Chapter Four

Linked Lists in More Details

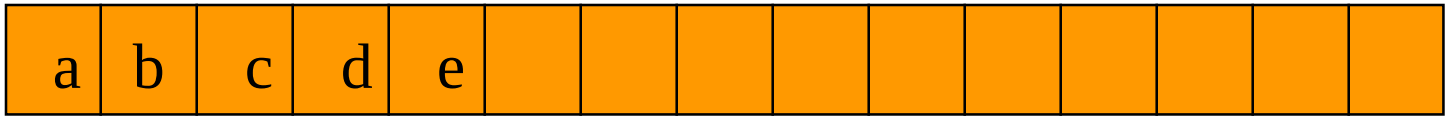
Linked Lists

- list elements are stored, in memory, in an arbitrary order

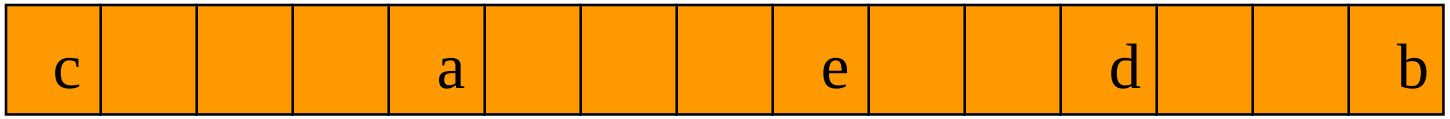
- explicit information (called a link) is used to go from one element to the next

Memory Layout

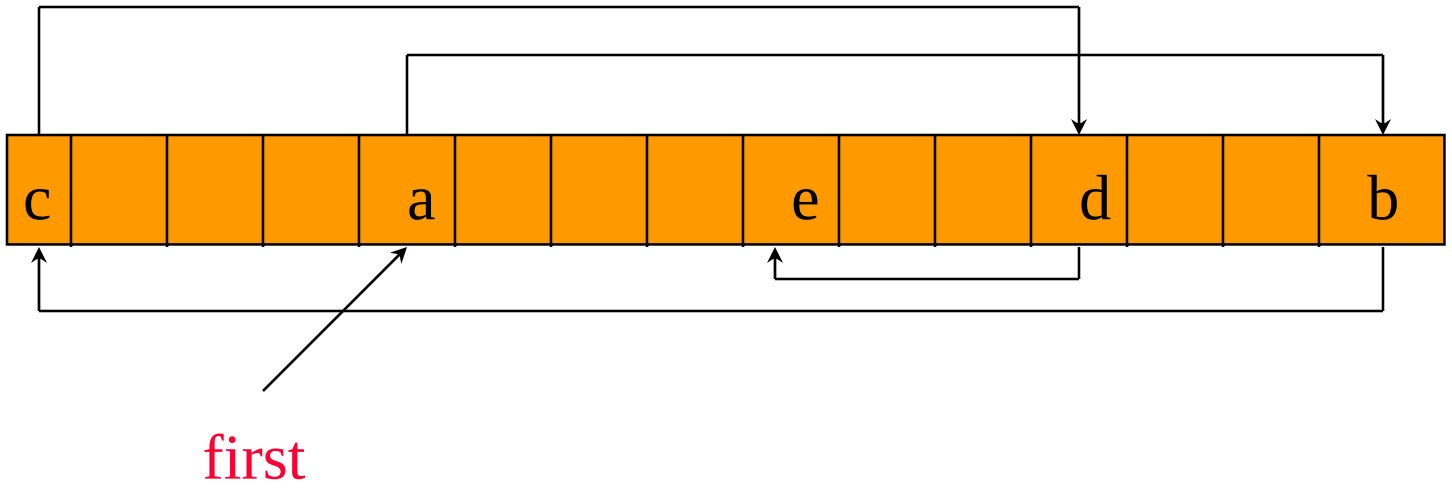
Layout of L = (a,b,c,d,e) using an array representation.



A linked representation uses an arbitrary layout.



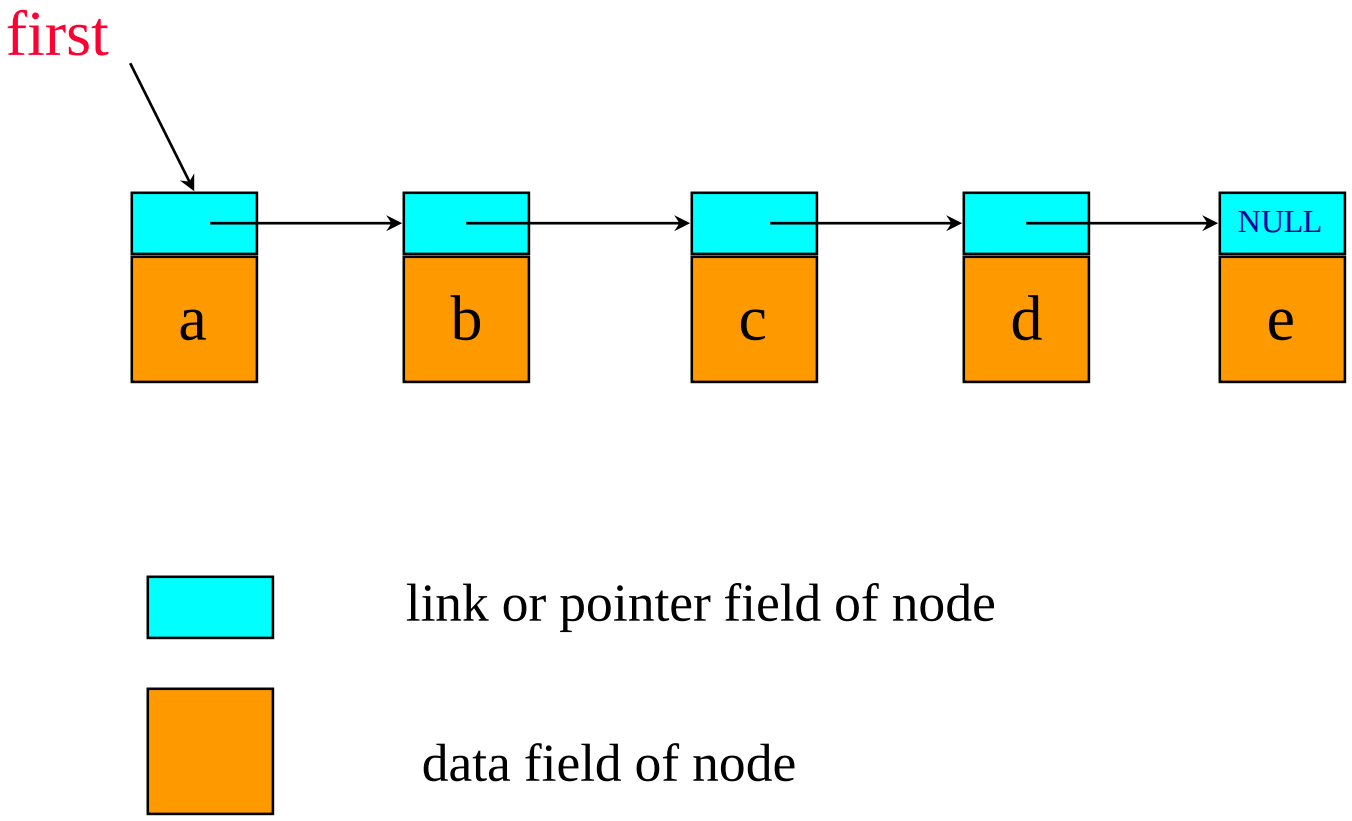
Linked Representation



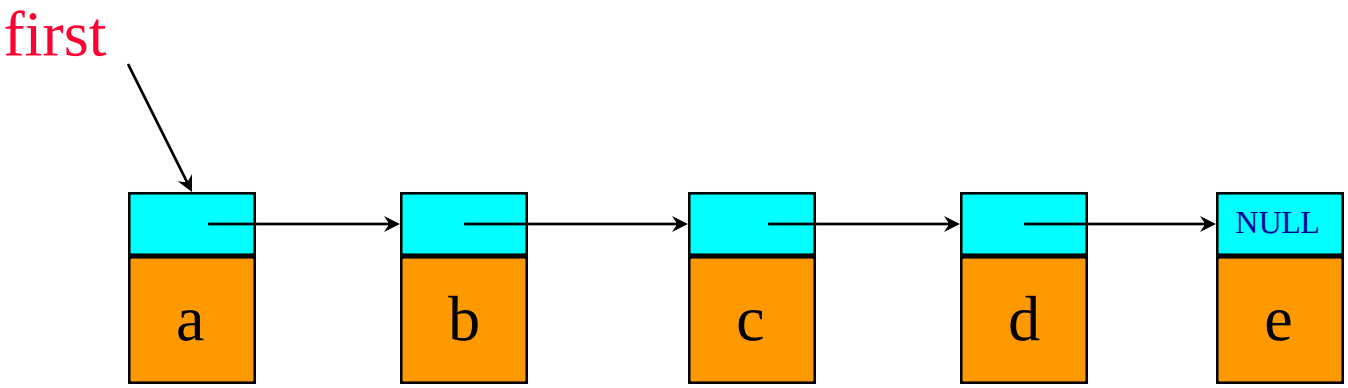
pointer (or link) in e is NULL

use a variable first to get to the first element a

Normal Way To Draw A Linked List



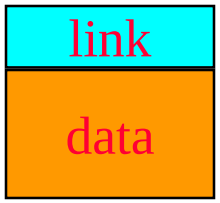
Chain



- A chain is a linked list in which each node represents one element.
- There is a link or pointer from one element to the next.
- The last node has a NULL (or 0) pointer.

