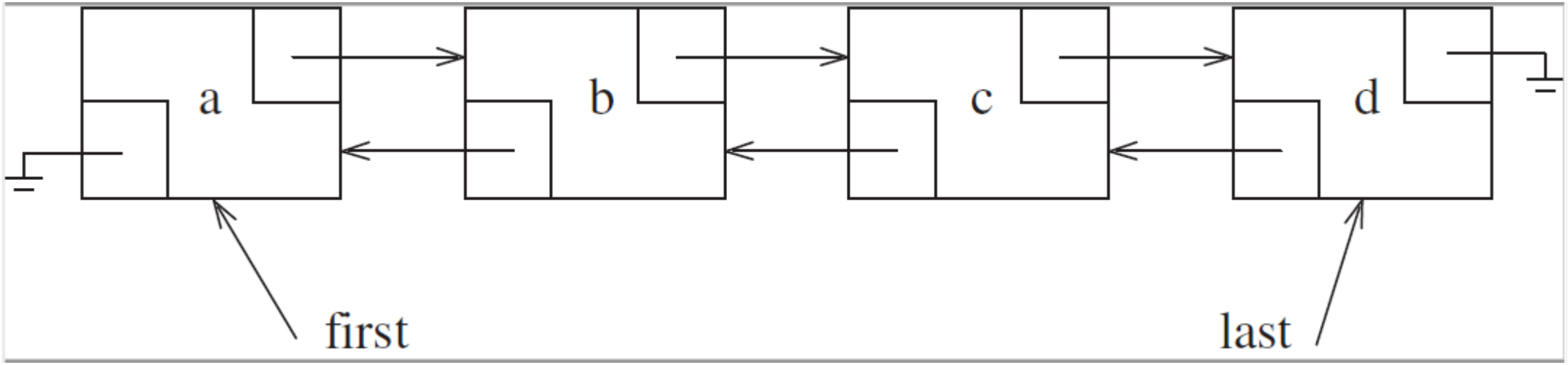
CSE 2020 Computer Science II

Module 3.1 – The ADT Class List

Instructor: Kerstin Voigt

School of CSE, CSUSB

Lists – Another Linear Data Structure



- Values stored in line
- All values of same type
- Each value within its own “Node”
- Each Node his link to “next” and “previous” Node
- List starts with an “empty” Node in front
- List ends with an “empty” Node in back
- Front of list referenced by pointer “head”
- Back of list referenced by point “tail”

