pos*** prior to the call to ***erase*** (time for list vs vector)

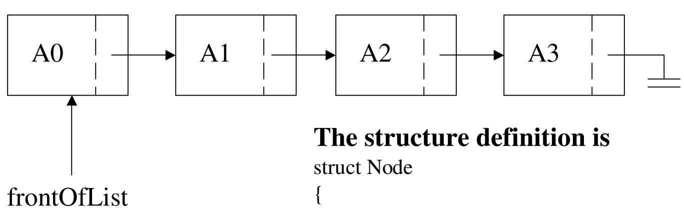
Linked lists vs. vectors

- Random access iterators are available only for vectors
- Insertion and removal from the middle of a container is efficiently supported for lists only
- Linked lists are used to avoid large amount of data movement



We choose representation by linked list.

Sequences/Linked Lists: Basic Ideas



The structure definition is

```
struct Node
{
    Object element;   //Some element
    Node *next;
};
```

```
Node *frontOfList;
```

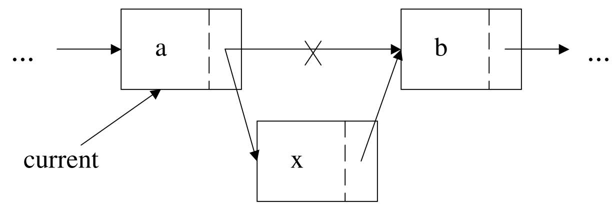
We will use name *linked list* as a synonym for the data structure *sequence.*

- Linked lists allow general access (arbitrary insertion, deletion, and find operations) through the list.

Insertion

- Insertion consists of splicing a node into the list and can be accomplished with one statement.

current->next = new Node(x, current->next);

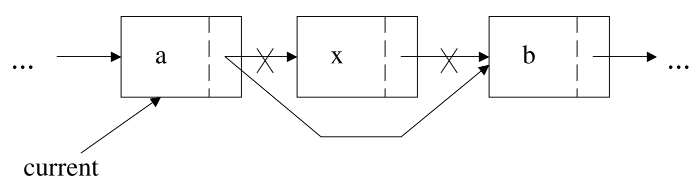


Insertion in a linked list:

- Create new node (tmp),
- Copy in x,
- Set tmp’s next pointer, and
- Set current’s next pointer.

Deletion

- Removal can be accomplished by bypassing the node. We need a pointer to the node prior to the one we want to remove.

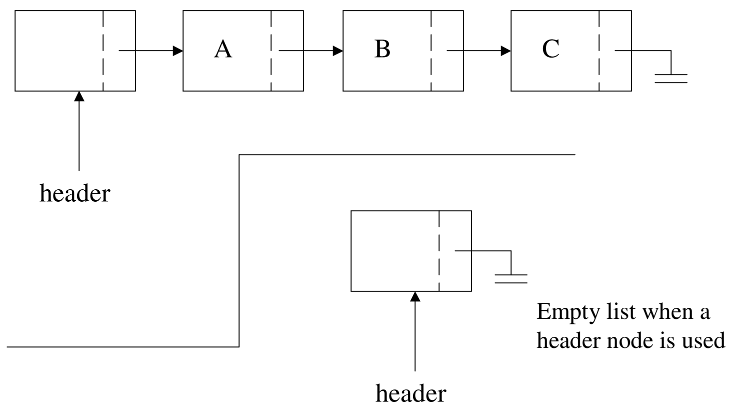



```
Node *deletedNode = current->next;    // Save pointer
current->next = current->next->next ;  // Bypass the node
delete deletedNode;                    // Free the memory
```

Linked list operations use only a constant number of data movements.

Header Nodes

- A header node holds no data but serves to satisfy the requirement that every node have previous node.
- A header node allows us to avoid a special cases such as insertion of a new first element and removal of the first element.



C++ Implementation

- A list is implemented as three separate classes:
  1. The list itself (*LList* class);
  2. The node (*LListNode* class);
  3. The position (*LListItr* class).

