CMSC 341

Lists - II

Doubly Linked List Implementation

Recall the List ADT

A list is a dynamic ordered tuple of homogeneous elements

$A_o, A_1, A_2, \ldots, A_{N-1}$

where A_i is the ith element of the list

Operations on a List

- create an empty list
- destroy a list
- construct a (deep) copy of a list
- find(x) – returns the position of the first occurrence of x
- remove(x) – removes x from the list if present
- insert(x, position) – inserts x into the list at the specified position
- isEmpty( ) – returns true if the list has no elements
- makeEmpty( ) – removes all elements from the list
- findKth(position) – returns the element in the specified position

The implementations of these operations in a class may have different names than the generic names above

A Linked List Implementation

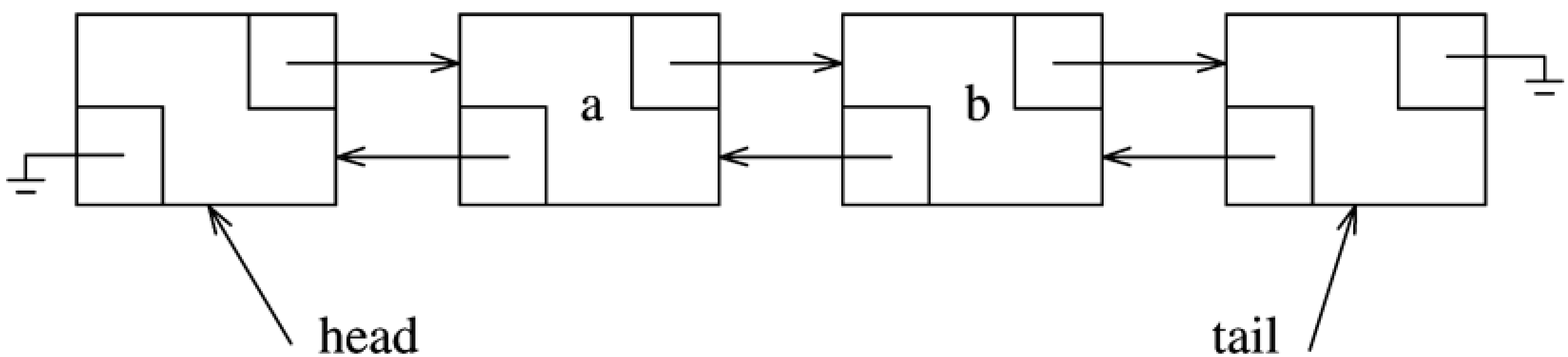
An alternative to the vector's array-based implementation of the List ADT is a linked-list implementation.

The STL provides the "list" container which is a singly linked list.

