CSE 2020 Computer Science II

Module 3.2 – Advanced List Operations

Instructor: Kerstin Voigt

School of CSE, CSUSB

List ADT – Focus on Iterators

```
template <typename T>
class List
{
  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 } { }
    };

    //---- class iterator -------------------------

    class iterator
    {
      public:

        iterator( )
          :current(nullptr)
          { }

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

        T & operator* ( )
          { return current->data; }
```

In order to simplify presentation and understanding, let class iterator be defined on its own, without subclassing off a class const_iterator

- Two constructors:
  - Default
  - To be initialized with pointer to Node as current

- Dereferencing operator * with two usages:
  - cout << *itr; (access value)
  - *itr = 77; (on lhs of assign)

List ADT – Focus on Iterators

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

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


protected:

   Node* current;
   friend class List<T>;
};
```

```
// checking out itr++ vs. ++itr;
```

Microsoft Visual Studio Debug Console

```
14 49 4 61 62 19 95 46 1 44 71 74 86 48 40 76 15 67 31


*itr1: 14

*itr2: 14

effect of itr3 = ++itr1: *itr1: 49  *itr3: 49

effect of itr4 = itr2++: *itr2: 49  *itr4: 14
```

Both ++ operations moved the current pointer, different values returned;

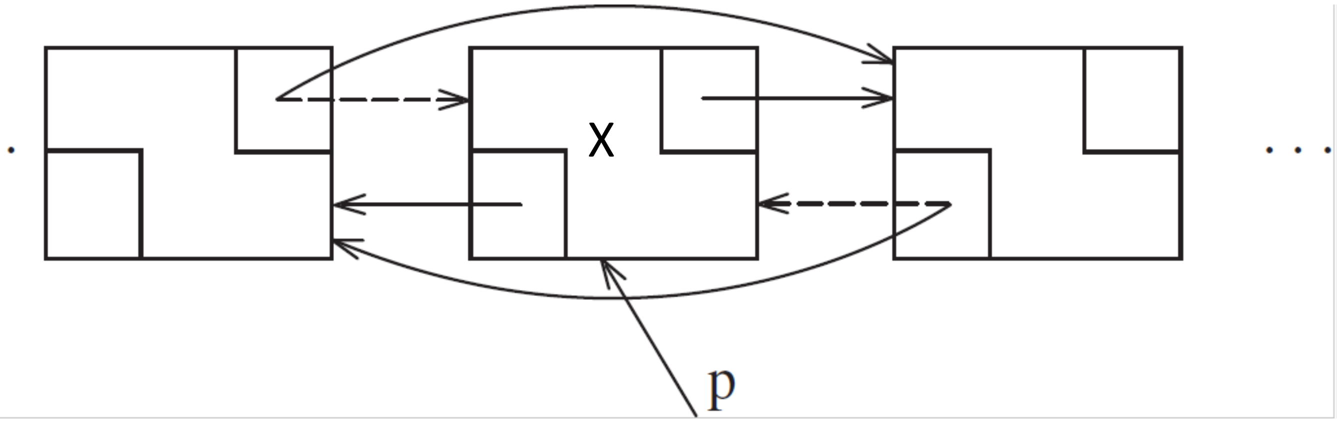
List ADT – Focus on Iterators

```
public:
  // for class List



   // … LOTS OF OMMITTED List code …

   // Return iterator representing beginning of list.

   iterator begin( )
     { return iterator( head->next ); }    // mutator

   // Return iterator representing endmarker of list.

   iterator end( )
     { return iterator( tail ); }           // mutator


  private:
    // for class List

    int   theSize;
    Node *head;
    Node *tail;

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

[Ignore yellow highlights … no significance in this context]

- See iterators used within List member functions begin() and end()
- Both call iterator constructor that will initialize the iterator's current pointer to the proper Node of list

Erase a Value from List at Iterator




```
Let p be pointer to Node to
be erased (p = itr.current)
```

```
Erase by “bridging” over
Node X
```

```
p->prev->next = p->next;
```

```
p->next->prev = p->prev;
```

```
Eliminate node X by
deleting p
```

```
Decrease theSize
```

Code for List Erase

```
// 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;
 }
```

Implementation choice:

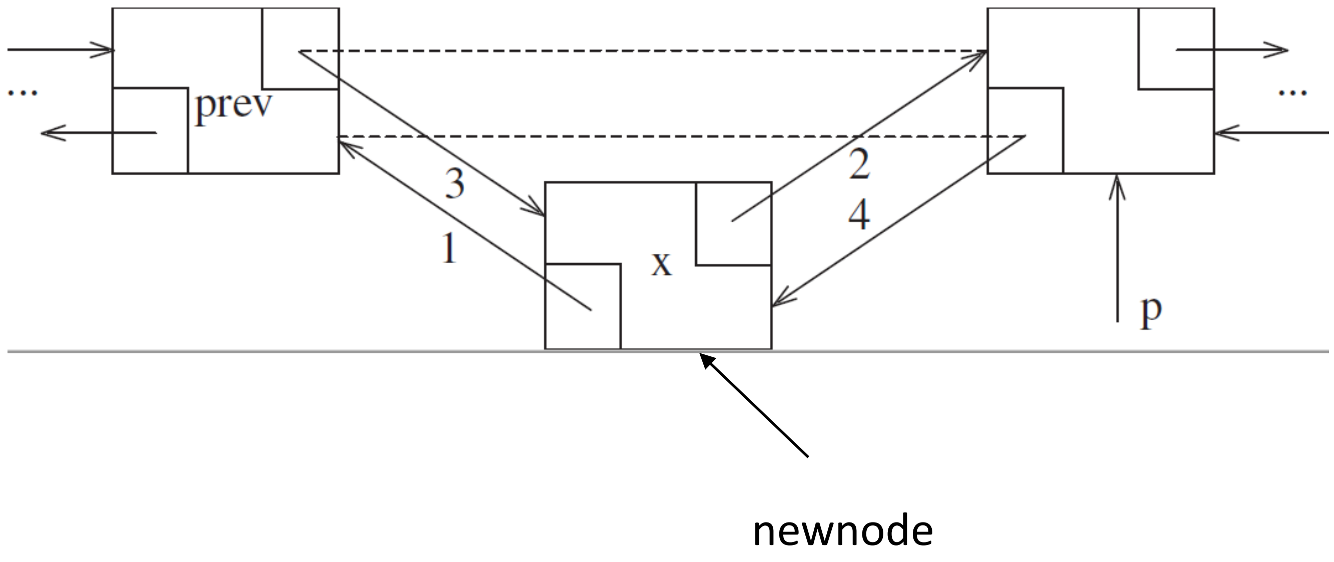
- Function returns an iterator on the Node next to the Node to be removed.

```
iterator retVal(p->next);
 …
return retVal;
```

A useful feature in certain contexts

Doing no harm when removal is the only objective

Insert a Value into before Iterator




```
Let p be pointer to Node in
front of which to insert
new value X

Insert by relinking to
connect new Node with value
X

newnode->prev = p->prev;

newnode->next = p;

p->prev->next = newnode;

p->prev = newnode;

Increase theSize
```

Code for List Insert

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

- COMPACT(!) CODE to accomplish insertion is taken straight from textbook
- Decode to understand equivalence to steps on previous slide
- Returns an iterator on the newly inserted Node

Pushing and Popping with Insert and Erase

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

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

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

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

Theoretical Expectation – O(?)

Erase at iterator:

Bridging a Node in order to remove it takes the same constant number of steps regardless of where in the List the remove is to happen.

Constant effort → O(1)

Insert at iterator:

Creating a new Node and linking it between two other Nodes (insert it) takes the same constant number of steps regardless of where in the List the insertion is to happen.

Constant effort → O(1)

Performance: Vectors vs. Linked Lists
average, worst case

	Vector	List	Note
insert()	O(N)	O(1)	
erase()	O(N)	O(1)	
push_back()	O(N) (*)	O(1)	(*) alloc new space, copy ☹
pop_back()	O(1)	O(1)	
push_front()	NA	O(1)	
pop_front()	NA	O(1)	
operator [](int)	O(1)	NA (*)	(*) cannot refer to position by number ☹

An academic exercise: List::operator [](int)

The Standard Template library <list> library does not provide a “random access operator” [ ] of the sort that we take for granted in vectors and arrays …. *Should we add one to our own class List implementation?*

Treating this as an academic questions, worth our exploration, see whether this woud be possible, if so, provide and implantation of operator [] for List, and comment on its merits.

→ Hypothetical Exam Question: Can you? Should you?

Textbook Reading:

Weiss, DSAC++, Chapter 3, sections ….

- 3.5 on Implementation of List … reread, in particular the subsections of iterators and the insert and erase member functions

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.

*** End of Module 3.2 ***