Chapter 4

Linked Lists

Why Linked Lists

- In the previous chapters
  - Arrays are used to represent successive items of the data object
    - Advantage: This provides easy access to any items in ordered list

Why Linked Lists

- „Disadvantage
  - Moving items during insertion and deletion may be costly.
    - E.g., inserting a new element into a[0], … , a[1000], say between the 500-th and 501-th, requires moving all elements after the 500-th one position rightward.

Why Linked Lists

- Possible solution to fix this

  - Linked list: inserting or removing an item requires only a marginal cost
    - Cost for iterating through the list may be ignored

  - The only items involved in the operation are before and after the item being inserted/removed.

Why Linked Lists

- More about arrays and linked lists:
- Merits of arrays (e.g., vectors in STL):
  - Using sequential memory
  - Faster for randomly accessing any *n*-th element
    - Accessing time is constant
    - I.e. takes the same amount of time regardless of the size of the array.

Why Linked Lists

- How about linked list:
  - If one wants to do random access in a linked list, he/she may have to go to the front of the list and iterate *n* times
    - Accessing time is linear in *n* (obviously slower).

Why Linked Lists

- However, merits of linked lists is:
  - Can be expanded in constant time with minimum memory overhead
    - only need to do a memory allocation on every insertion

Why Linked Lists

- If doing this by array, problem arises: E.g.:
  - When making an array, one must allocate memory for a certain number of elements.
  - If the original array is not big enough to accommodate new elements, then two solutions are:

Why Linked Lists

- (1) May create a new, but bigger array; then copy the old array into the new array, but
  - this may take lots of time.
- (2) Can allocate lots of space initially, but might allocate more than needed, so
  - this wastes memory.

Why Linked Lists

- Merits of linked lists (cont.):

  - More efficient in inserting/deleting than using arrays →
    - No need to move the elements currently in the list.
    - One can insert/delete at the front or the back, or in the middle.

Quick Summary

- Merits of arrays: efficient when accessing elements

- Merits of linked lists: efficient when inserting/deleting elements

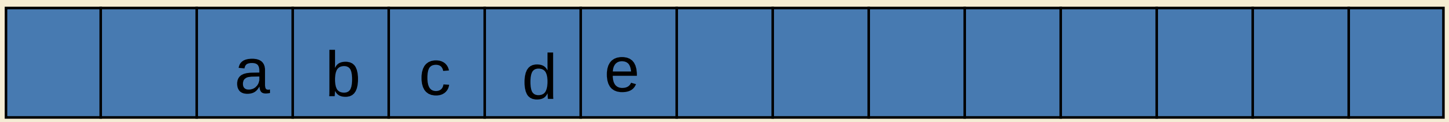
Linked Lists

- Elements in a list are stored, in memory, in an arbitrary order

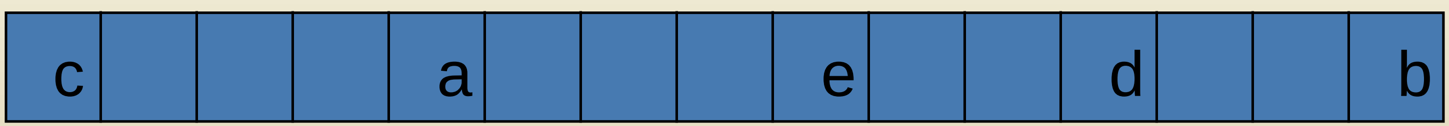
- Explicit information (called a link) is used to go from one element to the next

Comparision – Memory Layout

- Layout of L = (a,b,c,d,e) by arrays:

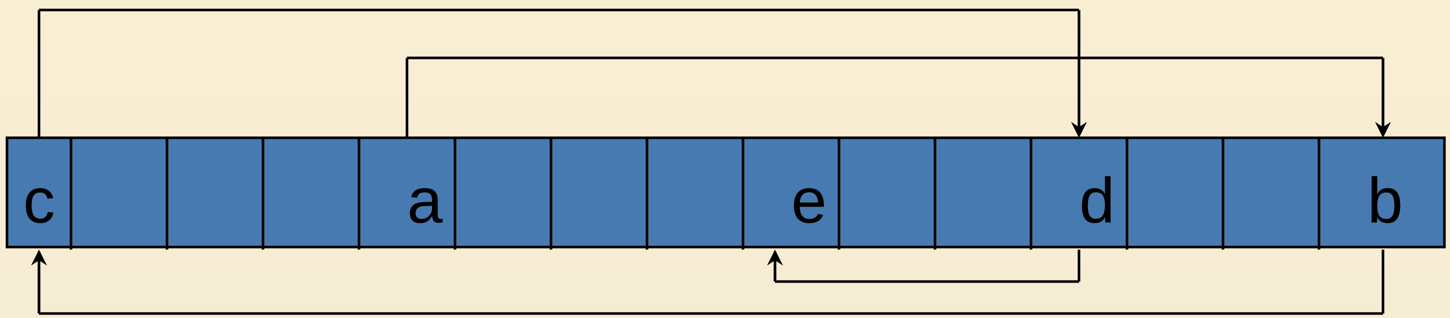


- A linked list uses an arbitrary layout:



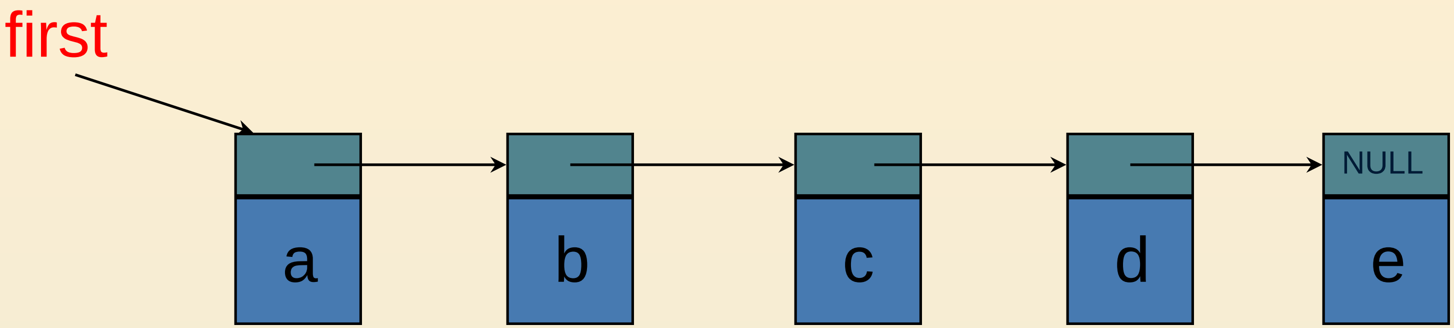
but adding more info to know the order of the elements (see next page).

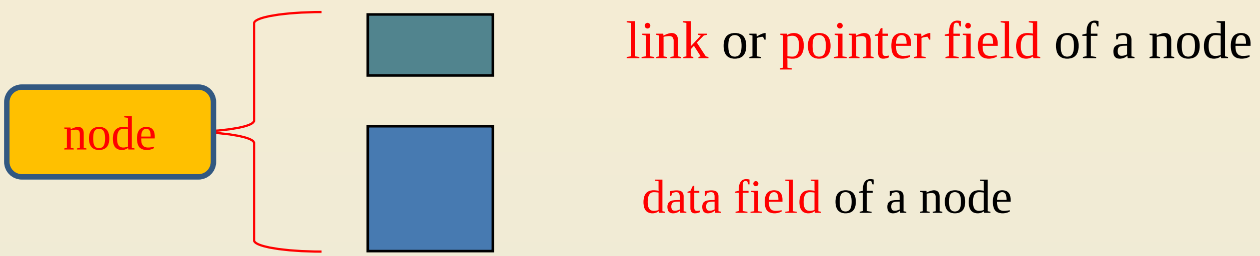
Linked Representation



- Implementations of linked lists:
  - Arrays (unusual, but sometimes are useful → won't talk about this due to time constraint)
  - Pointers (most common)

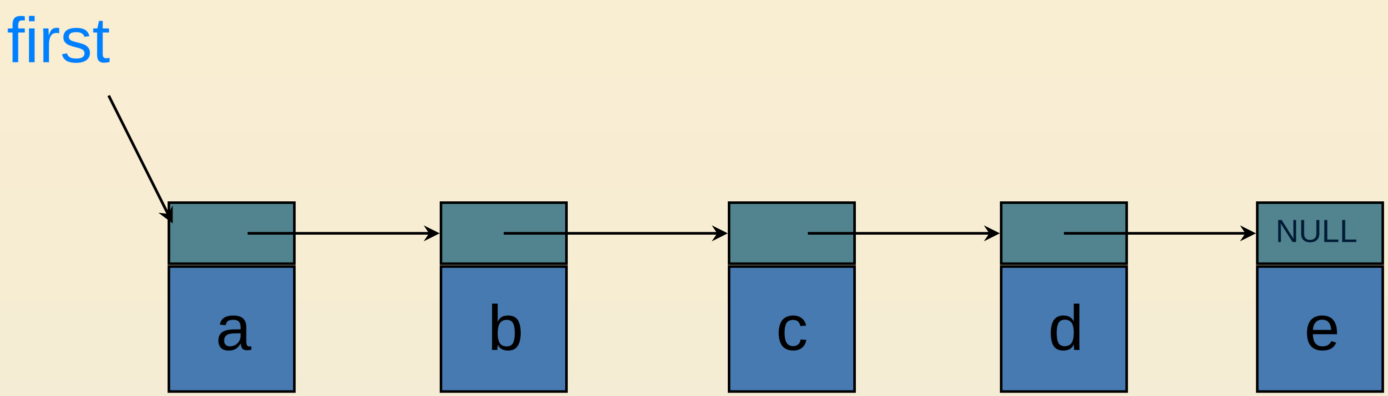
Pointer-based Linked List





- How to access this linked list:
  - Declaring a variable “first” to point to the first element a
  - “first” could be a node or simply a pointer
    - If “first” is simply a pointer, then it’s called a “head pointer” → containing no data, so it’s a dummy variable.