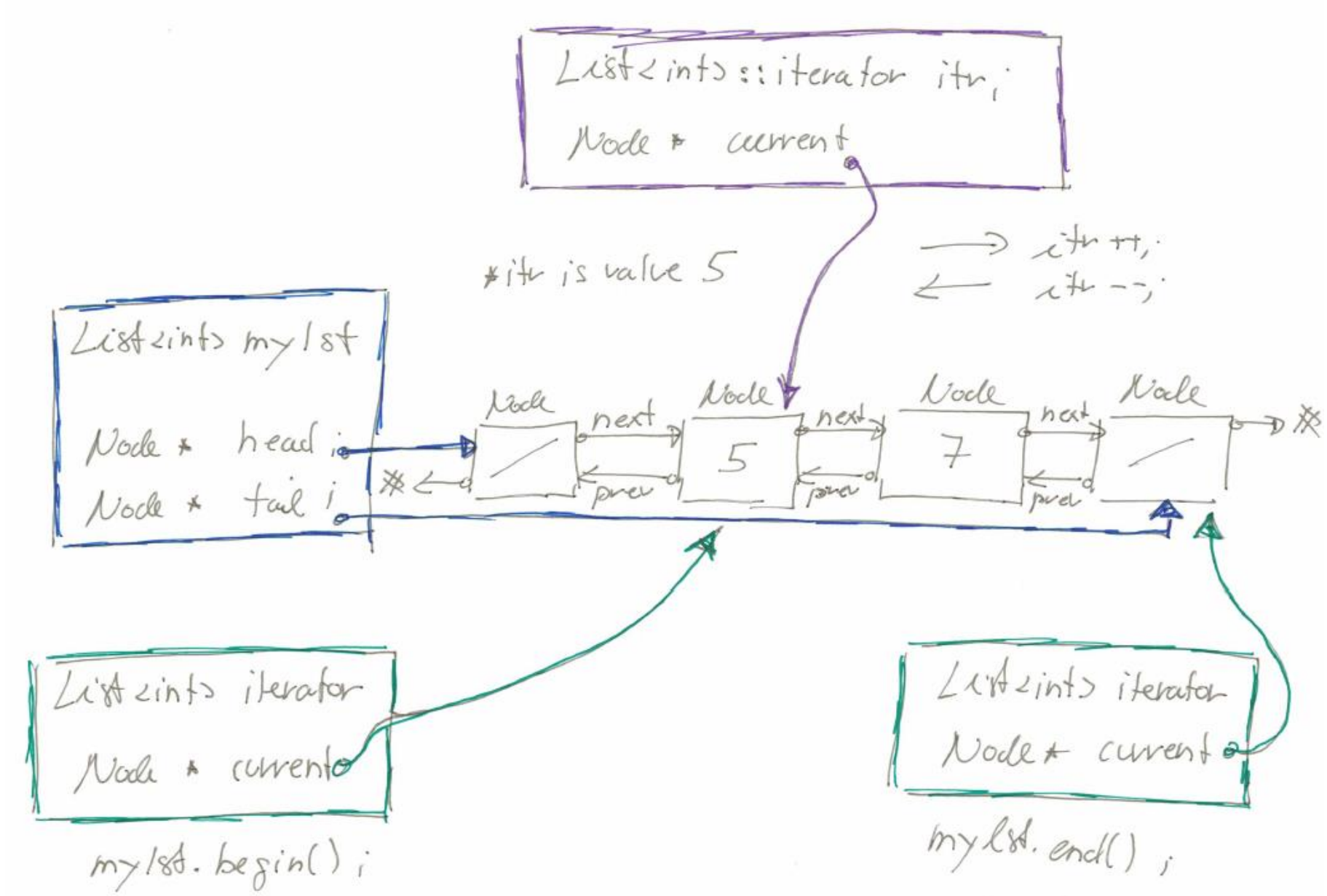
The C++ List Class

- A template class so that the stored values can be of any type
- All code in a single .h header file (splitting into .h and .cpp for template classes possible, but with compiler issues)

Be prepared to see:

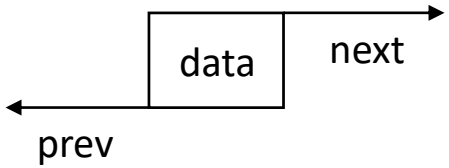
- A large class; code distributed over many slides
- Structure and class defined within a class

- Needing even more attention to detail


List<int>::iterator itr;
Node * current
*itr is value 5
--> itr++;
<-- itr--;
List<int> mylst
Node * head;
Node * tail;
Node
next
prev
5
7
List<int> iterator
Node * current
mylst.begin();
List<int> iterator
Node * current
mylst.end();

C++ Class List – Code Overview

```
template <typename T>
class List
{
  private:

    struct Node                    // a struct "just for List"
    {                              // invisible outside of class List
      T  data;
      Node   *prev;
      Node   *next;
    };

  public:
    class const_iterator       // an iterator "just for List"
    {
      public:
      …
      protected:
        Node *current;
        friend class List<T>;
    };
```



- Class List defines its own building block: struct Node
- Node bundles up vital information: type T data item, two pointers to neighboring Nodes, to left (prev) and to right (next)
- Class List defines its own iterators ("steppers")
- Notice 'class const_iterator' WITHIN class List
- Thinking of iterators as "pointers" is a good conceptual crutch until we study specifics in next lecture ….

C++ Class List – Code Overview cont.

```
class iterator : public const_iterator
{
  public:              // another iterator "just for List"
  …                    // which is subclass of const_iterator
  protected:
    …
    friend class List<T>;
};

public:                          // the public interface of class List

  List( )
    { init( ); }

  … etc.

private:                         // a List<T> is defined to have a
```

- The non-const iterator … more in next lecture
- 'friend' declaration has declaring class give permission to other class to access is private or protected data members
- …. Now the beginning of the public interface of ADT class List; details to come
- …. Moving on to the private data members of class List