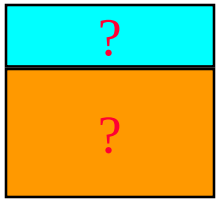
Node Representation

```
template <class T>
class ChainNode
{
  private:
    T data;
    ChainNode<T> *link;

    // constructors come here
};
```

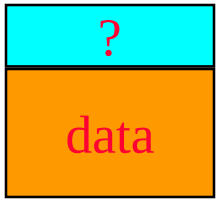


Constructors Of ChainNode

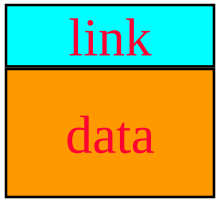
ChainNode() {}



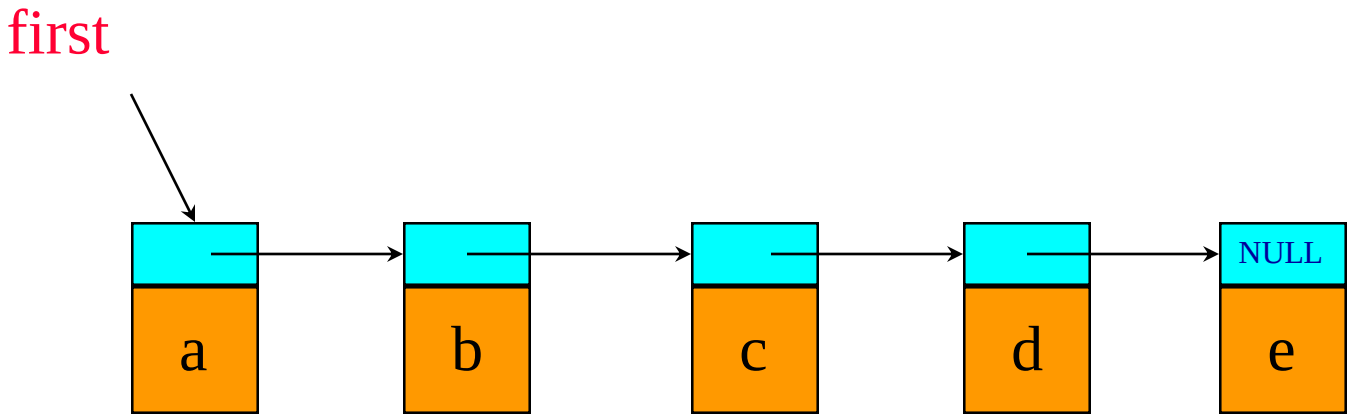
ChainNode(const T& data)
{this->data = data;}



ChainNode(const T& data, chainNode<T>* link)
{this->data = data;
this->link = link;}



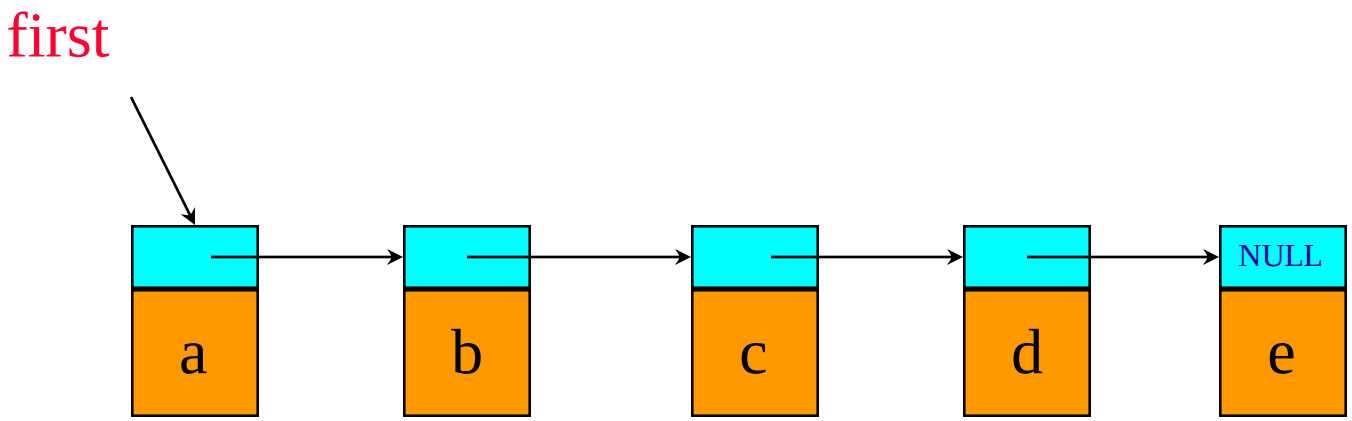
Get(0)



desiredNode = first; // gets you to first node

return desiredNode–>data;

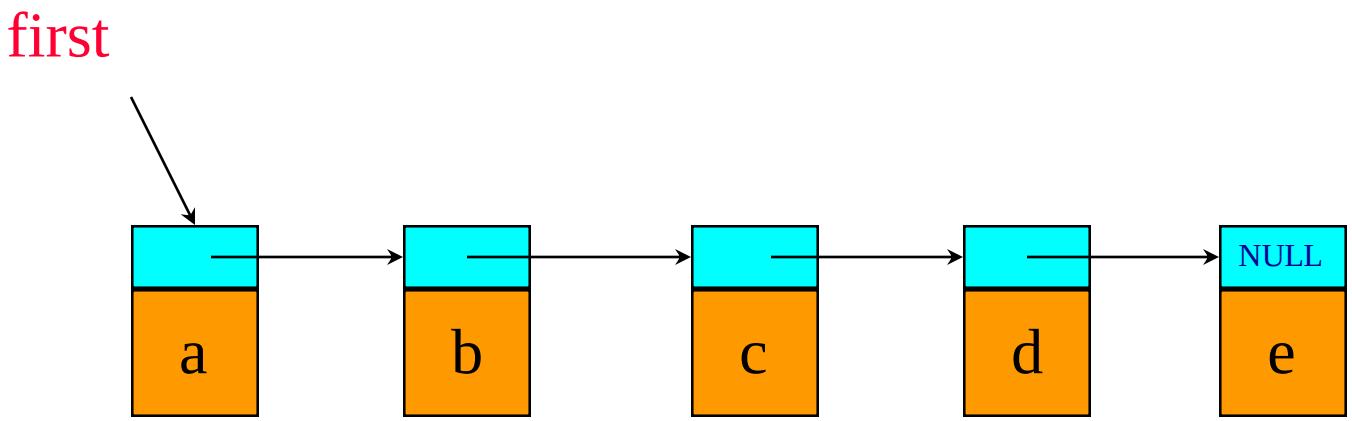
Get(1)



desiredNode = first–>link; // gets you to second node

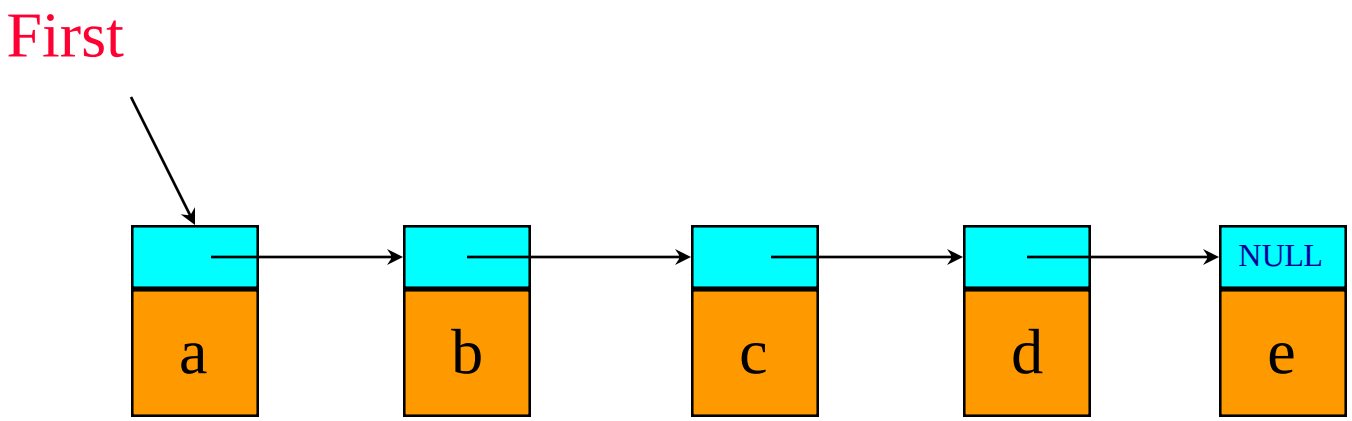
return desiredNode–>data;