Type declaration of the linked list node *LListNode*

- consists of two data members: the stored element and the link to the next node
- the only methods are the constructors
- A friend of a class is granted access to the class's private section.
- An incomplete class declaration is used to inform the compiler of the existence of another class.

```
template <class Object>
class LList;     // Incomplete declaration.

template <class Object>
class LListItr;  // Incomplete declaration.

template <class Object>
class LListNode
{
  LListNode( const Object & theElement = Object( ),
           LListNode * n = NULL )
    : element( theElement ), next( n ) { }

  Object     element;
  LListNode *next;

  friend class LList<Object>;
  friend class LListItr<Object>;
};
```

Iterator class *LListItr*

```
// LListItr class; maintains "current position".
//
// CONSTRUCTION: With no parameters. The LList class may
//     construct a LListItr with a pointer to a LListNode.
// ******************PUBLIC OPERATIONS*********************
// bool isPastEnd( )      --> True if past end position in list
// void advance( )        --> Advance (if not already NULL)
// Object retrieve( )     --> Return item in current position
// ******************ERRORS********************************
// Throws BadIterator for illegal retrieve.

template <class Object>
class LListItr
{public:
   LListItr( ) : current( NULL ) { }

   bool isPastEnd( ) const
     { return current == NULL; }

   void advance( )
     { if( !isPastEnd( ) ) current = current->next; }

   const Object & retrieve( ) const
     { if( isPastEnd( ) ) throw BadIterator( );
       return current->element; }
```

```
 private:
   LListNode<Object> *current;
   // Current position

   LListItr( LListNode<Object> *theNode )
     : current( theNode ) { }

   friend class LList<Object>;
   // Grant access to constructor
};
```


The *LList* class interface

```
// LList class.
//
// CONSTRUCTION: with no initializer.
// Access is via LListItr class.
// ******************PUBLIC OPERATIONS********
// bool isEmpty( )        --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// LListItr zeroth( )     --> Return position to prior to first
// LListItr first( )      --> Return first position
// void insert( x, p )    --> Insert x after current iterator position p
// void remove( x )       --> Remove x
// LListItr find( x )     --> Return position that views x
// LListItr findPrevious( x )   --> Return position prior to x
// ******************ERRORS********************
// No special errors.

template <class Object>
class LList
{public:
   LList( );
   LList( const LList & rhs );
   ~LList( );

   bool isEmpty( ) const;
   void makeEmpty( );
   LListItr<Object> zeroth( ) const;
```

```
   LListItr<Object> first( ) const;
   void insert( const Object & x, const LListItr<Object> & p );
   LListItr<Object> find( const Object & x ) const;
   LListItr<Object> findPrevious( const Object & x ) const;
   void remove( const Object & x );

   const LList & operator=( const LList & rhs );

  private:
   LListNode<Object> *header;
};
#include "LinkedList.cpp"
```

Member function implementations (LinkedList.cpp)

```
#include " LinkedList.h“

// Construct the list.
template <class Object>
LList<Object>::LList( )
{ header = new LListNode<Object>; }

// Destructor.
template <class Object>
LList<Object>::~LList( )
{  makeEmpty( );
   delete header;   }

// Test if the list is logically empty.
// Return true if empty, false, otherwise.
template <class Object>
bool LList<Object>::isEmpty( ) const
{  return header->next == NULL; }

// Make the list logically empty.
template <class Object>
void LList<Object>::makeEmpty( )
{ while( !isEmpty( ) )
      remove( first( ).retrieve( ) );    }

// Return an iterator representing the header node.
template <class Object>
LListItr<Object> LList<Object>::zeroth( ) const
{ return LListItr<Object>( header ); }

// Return an iterator representing the first node in the list.
// This operation is valid for empty lists.
template <class Object>
LListItr<Object> LList<Object>::first( ) const
{ return LListItr<Object>( header->next );  }

// Return iterator corresponding to the first node
// containing an item x.       (takes O(N) time)
// Iterator isPastEnd if item is not found.
template <class Object>
LListItr<Object> LList<Object>::find( const Object & x ) const
{  LListNode<Object> *p = header->next;
   while( p != NULL && p->element != x )
      p = p->next;
   return LListItr<Object>( p );
}
```


Member function implementations (cont.)