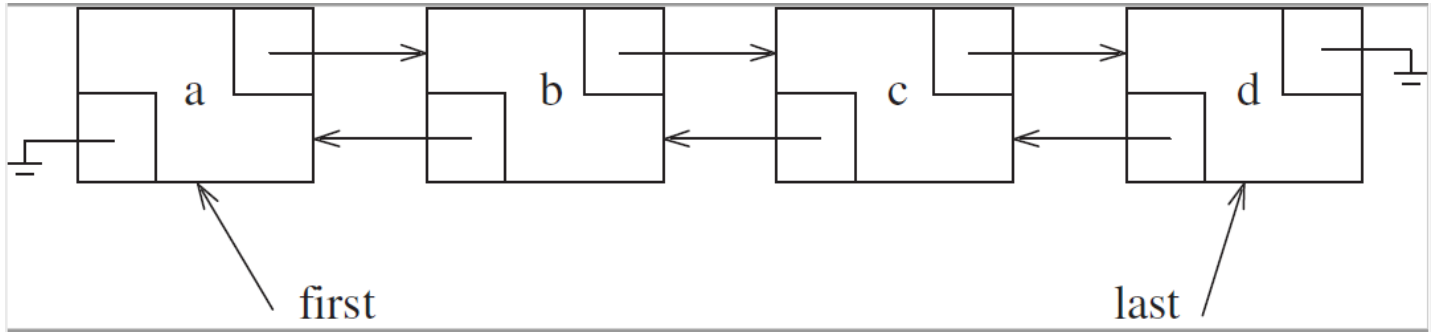
C++ Class List – Code Overview cont. cont.

```
    int   theSize;                  // size (theSize)and two pointers
    Node *head;                     // to data-containing Nodes which
    Node *tail;                     // point to succeeding or preceding
                                    // other Nodes … a "chain" of Nodes

void init( )
    {
        theSize = 0;
        head = new Node;
        tail = new Node;
        head->next = tail;
        tail->prev = head;
    }
};
```

What makes a List:

- Data member head, a pointer to a Node which marks the front end of a "linked list"
- Data member tail, a poiner to a Node which marks the back end of a "linked list"
- Note: we have define a DOUBLY-LINKED LIST
- An integer data member that holds the number of values stored in the linked list.
- A private (?!) member fct init() that initializes a List object to be "empty"

Revisiting the Linked List Diagram …



Node a: its purpose is to mark the head of the list; it is to not hold any value, its pointer to prev is NULL

Nodes b and c: regular Nodes to hold a value; linked to neighboring nodes with prev and next pointers

Node d: its purpose is to mark the tail (end) of the list; it is to not hold any value; its next pointer is NULL

C++ Class List – All the Code

```
#ifndef LIST_H
#define LIST_H

#include <algorithm>
using namespace std;

template <typename T>
class List
{
  private:
    // The basic doubly linked list node.
    // Nested inside of List, can be public
    // because the Node is itself private
    struct Node
    {
        T  data;
        Node   *prev;
        Node   *next;

        Node( const T & d = T{ }, Node * p = nullptr, Node * n = nullptr
)
          : data{ d }, prev{ p }, next{ n } { }


    };
```

- C++ struct can have constructor like classes
- Node constructor shown with
  - Default parameters
  - Initializers of form

    :<data member>(value)

- Example: Node mynode;

  A node object mynode that without specified value (data) and prev and next poiters to NULL

  An "empty" Node

```
public:

   class const_iterator
   {
     public:

       // Public constructor for const_iterator.
       const_iterator( ) : current{ nullptr }
         { }

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

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

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

Defined within class List … an iterator class which defines iterators suited to referencing list items that are const (= not mutable)

- The core of an iterator is data member

  Node* current;

- Iterator member functions set pointer current to Nodes of List and move current along the Nodes' prev and current pointers

- *this is an objects pointer itself

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

  const_iterator operator-- ( int )
  {
      const_iterator old = *this;
      --( *this );
      return old;
  }

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

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


protected:
  Node *current;
```

More of class const_iterator ….

- Notice protected data member
- Node* current;

- Move iterator one Node to right with ++ (prev slide)
- Move iterator one Node to left with –
- Access stored value under the current iterator with * (just like dereferencing a regular pointer)
- Assign iterators with =

Take in the basic idea, more lecturing on this topic coming up

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

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

    friend class List<T>;
};

class iterator : public const_iterator
{
  public:

    iterator( )
      { }

    T & operator* ( )
      { return const_iterator::retrieve( ); }

    const T & operator* ( ) const
      { return const_iterator::operator*( ); }
```

Also defined within list, class iterator as a SUBCLASS of const_iterator

- Same functionality as const_iterator

- Allows mutating referenced Nodes
- Our CSE 2020 code more likely to use these
- Examine both iterator classes carefully on your own time to spot the differences ..

```
iterator & operator++ ( )
{
    this->current = this->current->next;
    return *this;
}

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

iterator & operator-- ( )
{
    this->current = this->current->prev;
    return *this;
}

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

More iterator implementation

```
  protected:

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

    friend class List<T>;
};

public:
  List( )
    { init( ); }

  ~List( )
  {
      clear( );
      delete head;
      delete tail;
  }
```

Protected portion of class iterator

Where is the data member??

- Inherited from class const_iterator!