Get(2)



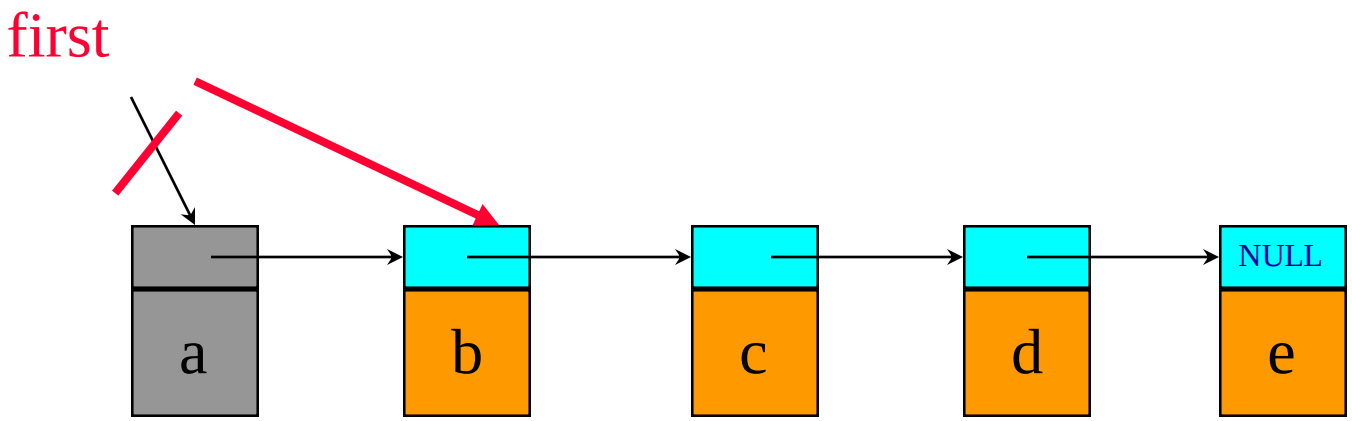

```
desiredNode = first–>link–>link; // gets you to third node
return desiredNode–>data;
```

Get(5)




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

Delete An Element



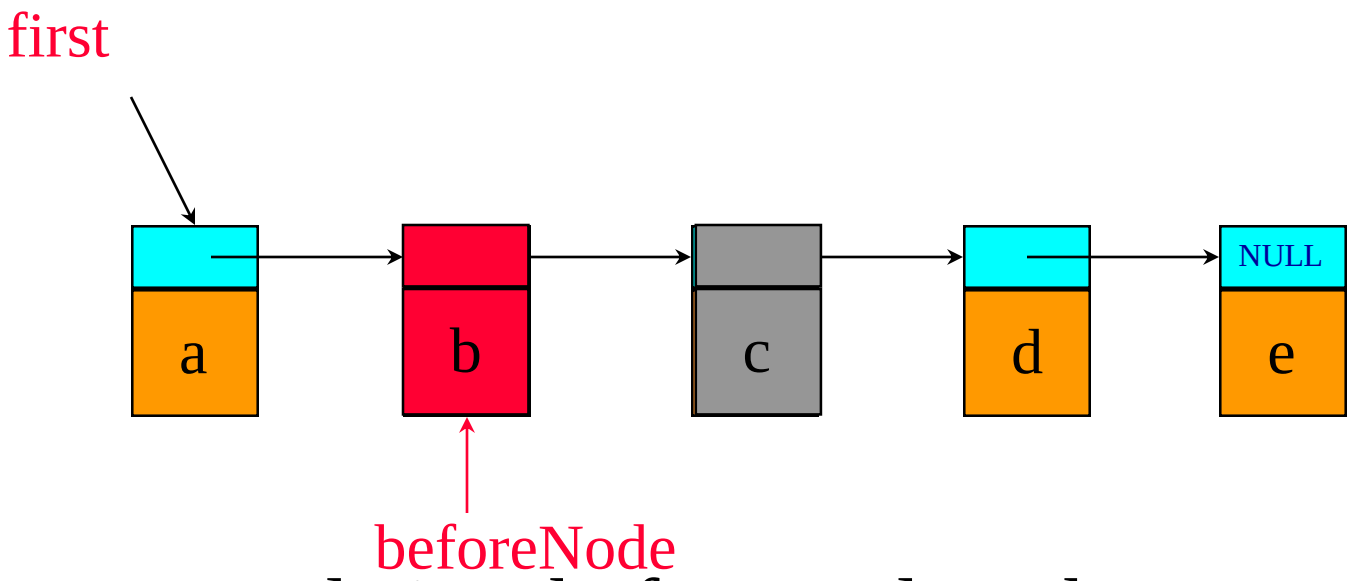
Delete(0)

deleteNode = first;

first = first–>link;

delete deleteNode;

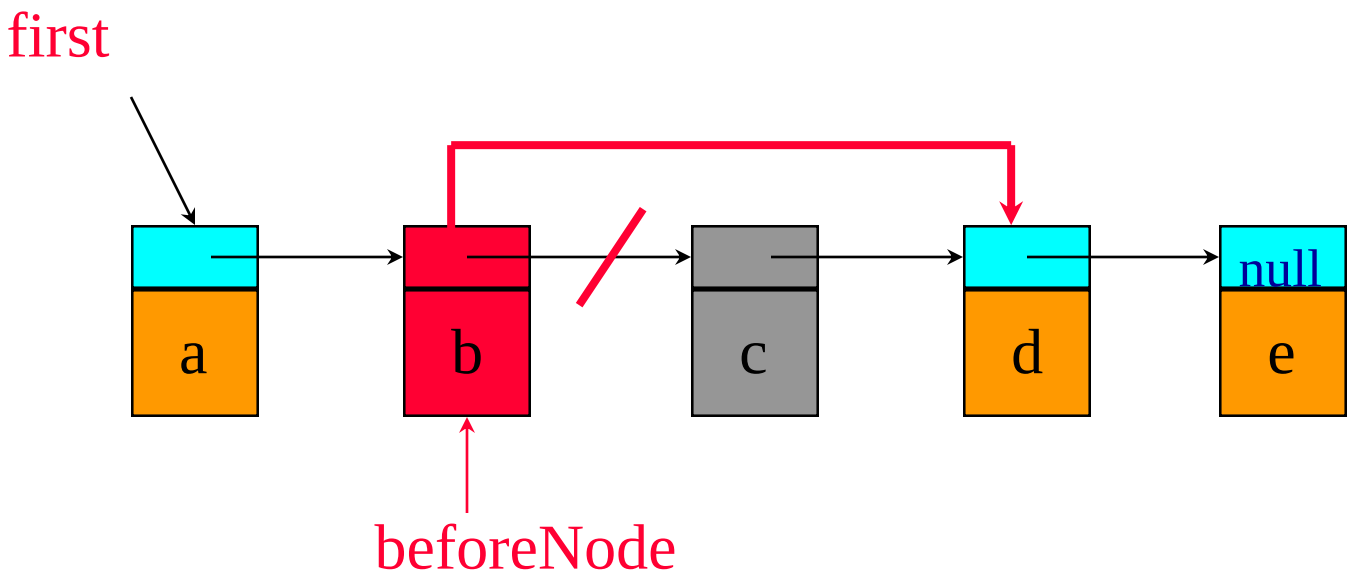
Delete(2)



first get to node just before node to be removed

beforeNode = first–>link;