Chain

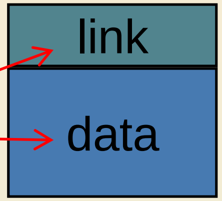


- Such a linked list is called a chain -- each node represents one element
- There is a pointer from one element to the next.
- The last node has a NULL (or 0) pointer.

Node Representation

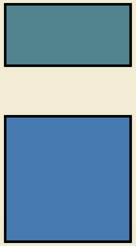
```
template <class T>
class ChainNode
{
  private:
   T data;
   ChainNode<T> *link;

   // constructors come here
};
```



Chain Node

- Use ChainNode to represent a node

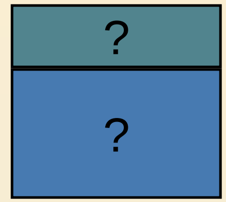


link (datatype is ChainNode<T>*)

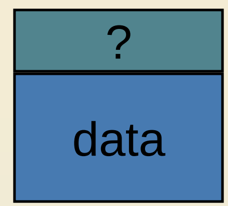
data (datatype is T)

Constructors of ChainNode

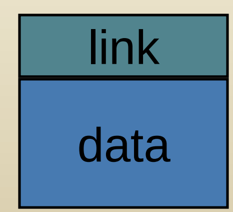
```
ChainNode() {}
```



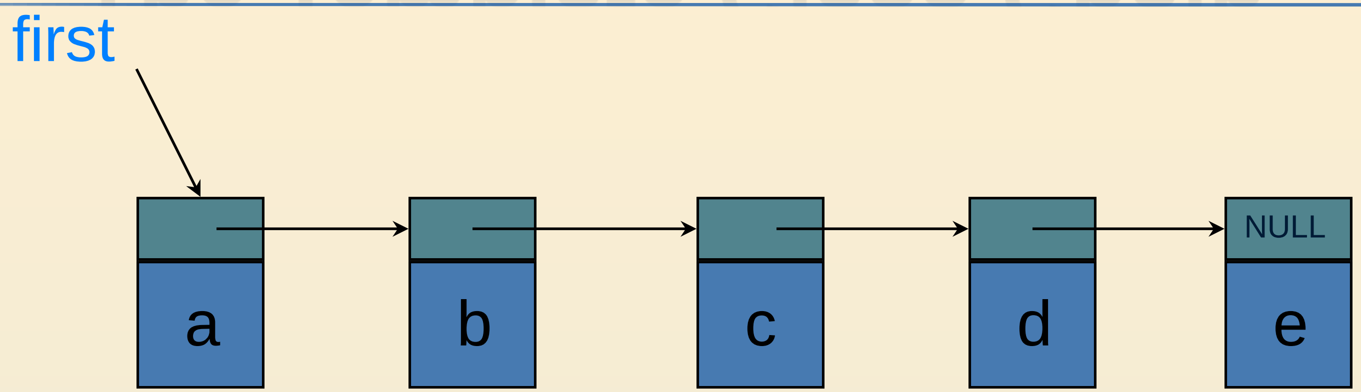

```
ChainNode(const T& data)
    {this->data = data;}
```




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



The Template Class Chain



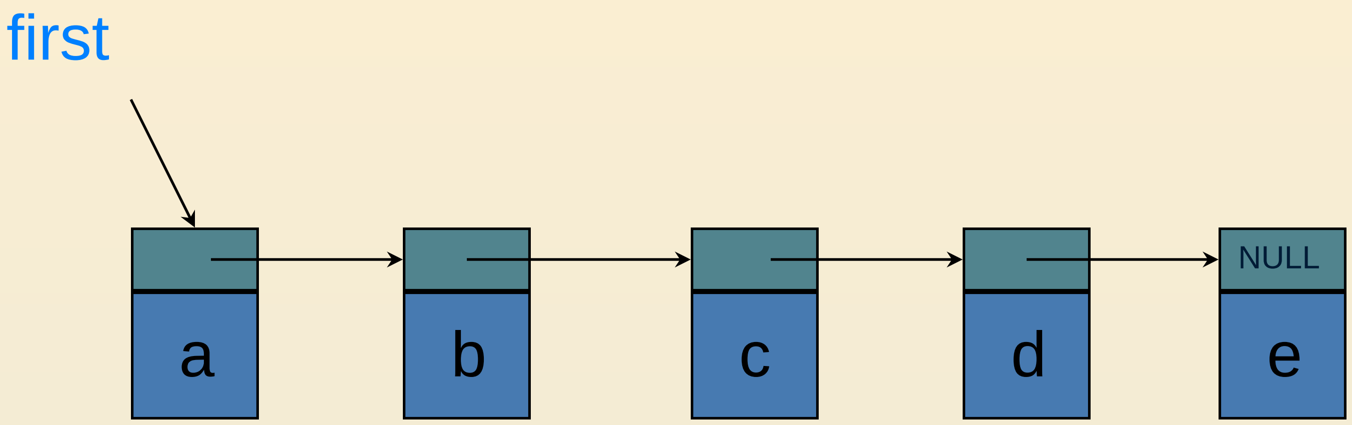

```
template<class T>
class Chain
{
   public:
      Chain() {first = Null;} // constructor, empty chain
      ~Chain(); // destructor
      bool IsEmpty() const {return first == Null;}
      // other methods defined here
   private:
      ChainNode<T>* first;
};
```

