Iterators & Chain Variants

Iterators

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

Forward Iterator

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

Forward Iterator Methods

- iterator(T* thePosition)

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

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 !=

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));
```

Merits Of An Iterator

- it is often possible to implement the ++ and -- operators so that their complexity is less than that of Get.
- this is true for a chain
- many data structures do not have a get by index method
- iterators provide a uniform way to sequence through the elements of a data structure

A Forward Iterator For Chain

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

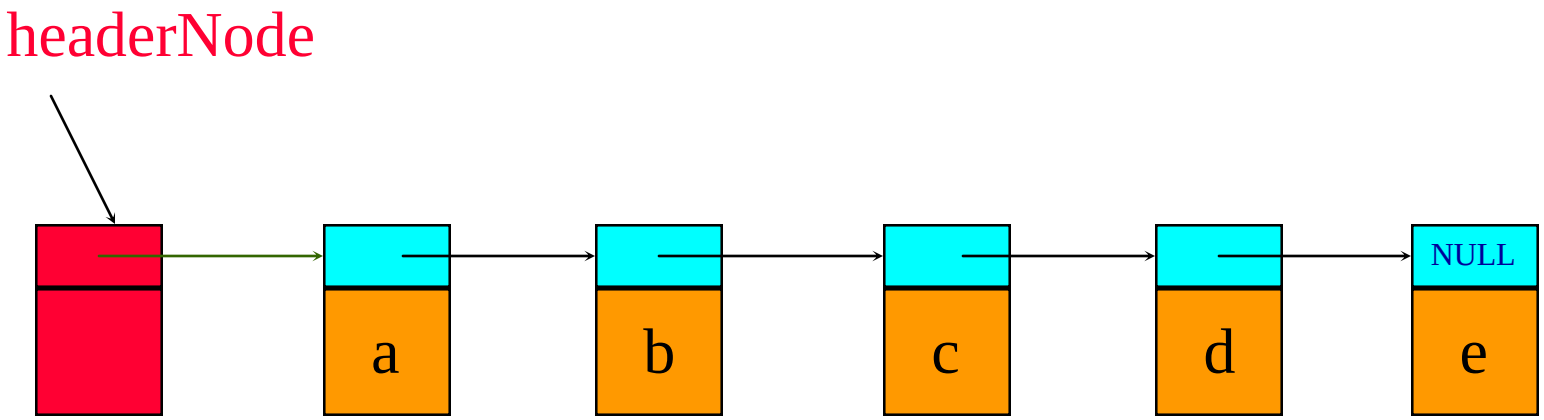
Increment

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

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

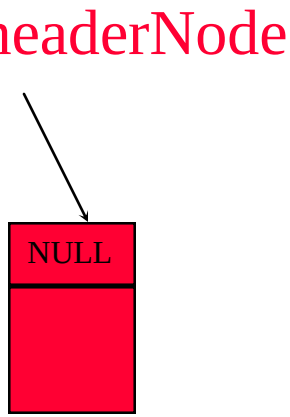
Equality Testing

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

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

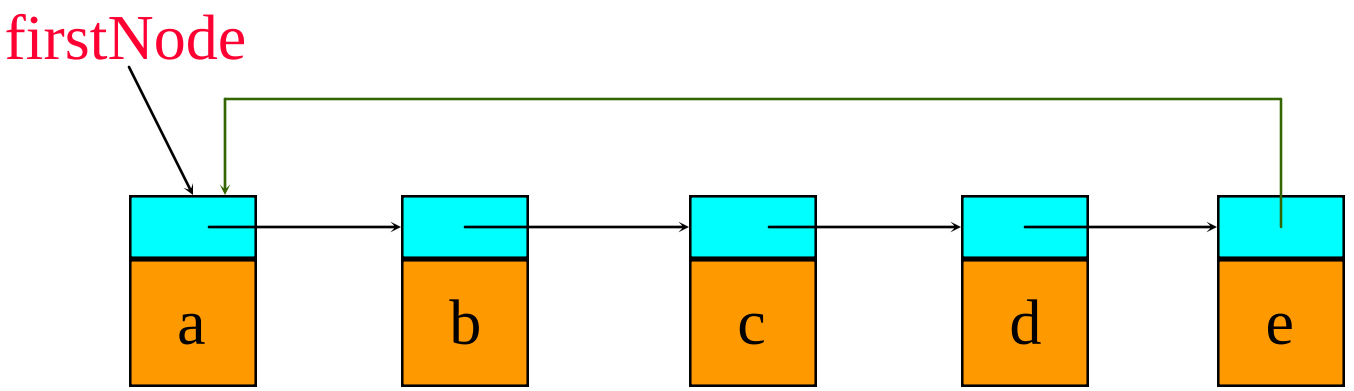