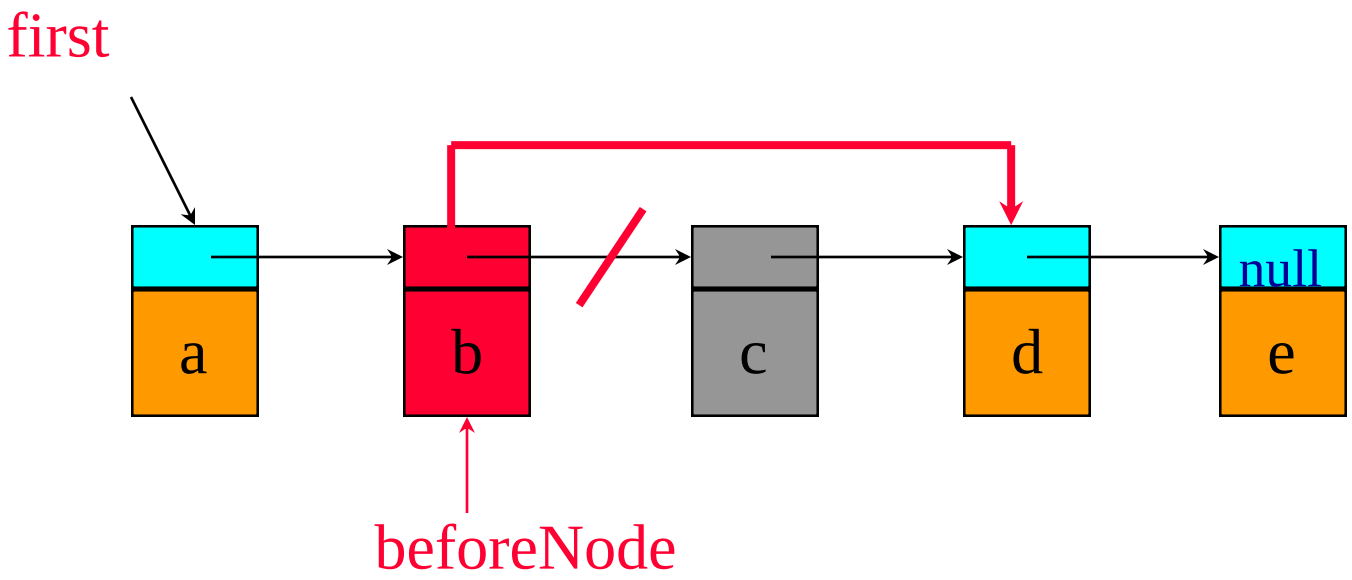
Delete(2)



save pointer to node that will be deleted

deleteNode = beforeNode–>link;

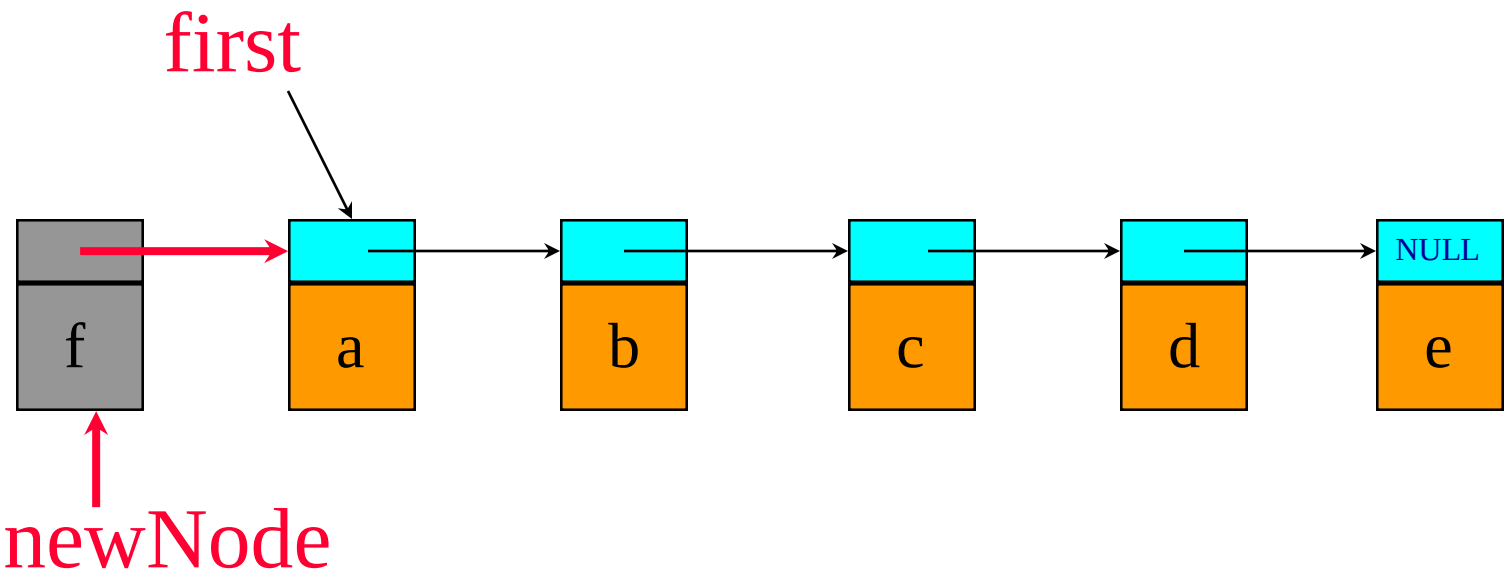
Delete(2)



now change pointer in beforeNode

```
beforeNode–>link = beforeNode–>link–>link;
delete deleteNode;
```

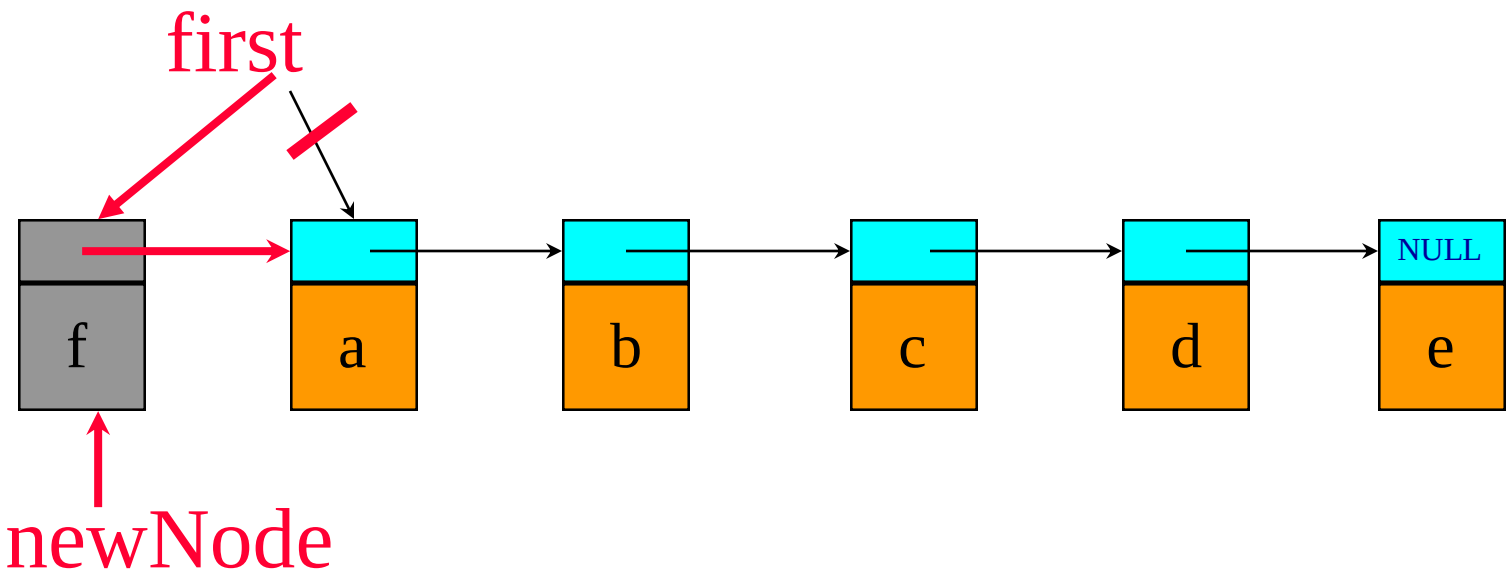
Insert(0,'f')



Step 1: get a node, set its data and link fields

```
newNode = new ChainNode<char>(theElement,
                              first);
```