Traversal of Linked List

- Many operations in linked lists are done by performing a traversal of these data structures.

- During the visit of a node, operations (e.g., accessing/printing the node's data, evaluating the operator…) on this node may be taken.

Example of Visiting Nodes — Get(0)

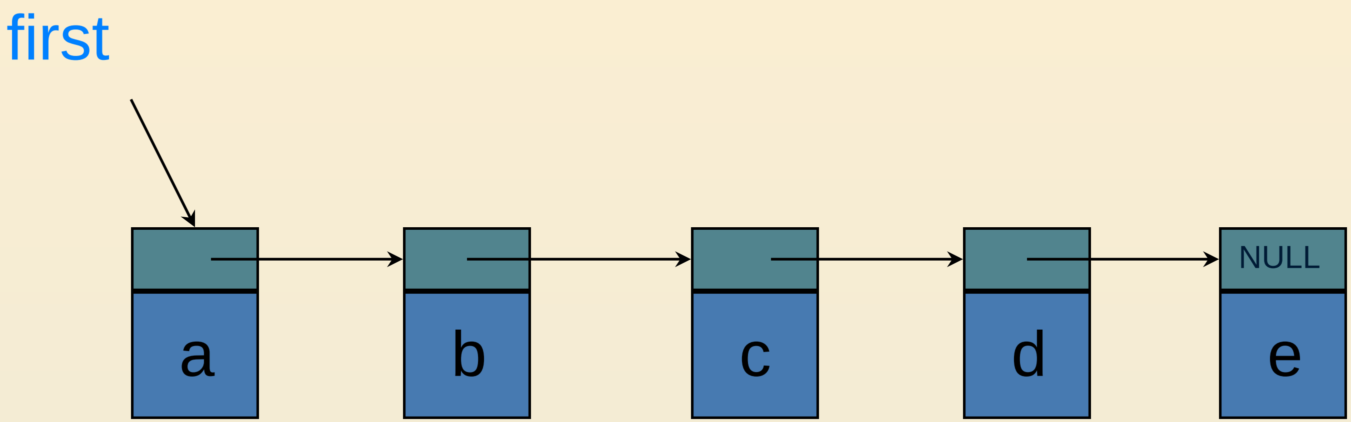


desiredNode = first; // gets you to the first node

return desiredNode–>data; //desired node is node **a** now

- Note that "first" is simply a pointer, not a node.
- If "first" is also a ChainNode, what would happen in the code above? Think about it!

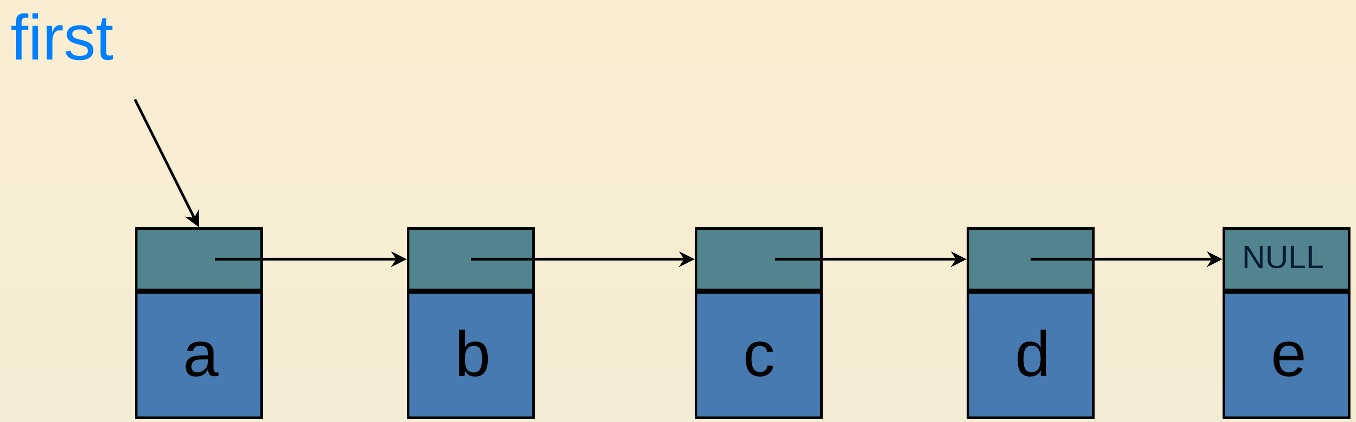
Get(1)