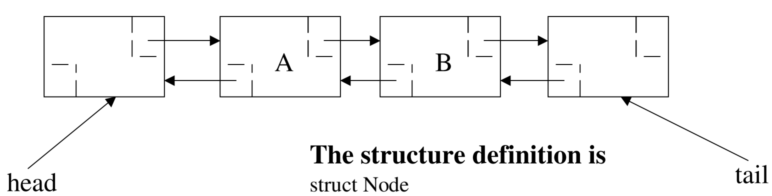
```
// Insert item x after p. (takes constant time)
template <class Object>
void LList<Object>::insert( const Object & x, const LListItr<Object> & p )
{  if( p.current != NULL )
      p.current->next = new LListNode<Object>( x, p.current->next );  }

// Return iterator prior to the first node containing an item x. (takes O(N) time)
template <class Object>
LListItr<Object> LList<Object>::findPrevious( const Object & x ) const
{  LListNode<Object> *p = header;
   while( p->next != NULL && p->next->element != x )
      p = p->next;
   return LListItr<Object>( p );  }

// Remove the first occurrence of an item x.
template <class Object>
void LList<Object>::remove( const Object & x )
{  LListNode<Object> *p = findPrevious( x ).current;
   if( p->next != NULL )
   {
      LListNode<Object> *oldNode = p->next;
      p->next = p->next->next;  // Bypass deleted node
      delete oldNode;
   }
}
```

Test program

```
#include "LinkedList.h"
#include <iostream>
using namespace std;

// Simple print function.
template <class Object>
void printList( const LList<Object> & theList )
{  if( theList.isEmpty( ) )
        cout << "Empty list" << endl;
    else
    {  LListItr<Object> itr = theList.first( );
        for( ; !itr.isPastEnd( ); itr.advance( ) )
            cout << itr.retrieve( ) << " ";     }
    cout << endl;
}

int main( )
{  LList<int>    theList;
    LListItr<int> theItr = theList.zeroth( );
    int i;
    printList( theList );

for( i = 0; i < 10; i++ )
    {  theList.insert( i, theItr );
        printList( theList );
        theItr.advance( ); }

    for( i = 0; i < 10; i += 2 )
        theList.remove( i );

    for( i = 0; i < 10; i++ )
        if( ( i % 2 == 0 ) != ( theList.find( i ).isPastEnd( ) ) )
            cout << "Find fails!" << endl;

    cout << "Finished deletions" << endl;
    printList( theList );

    cout << "A copy" << endl;
    LList<int> list2;  list2 = theList;  printList( list2 );

    list2 = LList<int>( );
    cout << "The original and an empty list" << endl;
    printList( theList ); printList( list2 );

    cout << "Sorted list3" << endl;
    SortedLList<int> list3;
    list3.insert( 3 );   list3.insert( 1 );  list3.insert( 7 );
    list3.insert( 5 );   list3.insert( 9 );  printList( list3 );

    return 0;
}
```


Doubly Linked Lists

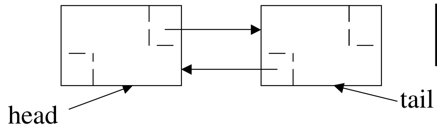
• A *doubly linked list* allows bidirectional traversal by storing two pointers per node.



The structure definition is

```
Node *head;
Node *tail;

struct Node
{
      Object element;    //Some element
      Node *next;
      Node *prev;
};
```

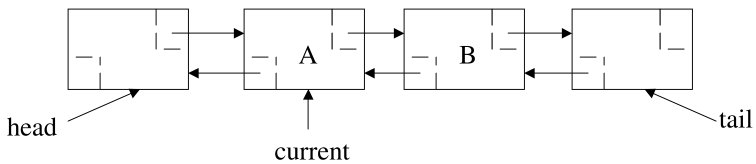
• Symmetry demands that we use both a head and a tail and that we support roughly twice as many operations.



An empty doubly linked list

- *head→prev* and *tail→next* are not needed and are not even initialized.
- The test for emptiness is now *head→next == tail (or tail→prev == head)*

Advance and Retreat

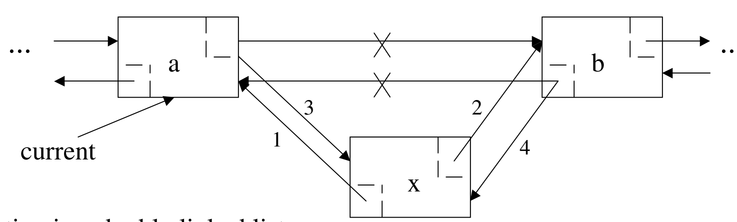


- The **advance** operation is implemented by *current=current→next*
- The **retreat** operation is implemented by *current=current→prev*
- When we advance past the end of the list, we now hit the tail node instead of NULL

- Insertion and removal involve twice as many pointer changes as for a singly linked list.