FINALLY: Implementing of actual CLASS LIST INTERFACE

o Default constructor (produces empty List object)

o Destructor (explicitly deallocates memory space referenced by pointers)

```
List( const List & rhs )
{
    init( );
    /* KV's cut …
    for( auto & x : rhs )
        push_back( x );
    */
    // more generic:
     const_iterator itr = rhs.begin();
     for (; itr != rhs.end(); ++itr)
        push_back(*itr);
}

List & operator= ( const List & rhs )
{
    List copy = rhs;
    std::swap( *this, copy );
    return *this;
}
```

List copy constructor and assignment operator

Usage:

```
List<int> lst1;

… fill lst1 with values …

List<int> lst2(lst1);

… remove odd values from lst2;

lst1 = lst2;

List<int> lst3 = lst1;
```

Now lst1, lst2, and lst3 all hold all odd values from the original lst1;

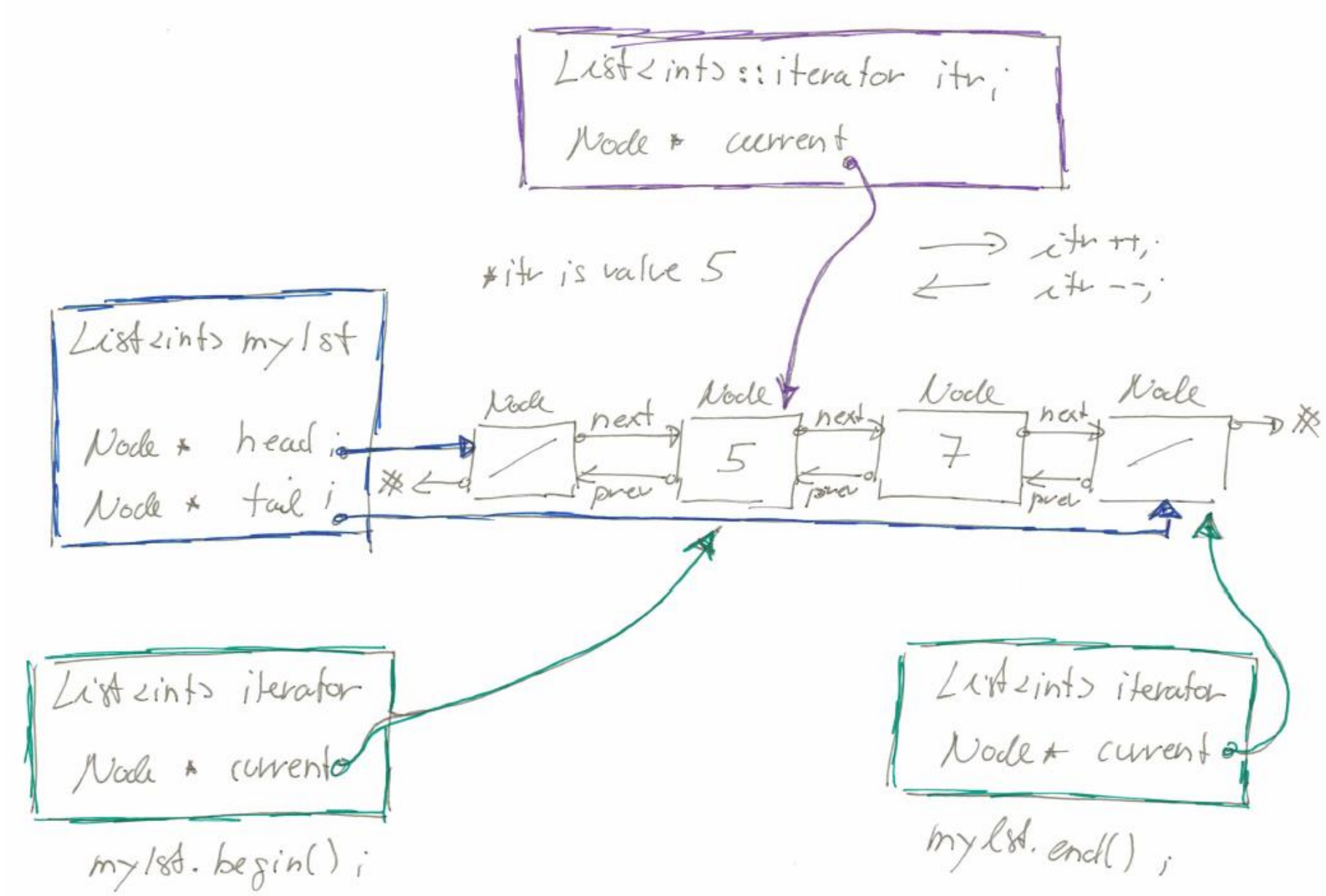
How would you test to verify?!

```
// Return iterator representing beginning of list.
// Mutator version is first, then accessor version.
iterator begin( )
  { return iterator( head->next ); }

const_iterator begin( ) const
  { return const_iterator( head->next ); }

// Return iterator representing endmarker of list.
// Mutator version is first, then accessor version.
iterator end( )
  { return iterator( tail ); }

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

List member functions that create and return iterators to

Front of List: notice how iterator(head->next) sets pointer current of iterator to the Node next to head; the first Node with value

Back of List: notice how iterator(tail) sets pointr current of iterator to the very last Node, which does NOT contain any value.

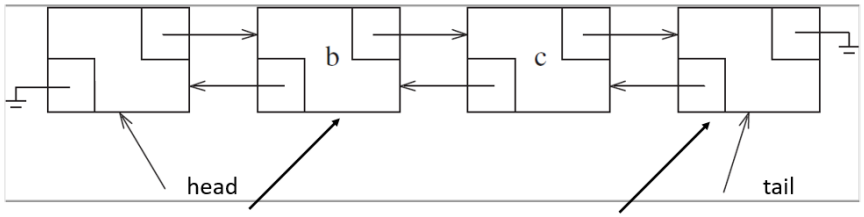

List<int>::iterator itr;
Node * current
*itr is value 5
--> itr++;
<-- itr--;
List<int> mylst
Node * head;
Node * tail;
Node
next
prev
5
7
List<int> iterator
Node * current
mylst.begin();
List<int> iterator
Node * current
mylst.end();