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

return desiredNode–>data; //desired node is node b now

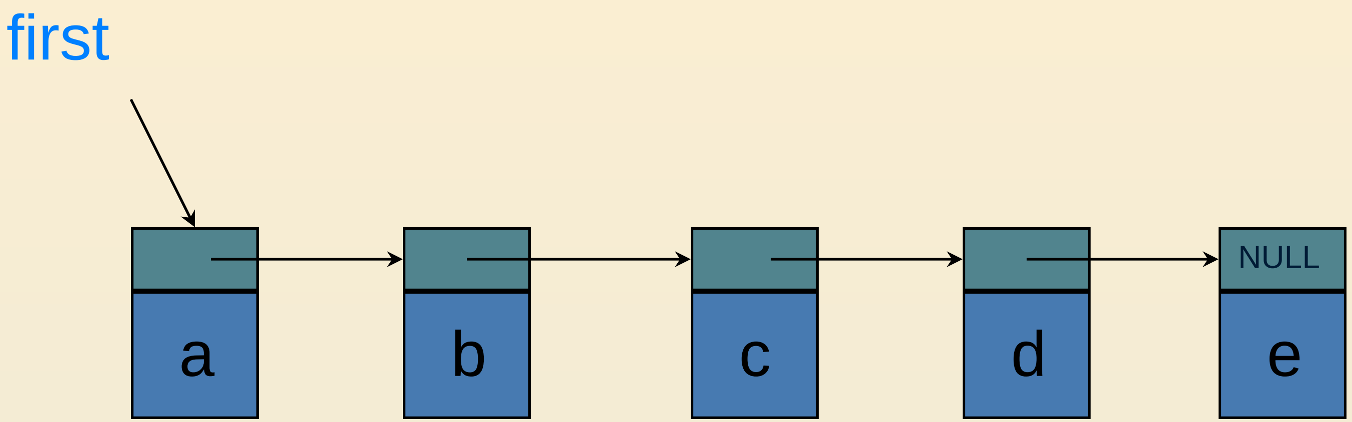
Get(2)



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

return desiredNode–>data; //desired node is node c now

Get(5)



desiredNode = first–>link–>link–>link–>link–>link;

// !!! Note: desiredNode = NULL

return desiredNode–>data; // NULL element. So what do you get?

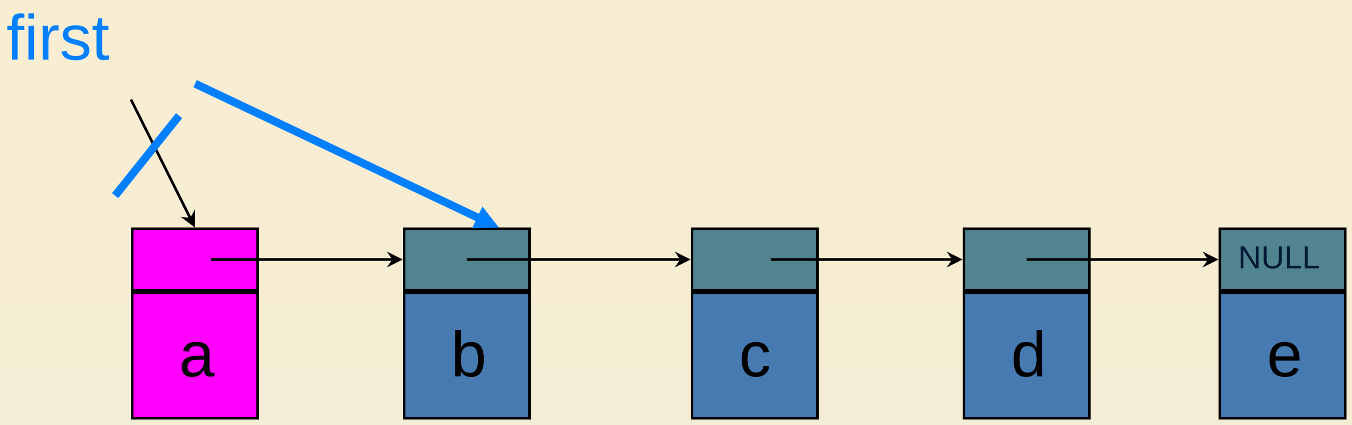
- You will see later how to trace the nodes more generally.