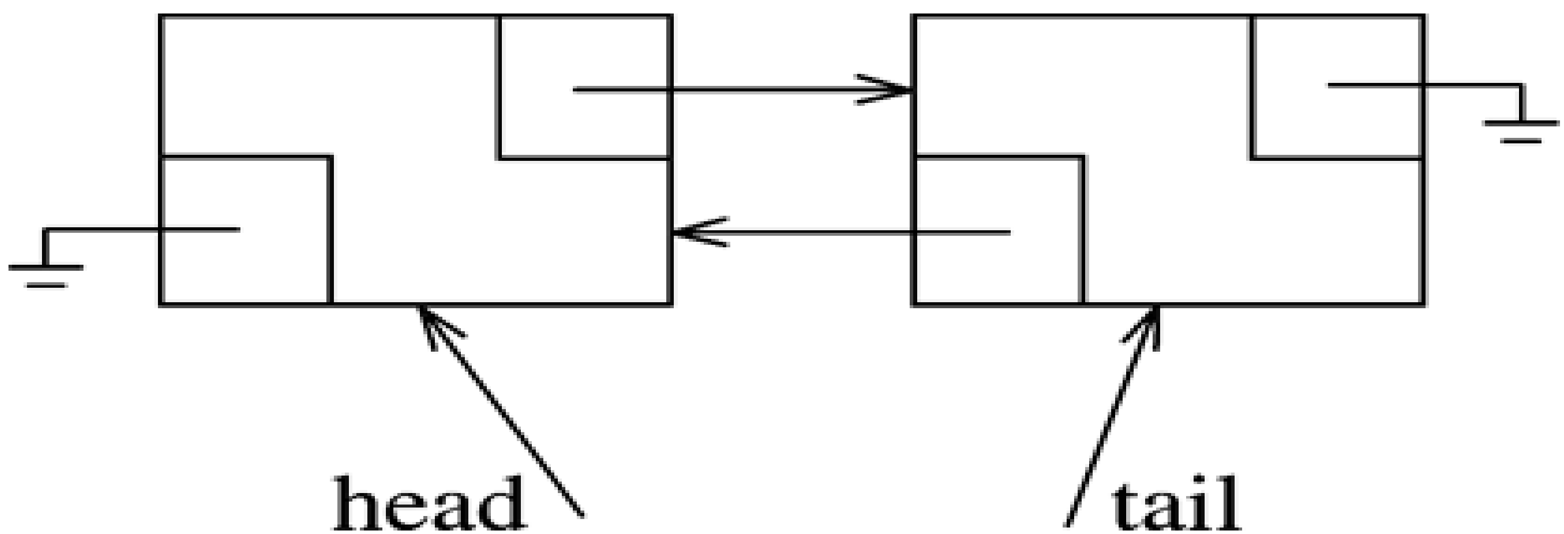
We will implement our own "List" class (note the upper-case "L") as a doubly linked list with both header and tail nodes.

As we'll see, the use of the header and tail nodes will simplify the coding by eliminating special cases.

A doubly-linked list with header and tail nodes



An empty List



List classes

To implement the doubly-linked List, four classes are required

- The List class itself which contains pointers to the header and tail nodes, all the list methods, and required supporting data
- A List Node class to hold the data and the forward and backward Node pointers
- A const_iterator class to abstract the position of an element in the List. Uses a Node pointer to the "current" node.
- An iterator class similar to the const_iterator class

- The Node and iterator classes will be nested inside the List class

The List class outline

```
template< typename Object>
class List
{
  private:
      struct Node
             { /* see following slide */ }

  public:
      class const_iterator
             { /* see following slide */ }
      class iterator : public const_iterator
             { /* see following slide */

      // A whole host of List methods

  private:
      int theSize;
      Node *head;
      Node *tail;

      // helper function(s)
};
```

The List's Node struct

The Node will be nested in the List class template and will be private, so a struct is sufficient and easier to code. What alternative ways are there to define the Node?

```
struct Node
{
  Object  data;
  Node   *prev;
  Node   *next;

  Node( const Object & d = Object( ),
        Node *p = NULL, Node *n = NULL )
          : data( d ), prev( p ), next( n )
      { /* no code */ }
};
```

const_iterator class

```
class const_iterator
{
   public:

        const_iterator( ) : current( NULL )
          { }

        const Object & operator* ( ) const
          { return retrieve( ); }

        bool operator== ( const const_iterator & rhs ) const
          { return current == rhs.current; }

        bool operator!= ( const const_iterator & rhs ) const
          { return !( *this == rhs ); }
```

const_iterator class (2)

```
// pre-increment
const_iterator & operator++ ( )
{
    current = current->next;
    return *this;
}

// post-increment
const_iterator operator++ ( int dummy)
{
    const_iterator old = *this;
    ++( *this );
    return old;
}
```

const_iterator class (3)

```
// pre-decrement
const_iterator & operator-- ( )
{
     current = current->prev;
     return *this;
}

// post-decrement
const_iterator operator-- ( int dummy)
{
     const_iterator old = *this;
     --( *this );
     return old;
}
```

const_iterator class (4)

```
protected:  // available to iterator class

    Node *current;

    Object & retrieve( ) const
      { return current->data; }

    const_iterator( Node *p ) : current( p )
      { }

    friend class List<Object>;  // why?
 };
```

