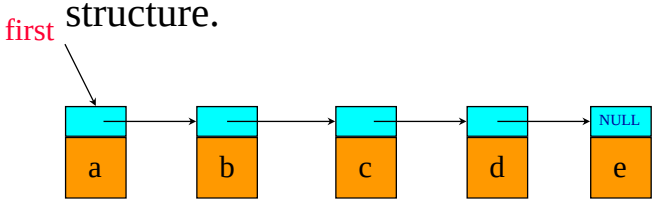
Iterators & Chain Variants

Iterators

- An iterator permits you to examine the elements of a data structure one at a time.
- C++ iterators
  - Forward iterator
  - Bidirectional iterator
  - Reverse iterator

Forward Iterator

Allows only forward movement through the elements of a data structure.



Iterator Class

- Assume that a forward iterator class ChainIterator is defined within the class Chain.
- Assume that methods Begin() and End() are defined for Chain.
  - Begin() returns an iterator positioned at element 0 (i.e., leftmost node) of list.
  - End() returns an iterator positioned one past last element of list (i.e., NULL or 0).

Using An Iterator

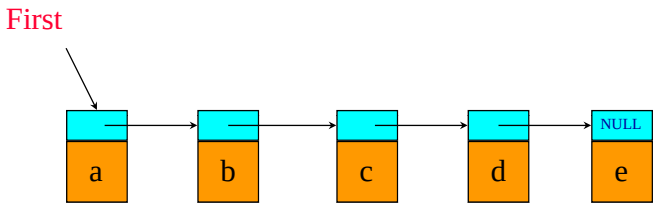
```
Chain<int>::iterator xHere = x.Begin();
Chain<int>::iterator xEnd = x.End();
for (; xHere != xEnd; xHere++)
    examine( *xHere);
```

vs

```
for (int i = 0; i < x.Size(); i++)
    examine(x.Get(i));
```

See next slide

Review Get(5)




```
desiredNode = first–>link–>link–>link–>link–>link;
                    // desiredNode = NULL
return desiredNode–>data;   // NULL.element
```


In Class Exercise

- Analyze the time complexity for traversal a list with size n

```
for (; xHere != xEnd; xHere++)
    examine( *xHere);
```

```
for (int i = 0; i < x.Size(); i++)
    examine(x.Get(i));
```


Required Methods for Forward Iterator

- iterator(T* thePosition)

  Constructs an iterator positioned at specified element (Like Begin())
- dereferencing operators * and ->
- Post and pre increment operators ++
- Equality testing operators == and !=

A Forward Iterator For Chain
See Program 4.10 @P 191

```
class ChainIterator {
public:
    // some typedefs omitted
    // constructor comes here
    // dereferencing operators * & ->, pre and post
    // increment, and equality testing operators
private:
  ChainNode<T> *current;
};
```


Constructor

```
ChainIterator(ChainNode<T> * startNode = 0)
  {current = startNode;}
```


Dereferencing Operators

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

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

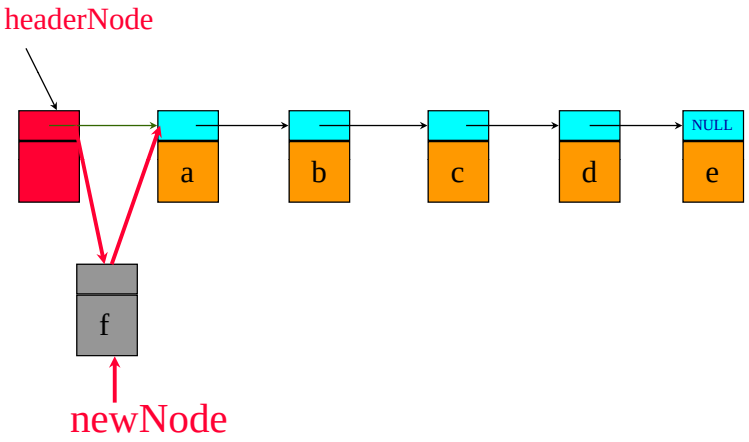

Increment

```
ChainIterator& operator++() // preincrement
  {current = current->link; return *this;}

ChainIterator& operator++(int) // postincrement
{
  ChainIterator old = *this;
  current = current->link;
  return old;
}
```

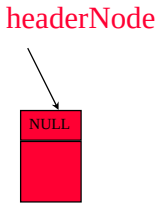
Equality Testing

```
bool operator!=(const ChainIterator right) const
  {return current != right.current;}

bool operator==(const ChainIterator right) const
  {return current == right.current;}
```