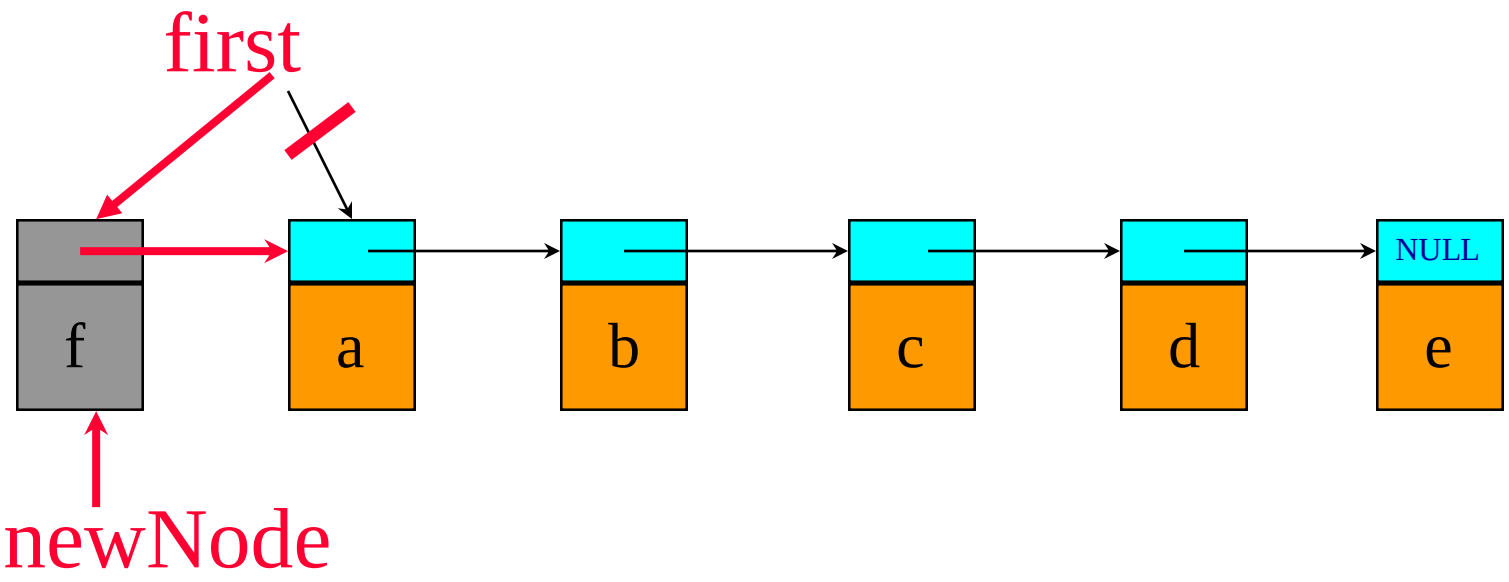
Insert(0,'f')



Step 2: update first

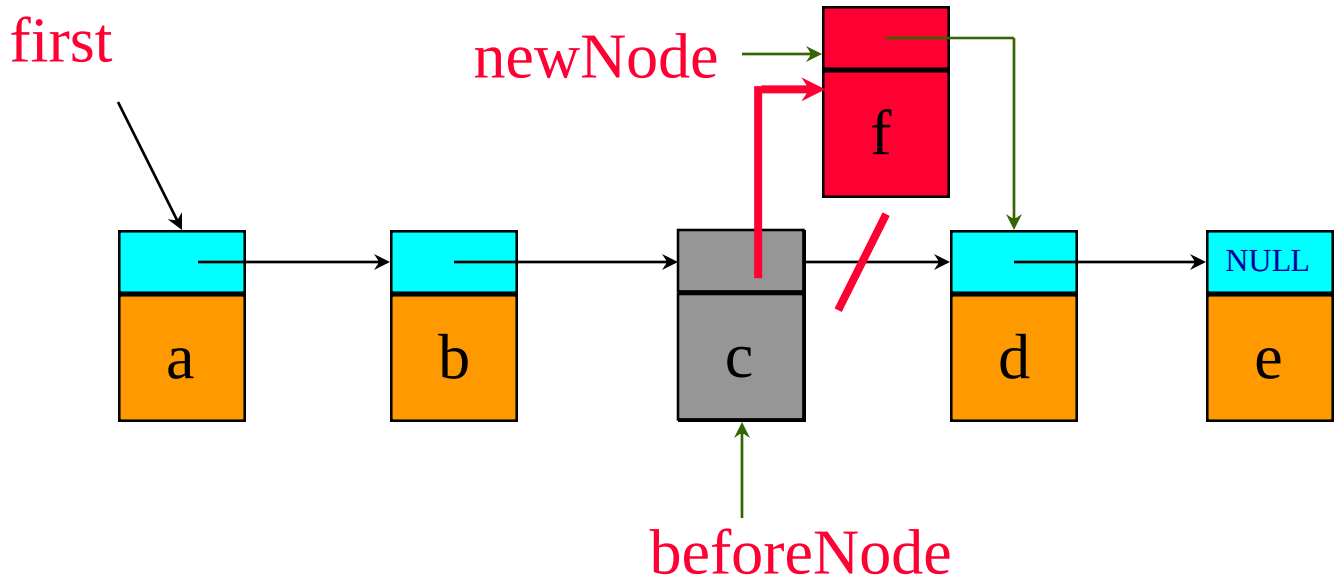
first = newNode;

One-Step Insert(0,'f')



first = new chainNode<char>('f', first);

Insert(3,'f')



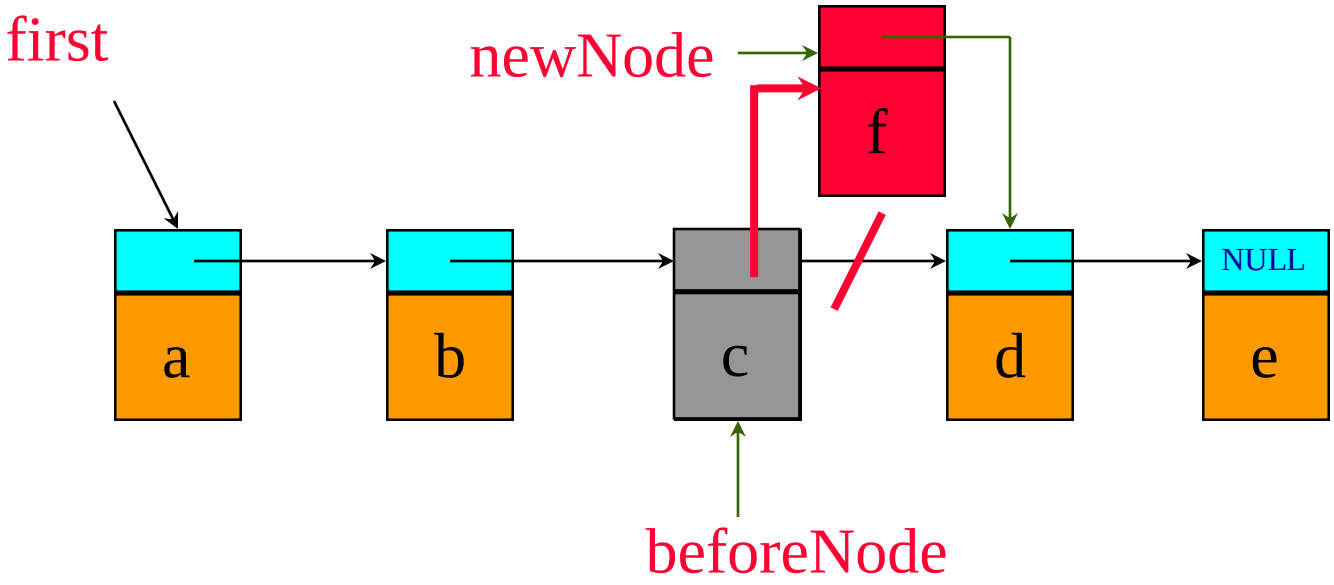
- first find node whose index is 2
- next create a node and set its data and link fields

```
ChainNode<char>* newNode = new ChainNode<char>( 'f',
                                                beforeNode->link);
```

- finally link beforeNode to newNode

```
beforeNode->link = newNode;
```

Two-Step Insert(3,'f')



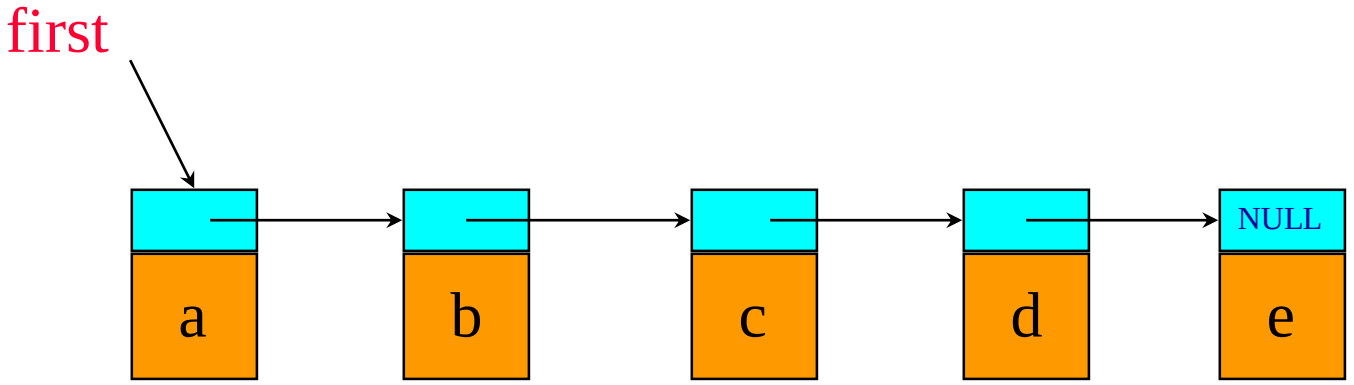

```
beforeNode = first–>link–>link;
beforeNode–>link = new ChainNode<char>
                                ('f', beforeNode–>link);
```

The Template Class Chain

Chain

- Linear list.
- Each element is stored in a node.
- Nodes are linked together using pointers.