Deletion of Nodes

Two cases to consider:

- Delete the 1st node: Delete(0)
- Delete the non-1st node: Delete(n) and n>0

Delete(0)



- Goal: to delete the 1st node (node a)

deleteNode = first;

first = first–>link; //moves to node b

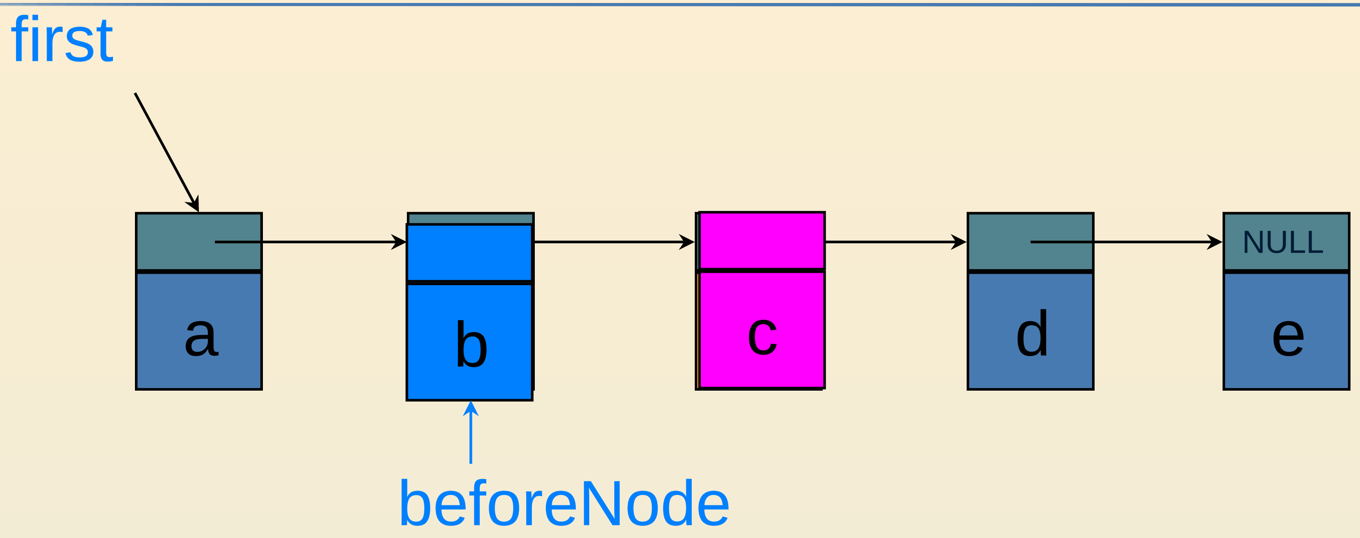
delete deleteNode;