iterator class

```
class iterator : public const_iterator
   {
     public:
       iterator( )
         { }

       // this is different than in const_iterator
       // because it’s not a const method
       Object & operator* ( )
         { return retrieve( ); }

       // explicitly reimplement const operator*
       // otherwise the original is hidden by operator* above
       const Object & operator* ( ) const
         { return const_iterator::operator*( ); }


       // operator== and operator!= inherited
```

iterator class (2)

```
// reimplement increment operators
// to override those in const_iterator
// because of different return types
iterator & operator++ ( )
{
    current = current->next;
    return *this;
}

iterator operator++ ( int dummy)
{
    iterator old = *this;
    ++( *this );
    return old;
}
```

iterator class (3)

```
// also reimplement decrement operators
// to override those in const_iterator
// because of different return type
iterator & operator-- ( )
{
    current = current->prev;
    return *this;
}

iterator operator-- ( int dummy)
{
    iterator old = *this;
    --( *this );
    return old;
}
```

iterator class (4)

```
  protected:
    // no data since the “current” is inherited


    iterator( Node *p ) : const_iterator( p )
    { /* no code */ }

     friend class List<Object>;  // why?
};
```

The List class

```
template< typename Object>
class List
{
   private:
        struct Node
                { /* see previous slide */ }

   public:
        class const_iterator
                { /* see previous slide */ }

        class iterator : public const_iterator
                { /* see previous slide */


   // public List methods (within class definition) follow
```

List class (2)

```
// default constructor and the Big-3
List( )
    { init( ); }
List( const List & rhs )
    {
        init( );
        *this = rhs;
    }
const List & operator= ( const List & rhs )
{
    if( this == &rhs )          // self-assignment check
        return *this;
    clear( );                   // make this List empty
    for(const_iterator itr = rhs.begin( ); itr != rhs.end( ); ++itr )
            push_back( *itr );
    return *this;
}
// destructor
~List( )
{
    clear( );           // delete all the data nodes
    delete head;        // delete the header node
    delete tail;        // delete the tail node
}
```

List Class (3)

```
// Functions that create const_iterators
const_iterator begin( ) const
{  return const_iterator( head->next ); }

const_iterator end( ) const
{  return const_iterator( tail ); }

// Functions that create iterators
iterator begin( )
{  return iterator( head->next ); }

iterator end( )
{  return iterator( tail ); }
```

List class (4)

```
// accessors and mutators for front/back of the List
Object & front( )
      { return *begin( ); }

const Object & front( ) const
      { return *begin( ); }

Object & back( )
      { return *--end( ); }

const Object & back( ) const
      { return *--end( ); }

void push_front( const Object & x )
      { insert( begin( ), x ); }

void push_back( const Object & x )
      { insert( end( ), x ); }

void pop_front( )
   { erase( begin( ) ); }

void pop_back( )
      { erase( --end( ) ); }
```

List class (5)

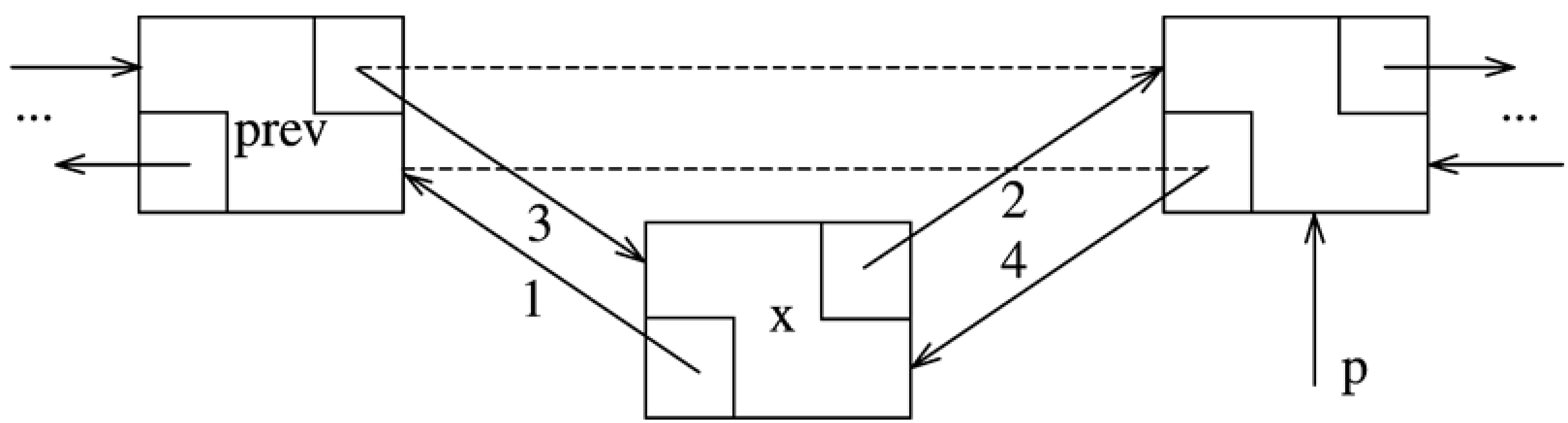
```
// how many elements in the List?
int size ( ) const
   { return theSize; }

// is the List empty?
bool empty( ) const
   { return size( ) == 0; }

// remove all the elements
void clear ( )
{
   while (! Empty( ) )
        pop_front( );
}
```