```
// Return number of elements currently in the list.
int size( ) const
  { return theSize; }

// Return true if the list is empty, false otherwise.
bool empty( ) const
  { return size( ) == 0; }

void clear( )
{
    while( !empty( ) )
        pop_front( );
}

// front, back, push_front, push_back, pop_front, and pop_back
// are the basic double-ended queue operations.
T & front( )
  { return *begin( ); }
```

Self explanatory …

- Obviously useful functions
- Easily implemented
- Notice how a class allows member functions to call other member functions of same class

Spot such instances!



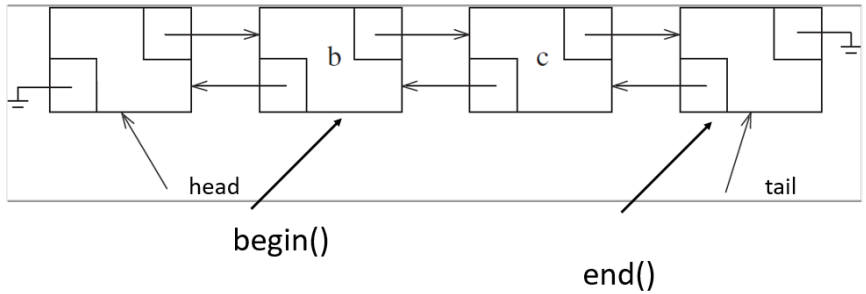
```
const T & front( ) const
  { return *begin( ); }

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

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

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

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

- Two more obvious and easy member functions: the front and back values sorted in the List

- Adding a value to the front and TO the back of a List is accomplished by INSERTION

  … which is subject of the

  NEXT LECTURE



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

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

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

    void init( )
    {
        theSize = 0;
        head = new Node;
        tail = new Node;
        head->next = tail;
        tail->prev = head;
    }
};

#endif
```

- Removing the front value and removing the back value are accomplished with ERASING

  … which will be subject of the NEXT LECTURE

- Having reached the end of the List implementation, we have another look at the private data members of List

- Also under private (so that only the class itself will use it): a "helper function" that bundles up those lines of code that initialize a List object to an EMPTY LIST

  - (zero size, no data stored, head and tail nodes without any Nodes between them marking the beginning and end of the List object.

Performance of class List

- We do not care about absolute numbers but the TREND by which the computational effort (time, number of significant ops) increases with the size N of the list (= number of data items stored)

- For lists, the computational “complexity” of its operations varies between constant O(1) and linear O(N).

  - O(1): the cost of an operation does not change with list size N
  - O(N): the cost of an operation increases in a linear fashion with list size N.

Performance of class List by Operation

Vector::	O(1)	O(N)	Note
empty()			
size()			
front()			
back()			
push_front()			
push_back()			
pop_front()			
pop_back()			
clear()			

Textbook Reading:

Weiss, DSAC++, Chapter 3, sections ….

- 3.5 on Implementation of List

Make the effort to read C++ code *line-by-line* and ponder its meaning!

Reading additional sections is not discouraged as preview and understanding material in larger context.