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 deleteNode;
    }
```

For Delete(0) (Note: there is an "else" later for Delete(n), n>0).

Delete(2)



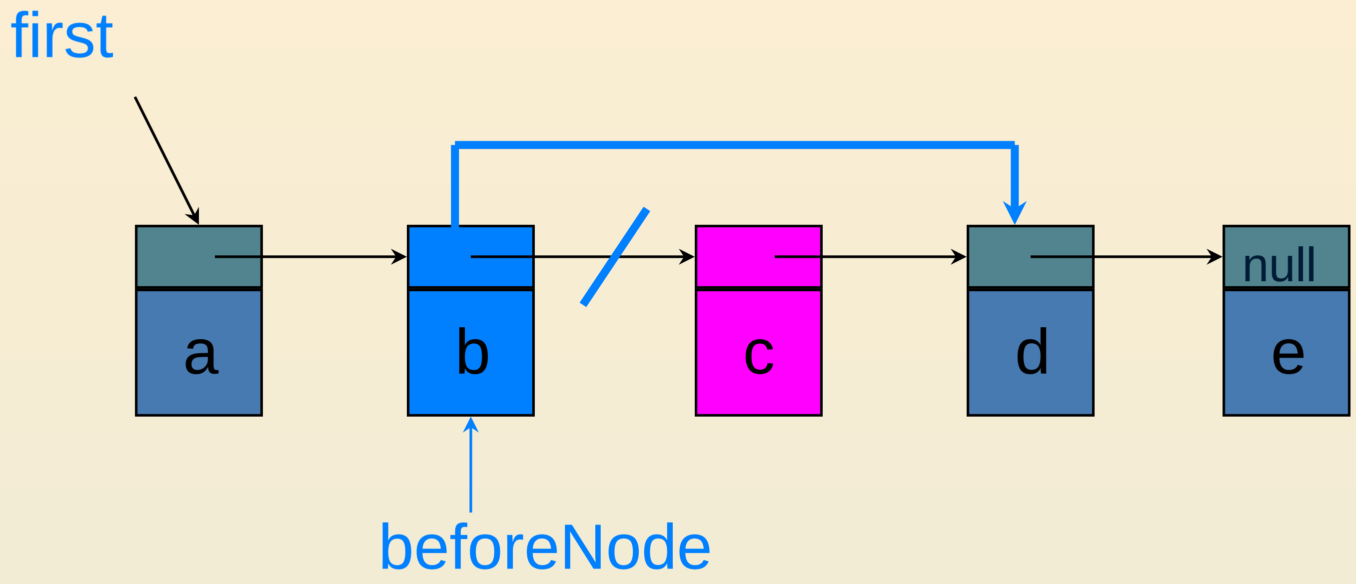
- To-be-deleted node is c
- Define a beforeNode (i.e., node b) to access the node right before c:

  beforeNode = first–>link;

- To delete node c, get beforeNode's pointer to c:

  deleteNode = beforeNode–>link;

Delete(2) (cont.)

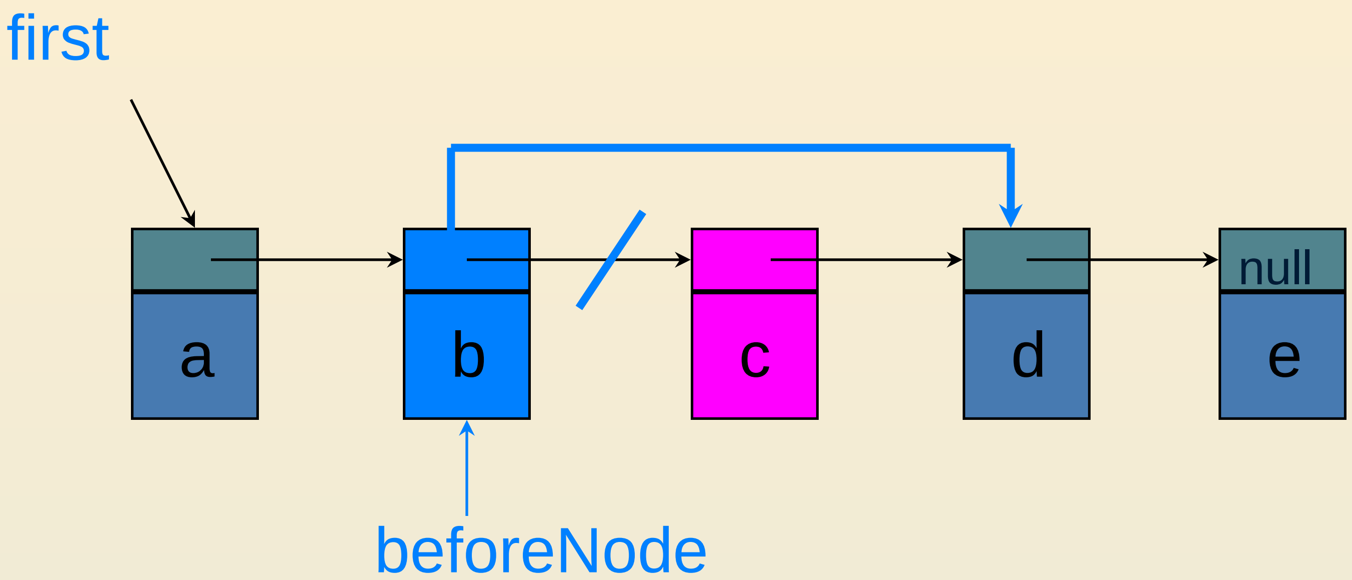


- Next update the pointer in beforeNode to node d.

beforeNode–>link = beforeNode–>link–>link;

delete deleteNode;

Delete(2) (cont.)



In summary,

beforeNode = first–>link;

deleteNode = beforeNode–>link;

beforeNode–>link = beforeNode–>link–>link;

delete deleteNode;

Delete an Element