List class (6)

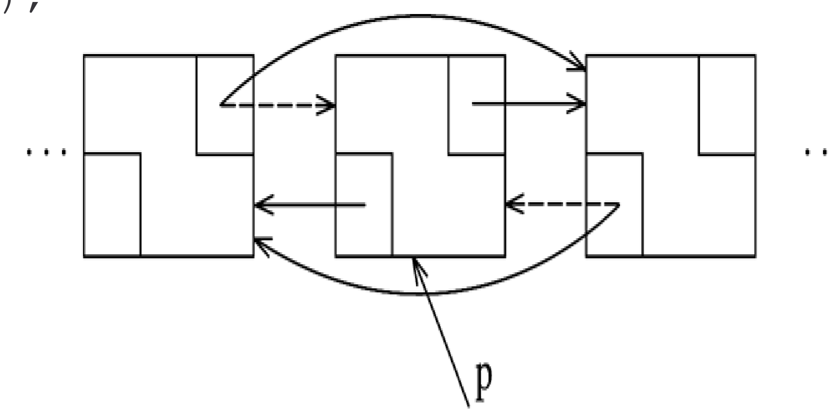
```
// Insert x before itr.
iterator insert( iterator itr, const Object & x )
{
  Node *p = itr.current;
   theSize++;
   return iterator( p->prev = p->prev->next
                  = new Node( x, p->prev, p ) );
}
```



List class (7)

```
// Erase item at itr.
 iterator erase( iterator itr )
 {
     Node *p = itr.current;
     iterator retVal( p->next );
     p->prev->next = p->next;
     p->next->prev = p->prev;
     delete p;
     theSize--;

     return retVal;
 }
```




```
  // erase items between "from" and "to"
  // including "from", but not including "to"
 iterator erase( iterator from, iterator to )
 {
     for( iterator itr = from; itr != to; )
         itr = erase( itr );

     return to;
 }
```

List class (8)

```
private:
    int   theSize;
    Node *head;
    Node *tail;

   // private helper function for constructors
   // creates an empty list
   void init( )
   {
        theSize = 0;
        head = new Node;
        tail = new Node;
        head->next = tail;
        tail->prev = head;
   }

};  // end of class definition
```

Problems with the code

What problems or inadequacies did you find in the code?

How can they be solved?

Also note that this code is written entirely within the class definition. How would the code be different if the methods were implemented outside the class definition (as some of them should be)?

Performance of List operations

What is the asymptotic performance of each List operation in terms of the number of elements in the list, N…

- When the List is implemented as a vector?
- When the List is implemented as a Doubly-Linked List?

Circular Linked List

- Use the header node’s “prev” pointer to point to the tail
- Use the tail node’s “next” pointer to point to the head

Sorted Lists

Suppose we decided that the data in the lists should be stored in sorted order.

- How would the sorted order be determined?
- What List code would need to be changed?
- How would sorting the data affect the performance of
  - Finding an element in the list
  - Inserting an element into the list
  - Removing an element from the list
  - other List functions