Chain With Header Node



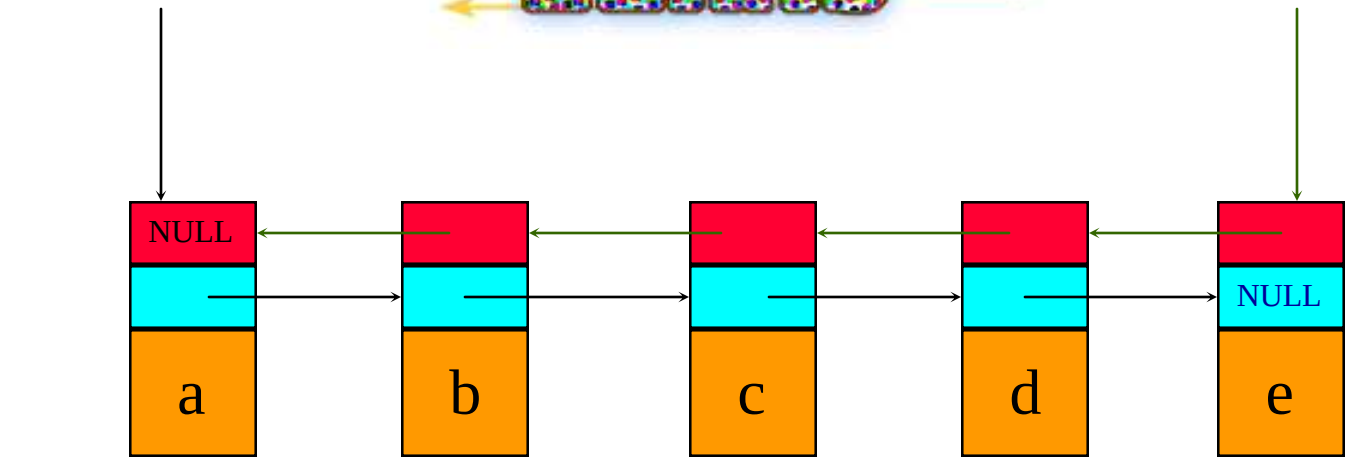
Empty Chain With Header Node



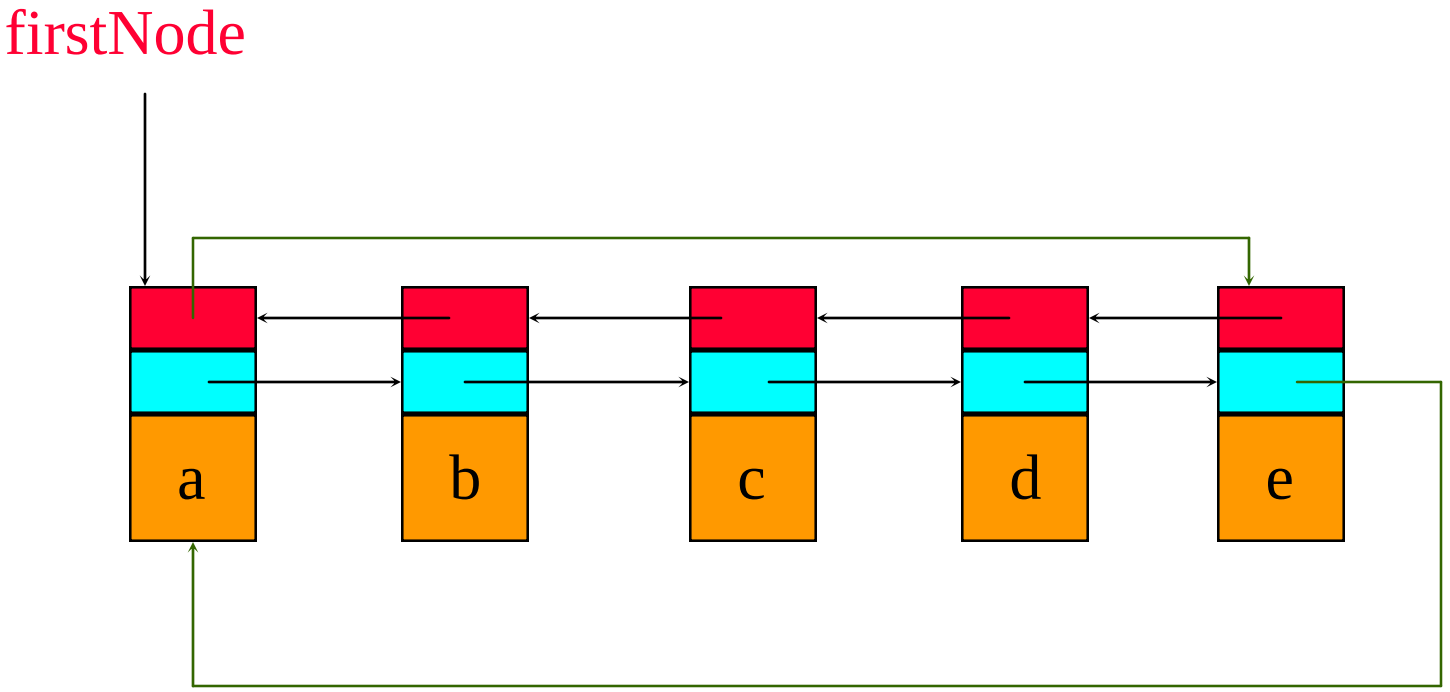
Circular List



Doubly Linked List

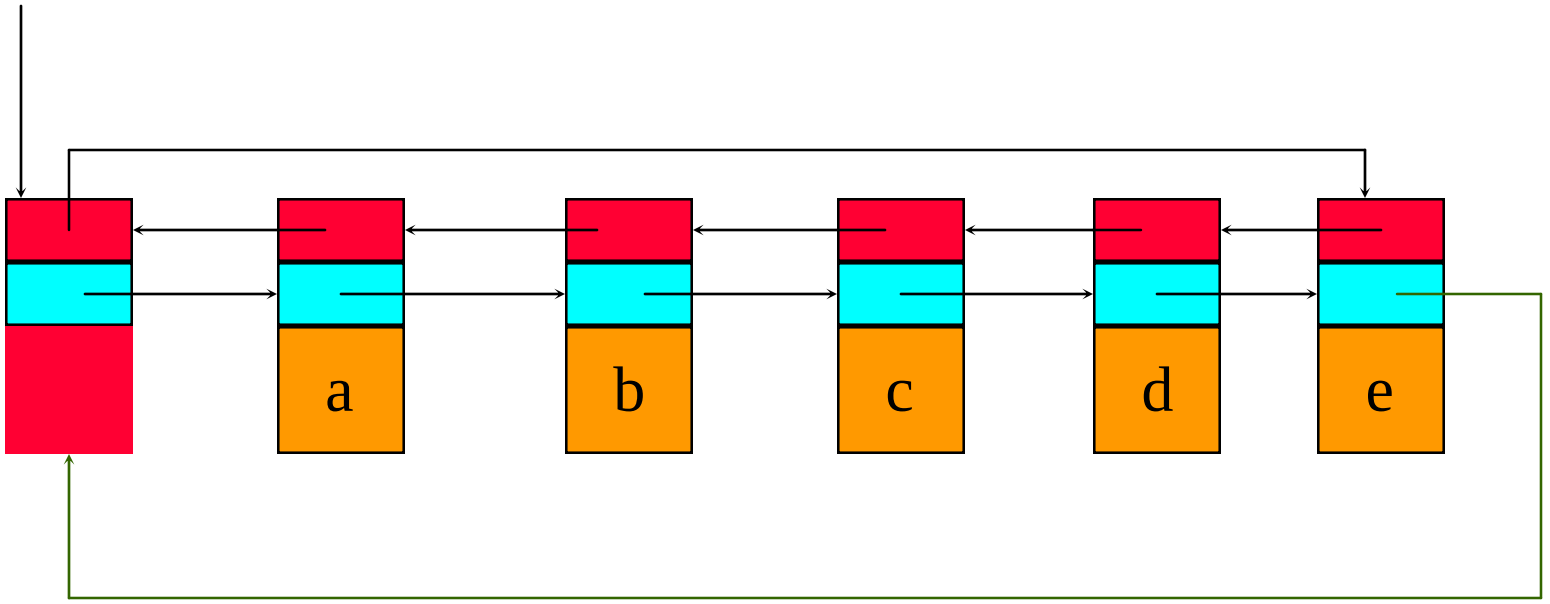


Doubly Linked Circular List

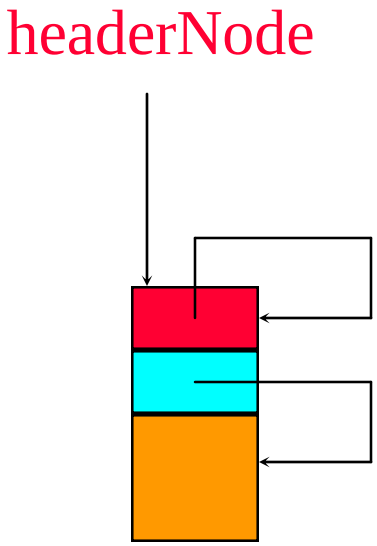


Doubly Linked Circular List With Header Node

headerNode



Empty Doubly Linked Circular List With Header Node



 The STL Class list 

- Linked implementation of a linear list.
- Doubly linked circular list with header node.
- Has many more methods than our Chain.
- Similar names and signatures.