The Class Chain



Use ChainNode

link (datatype ChainNode<T>*)

data (datatype T)

The Template Class Chain

```
template<class T>
class Chain
{
   public:
      Chain() {first = 0;}
         // constructor, empty chain
      ~Chain(); // destructor
      bool IsEmpty() const {return first == 0;}
      // other methods defined here
   private:
      ChainNode<T>* first;
};
```

The Destructor

```
template<class T>
chain<T>::~chain()
{// Chain destructor. Delete all nodes
 // in chain.
   while (first != NULL)
   {// delete first
      ChainNode<T>* next = first->link;
      delete first;
      first = next;
   }
}
```

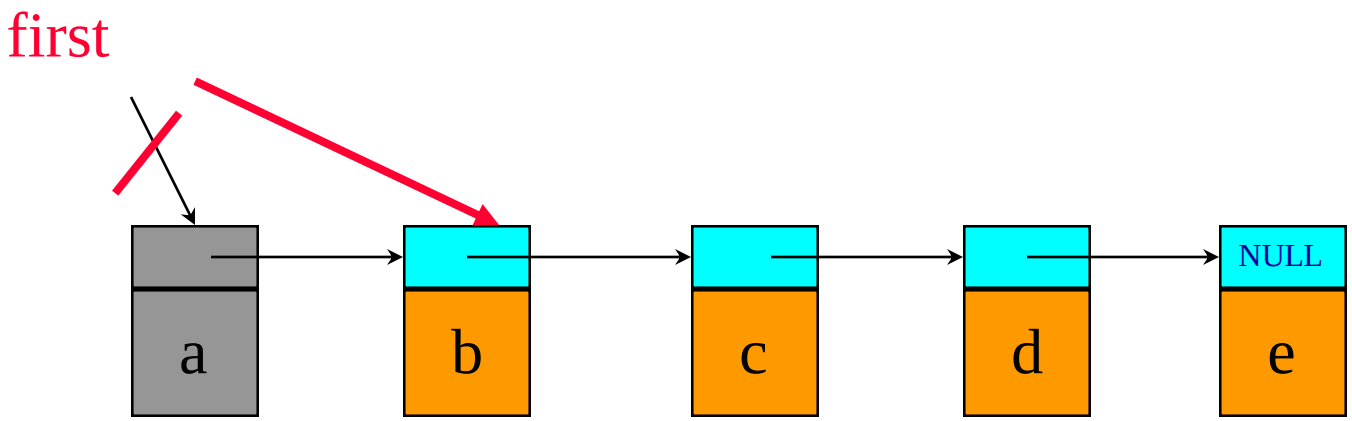
The Method IndexOf

```
template<class T>
int Chain<T>::IndexOf(const T& theElement) const
{
   // search the chain for theElement
   ChainNode<T>* currentNode = first;
   int index = 0;  // index of currentNode
   while (currentNode != NULL &&
          currentNode->data != theElement)
   {
      // move to next node
      currentNode = currentNode->next;
      index++;
   }
```

The Method IndexOf

```
    // make sure we found matching element
    if (currentNode == NULL)
        return -1;
    else
        return index;
}
```

Delete An Element



delete(0)

deleteNode = first;

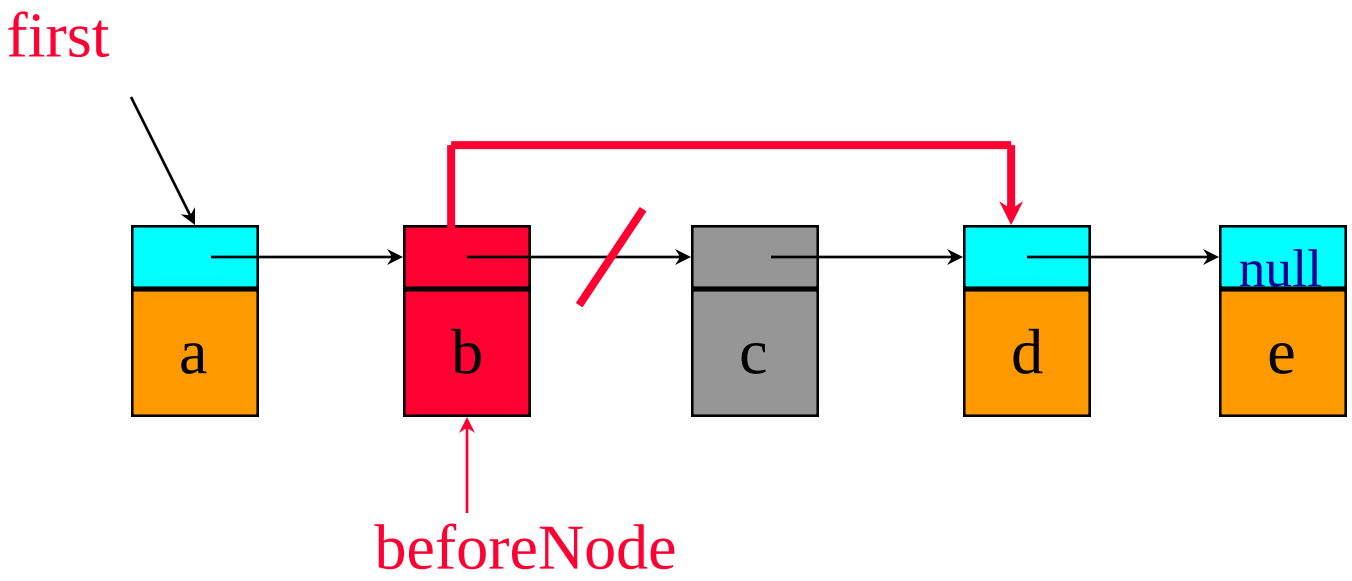
first = first–>link;

delete deleteNode;

Delete An Element

```
template<class T>
void Chain<T>::Delete(int theIndex)
{
   if (first == 0)
   throw "Cannot delete from empty chain";
   ChainNode<T>* deleteNode;
   if (theIndex == 0)
   {// remove first node from chain
      deleteNode = first;
      first = first->link;
   }
```

Delete(2)



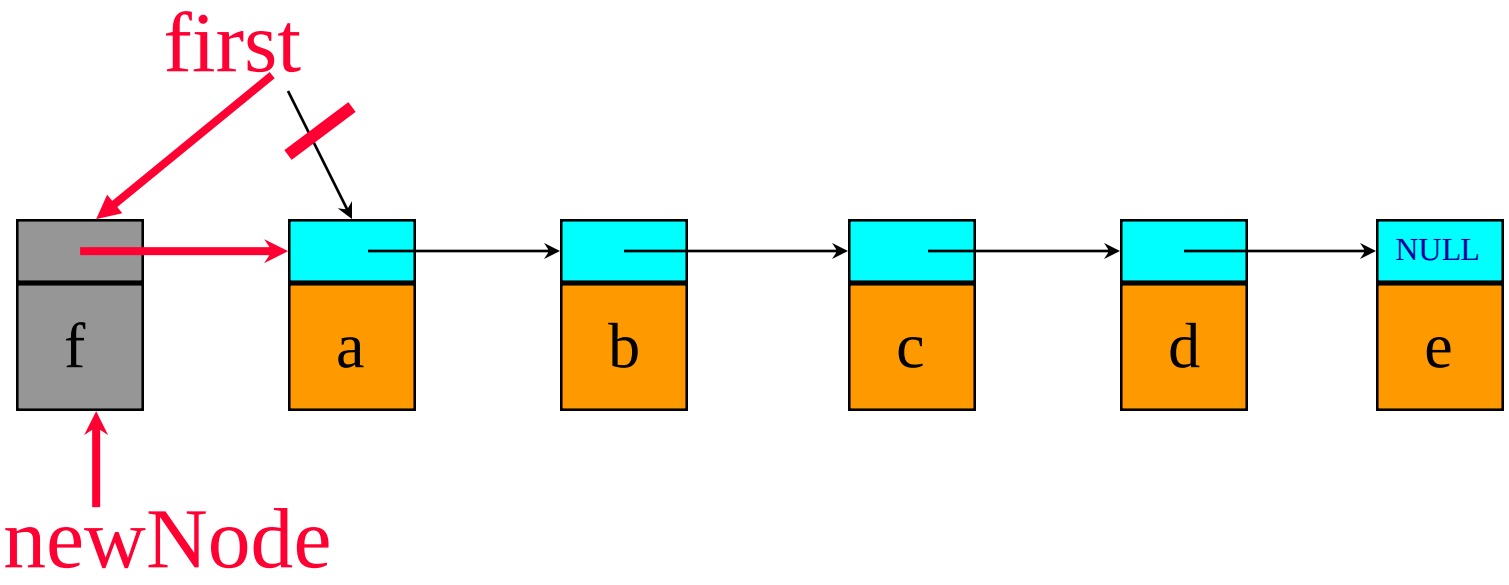
Find & change pointer in beforeNode

```
beforeNode–>link = beforeNode–>link–>link;
delete deleteNode;
```