Insertion

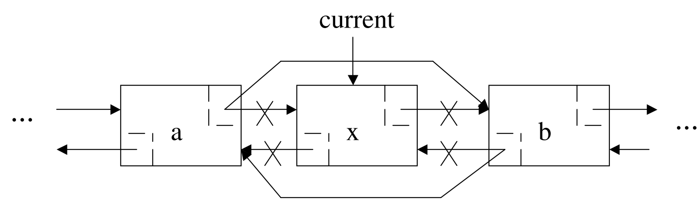
- We can consider both *insertBefore* and *insertAfter*



Insertion in a doubly linked list:

- newNode = new Node(x);
- newNode→ prev = current; // Set x's prev pointer
- newNode→ next = current→ next; // Set x's next pointer
- newNode→ prev→ next = newNode; // Set a's next pointer
- newNode→ next→ prev = newNode; // Set b's prev pointer
- current = newNode;

Deletion



• The remove operation can proceed from the current node because we can obtain the previous node instantly.

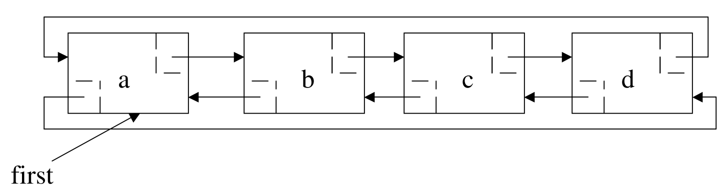
Deletion in a doubly linked list:

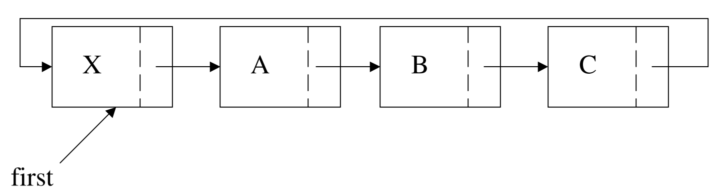
- oldNode = current;
- oldNode→ prev→ next = oldNode→ next; // Set a's next pointer
- oldNode→ next → prev = oldNode→ prev; // Set b's prev pointer
- delete oldNode;
- current = head; // So current is not stale

• A doubly linked list is used to implement the STL *list* class.

Circularly Linked Lists

• In a *circularly linked list,* the last cell's *next* pointer points to *first*. This action is useful when wraparound matters (head and tail are not needed).





Sorted Linked Lists

- We can maintain items in sorted order by deriving a *SortedLList* class from *LList.*
- The insert method should be declared virtual in the base class.

```
#include "LinkedList.h"
// SortedList class
// CONSTRUCTION: with no initializer.  Access is via LListItr class
//// ******************PUBLIC OPERATIONS******
// bool isEmpty( )       --> Return true if empty; else false
// void makeEmpty( )     --> Remove all items
// LListItr zeroth( )    --> Return position to prior to first
// LListItr first( )     --> Return first position
// void insert( x )      --> Insert x in sorted order
// void insert( x, p )   --> Insert x in sorted order; ignore p
// void remove( x )      --> Remove x
// LListItr find( x )    --> Return position that views x
// LListItr findPrevious( x )  --> Return position prior to x
// ******************ERRORS********************
// No special errors
template <class Object>
class SortedLList : public LList<Object>
{ public:
   void insert( const Object & x );
   void insert( const Object & x, const LListItr<Object> & p )
     { insert( x ); }                          };
```


Sorted Linked Lists: Insertion

```
// Insert item x into the list.
template <class Object>
void SortedLList<Object>::insert( const Object & x )
{
   LListItr<Object> prev = zeroth( );
   LListItr<Object> curr = first( );

   while( !curr.isPastEnd( ) && curr.retrieve( ) < x )
   {
      prev.advance( );
      curr.advance( );
   }

   LList<Object>::insert( x, prev );
}
```

Homework

1. Problem 17.6 pages 599-600 of the text
2. Problem 17.9 page 600 of the text
3. Problem 17.10 page 600 of the text
4. Problem 17.17 page 601 of the text