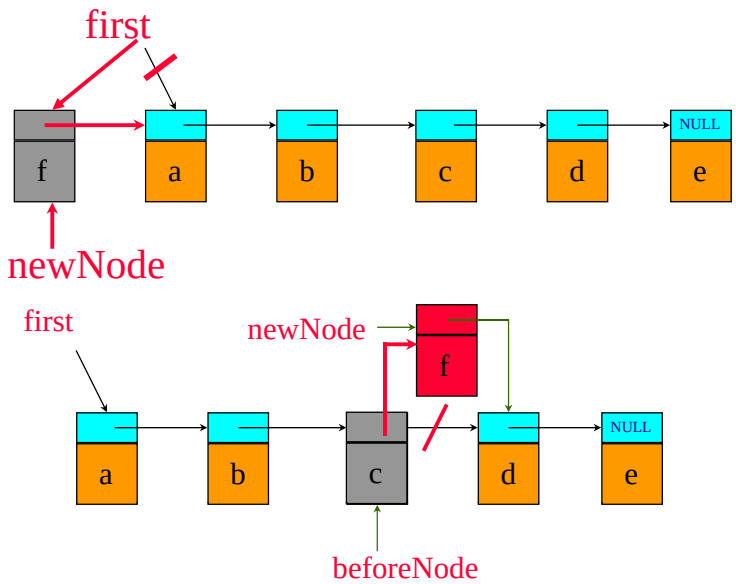

Chain With Header Node and Insertion



Empty Chain With Header Node



Compare with the Linked list without Header Node (Two different Cases)

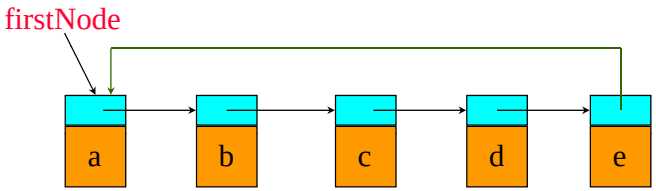


Insertion (See Lec. 11)

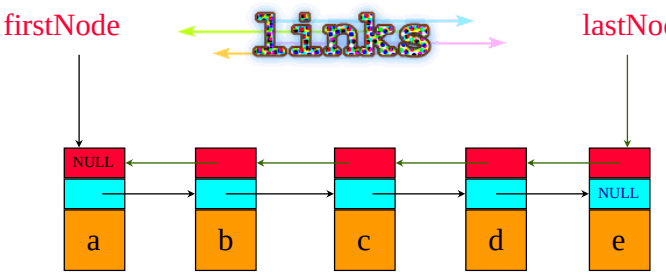
```
template<class T>
void Chain<T>::Insert(int theIndex,
                      const T& theElement)
{
  if (theIndex < 0)
     throw "Bad insert index";

  if (theIndex == 0)
     // insert at front
     first = new chainNode<T>
           (theElement, first);
  else
    {  // find predecessor of new element
      ChainNode<T>* p = first;
      for (int i = 0; i < theIndex - 1; i++)
      {if (p == 0)
       throw "Bad insert index";
       p = p->next;}
      // insert after p
      p->link = new ChainNode<T>
                (theElement, p->link);
    }
}
```

Can be removed.

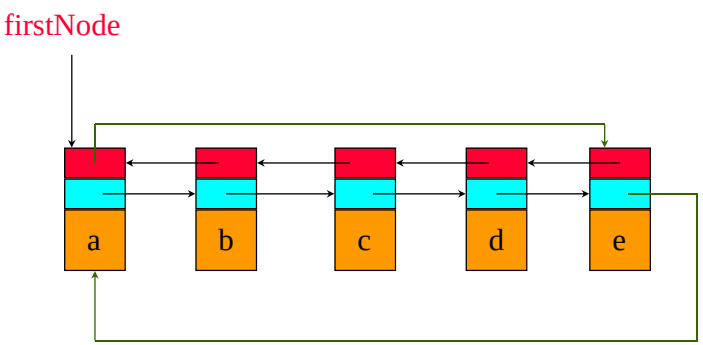
Circular List



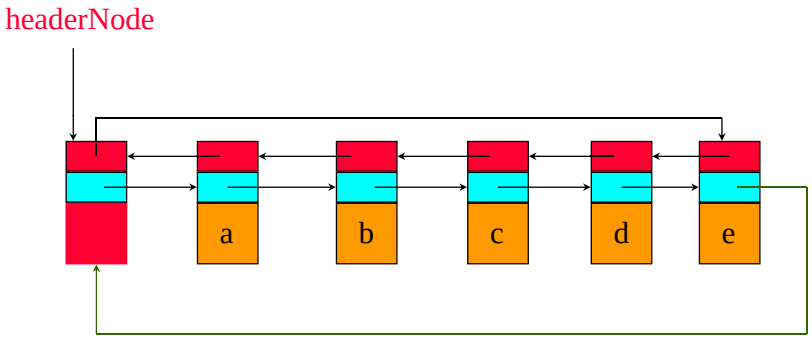
Doubly Linked List



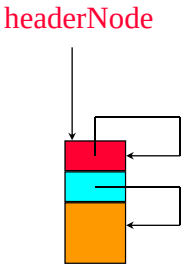
Doubly Linked Circular List



Doubly Linked Circular List With Header Node



Empty Doubly Linked Circular List With Header Node



Bidirectional Iterator

Allows both forward and backward movement through the elements of a data structure.

Bidirectional Iterator Methods

- iterator(T* thePosition)

  Constructs an iterator positioned at specified element
- dereferencing operators * and ->
- Post and pre increment and decrement operators ++ and —
- Equality testing operators == and !=

Homework

- Sec. 4.10 Exercise 5 @P228
  - 可利用 pesudo code + ADT表示