Delete An Element

```
else
{  // use p to get to beforeNode
   ChainNode<T>* p = first;
   for (int i = 0; i < theIndex - 1; i++)
   {if (p == 0)
      throw "Delete element does not exist";
    p = p->next;}
   deleteNode = p->link;
   p->link = p->link->link;
 }
 delete deleteNode;
}
```

One-Step Insert(0,'f')

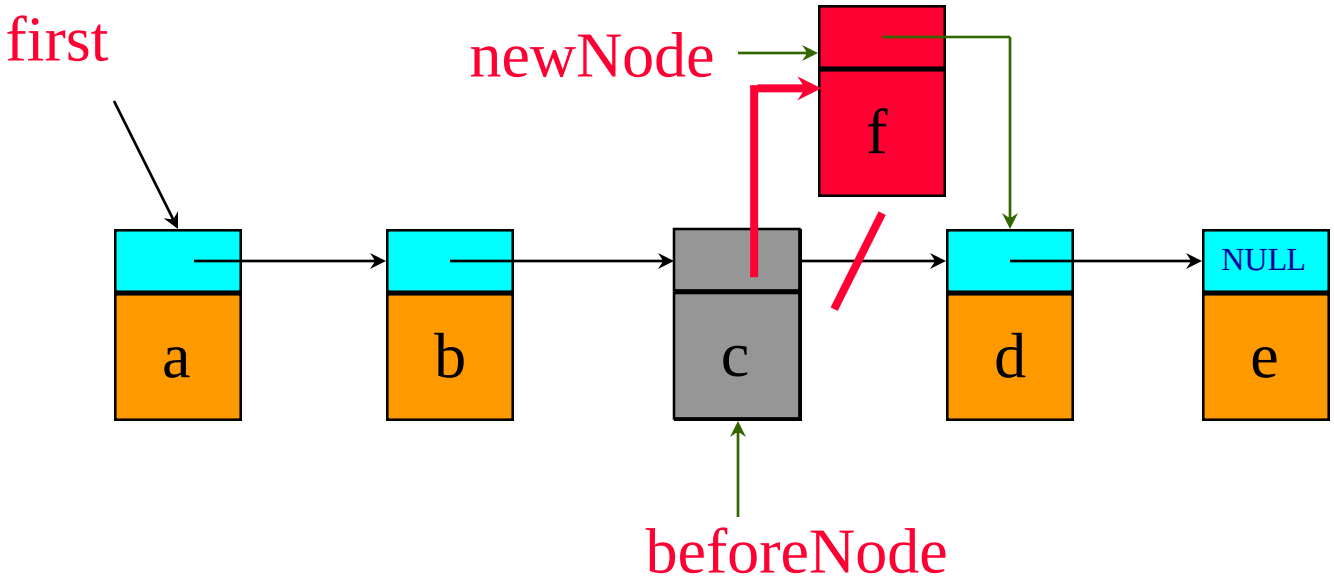


first = new ChainNode<char>('f', first);

Insert An Element

```
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);
```

Two-Step Insert(3,'f')




```
beforeNode = first–>link–>link;
beforeNode–>link = new ChainNode<char>
                                ('f', beforeNode–>link);
```

Inserting An Element

```
    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);
    }
}
```

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;}
```

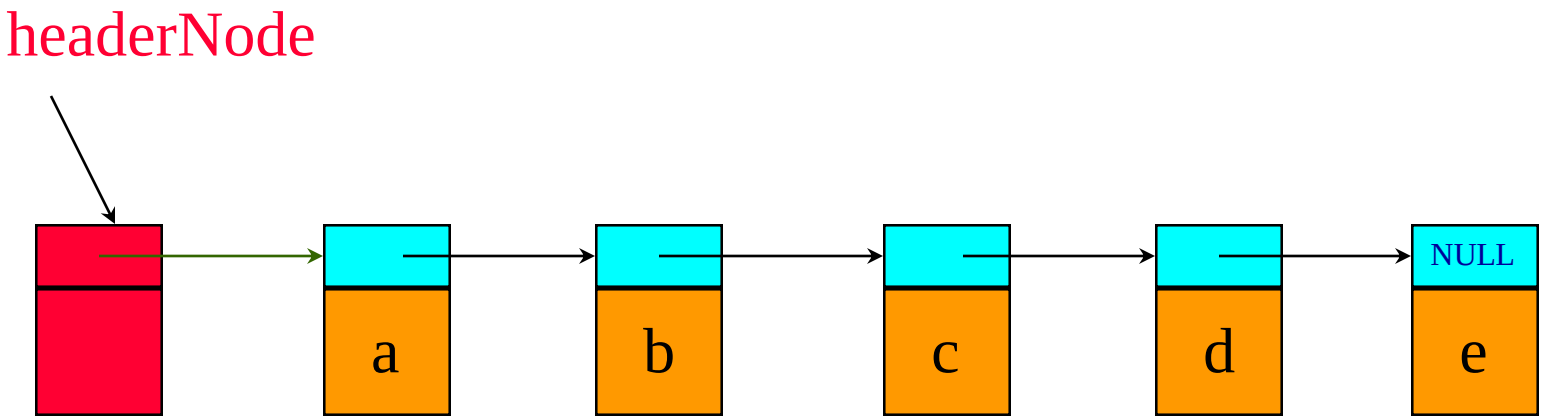
Increment

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

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

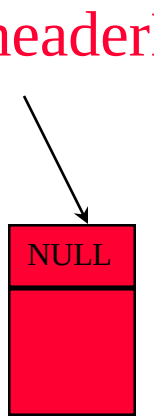
Equality Testing

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

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

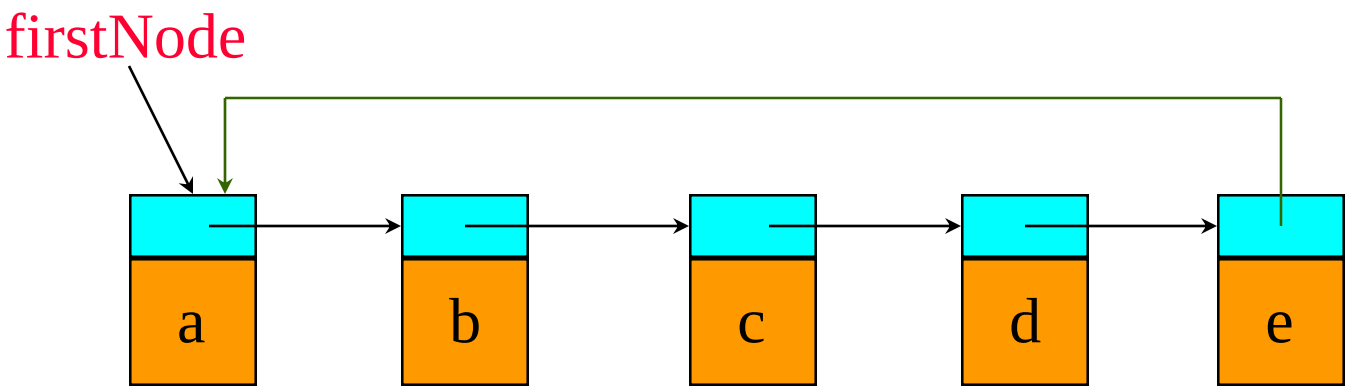
Chain With Header Node



Empty Chain With Header Node



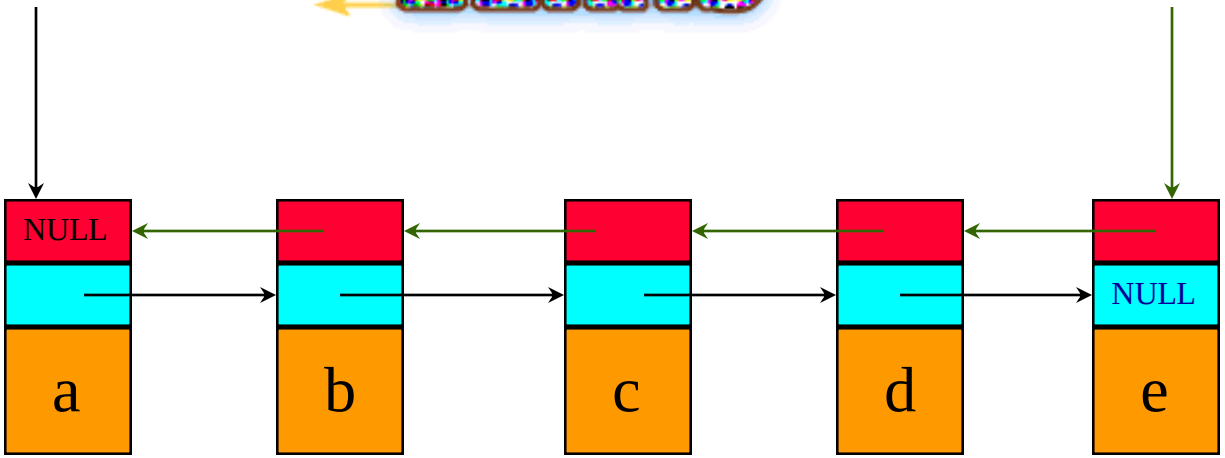
Circular List



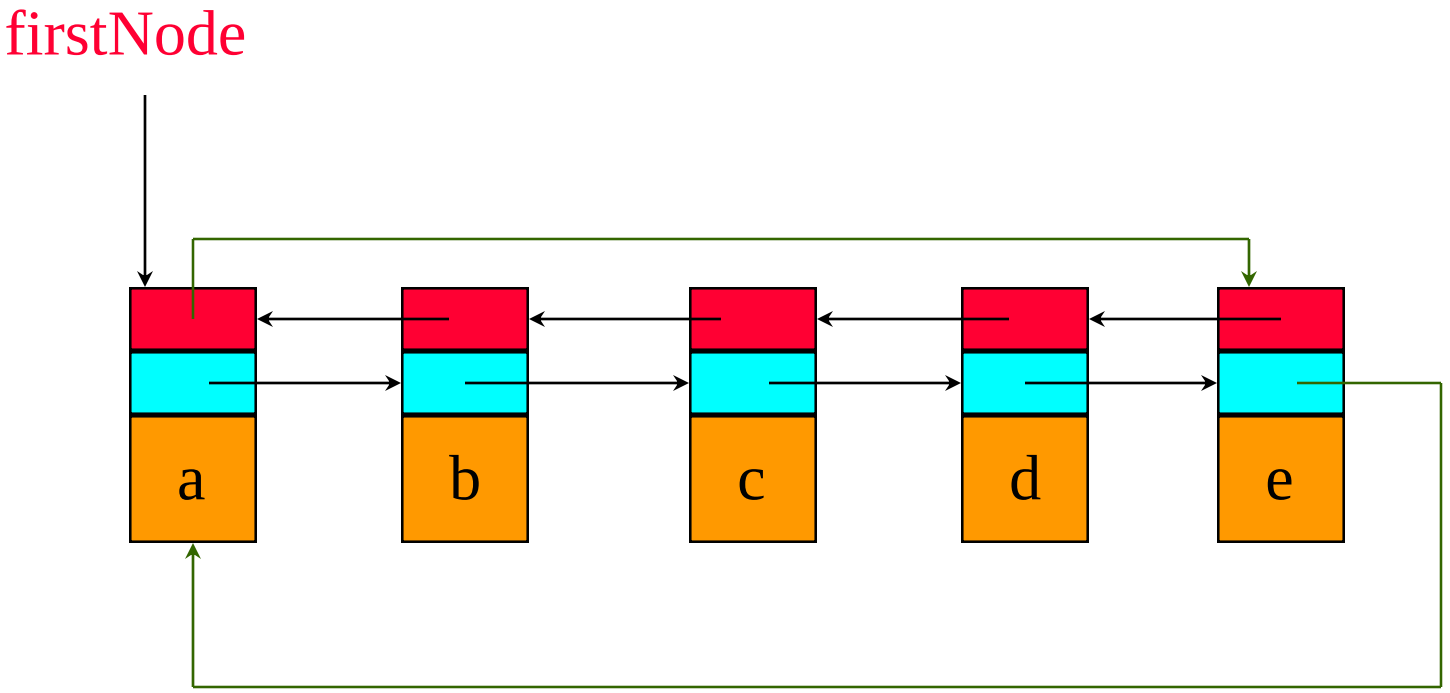
Doubly Linked List

firstNode

lastNode

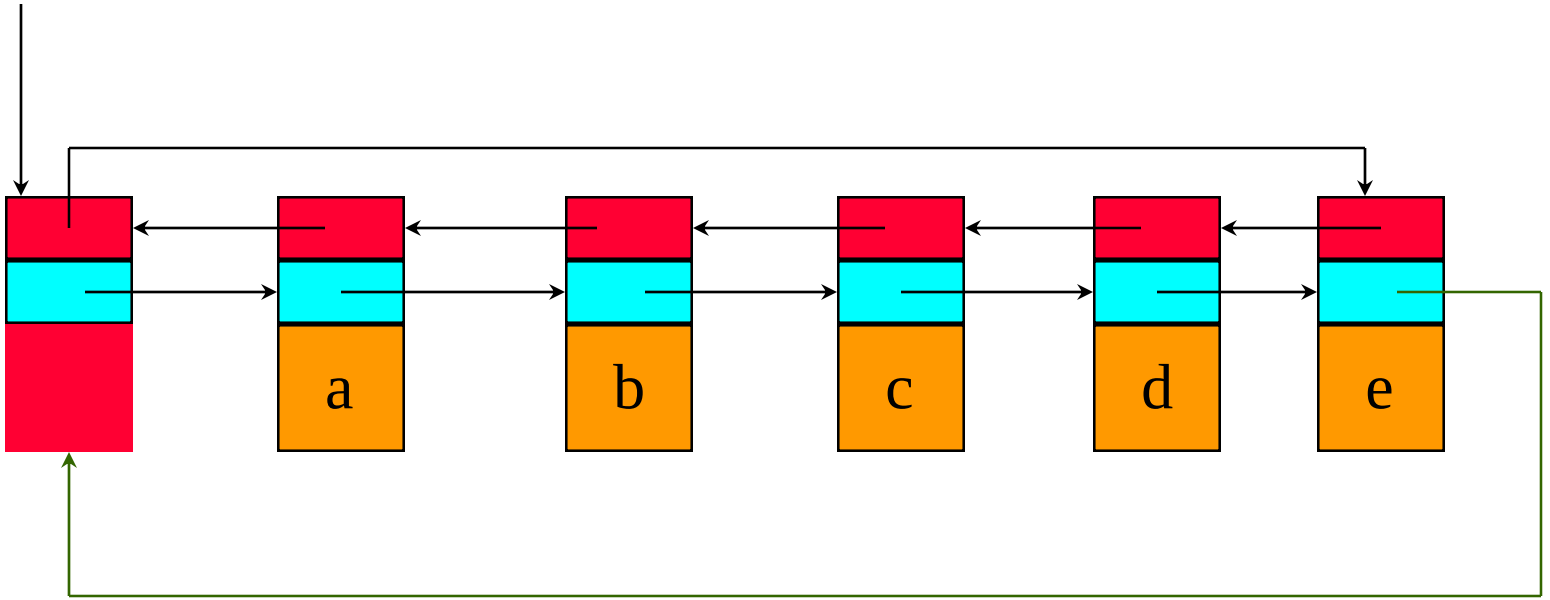


Doubly Linked Circular List

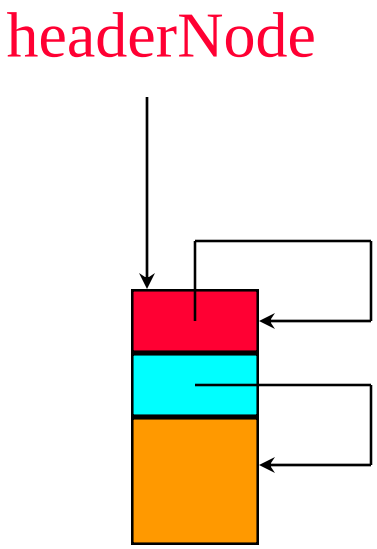


Doubly Linked Circular List With Header Node

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.