```
    else
    { // use p as beforeNode
        ChainNode<T>* p = first;
        for (int i = 0; i < theIndex - 1; i++)
        {
            p = p->link;
        }
        deleteNode = p->link;
        p->link = p->link->link;
    }
    delete deleteNode;
}
```

The Destructor

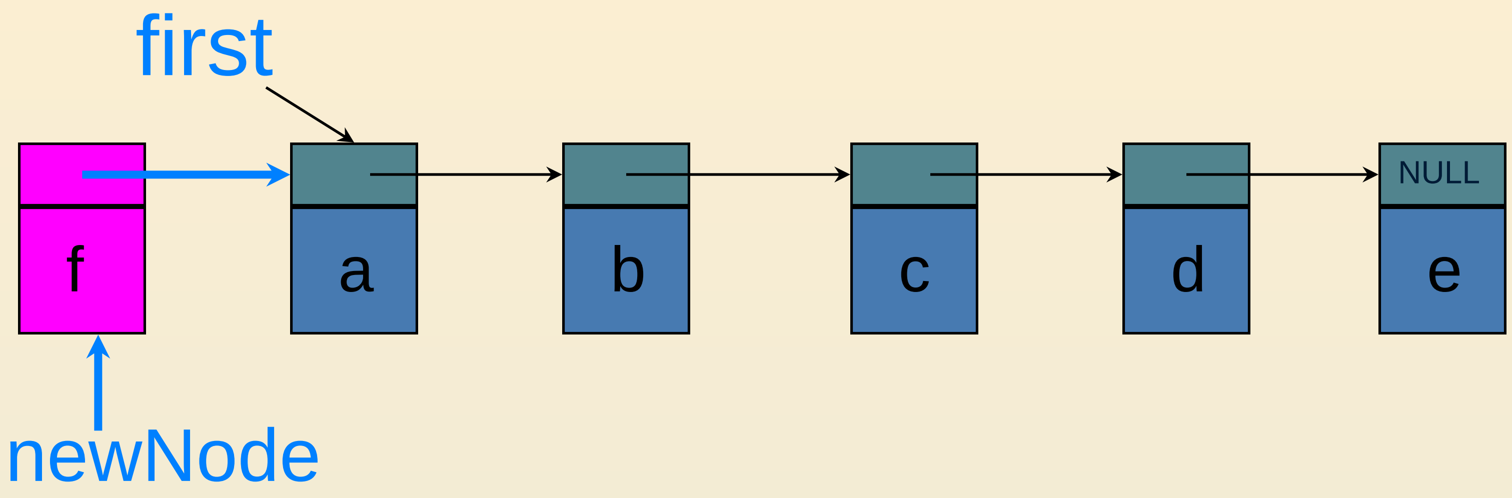
```
template<class T>
chain<T>::~chain()
{// Chain destructor. Delete all nodes
 // in chain.
   while (first != NULL)
   {// delete first;
      ChainNode<T>* next = first->link;
      delete first;
      first = next;
   }
}
```

Insertion of Nodes

Two cases to consider:

- Insert at the front of the list
- Insert in the middle of the list

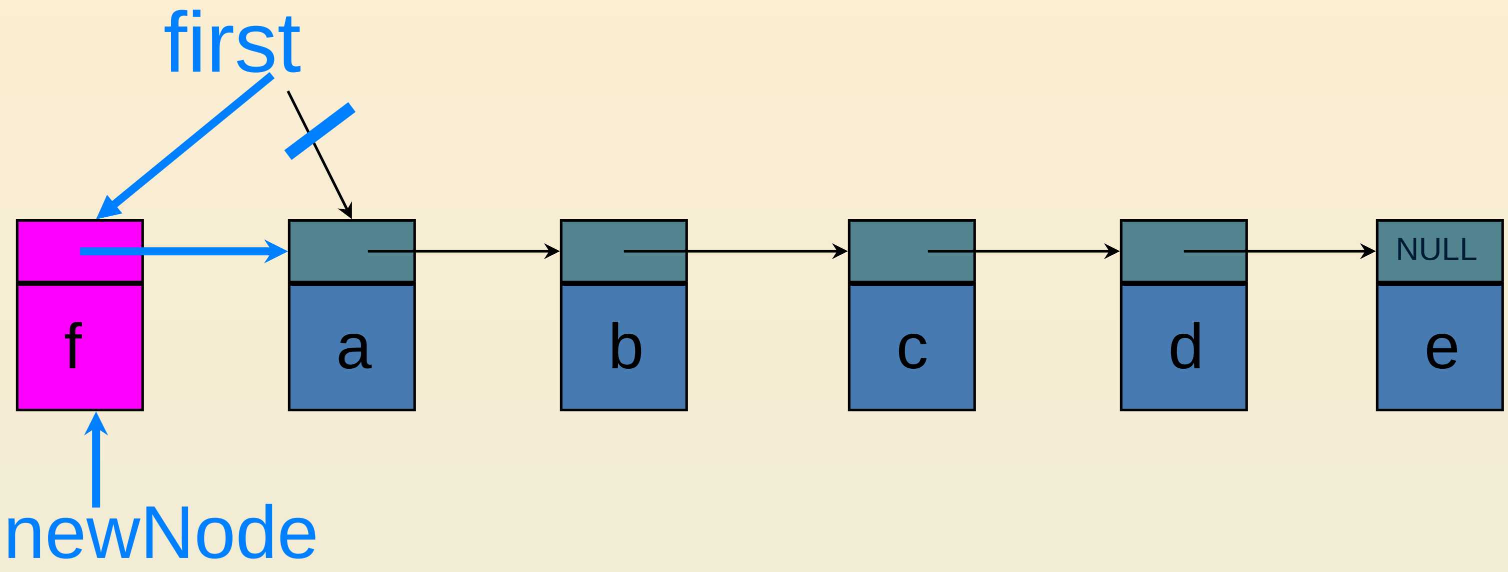
Insert(0,'f')



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

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

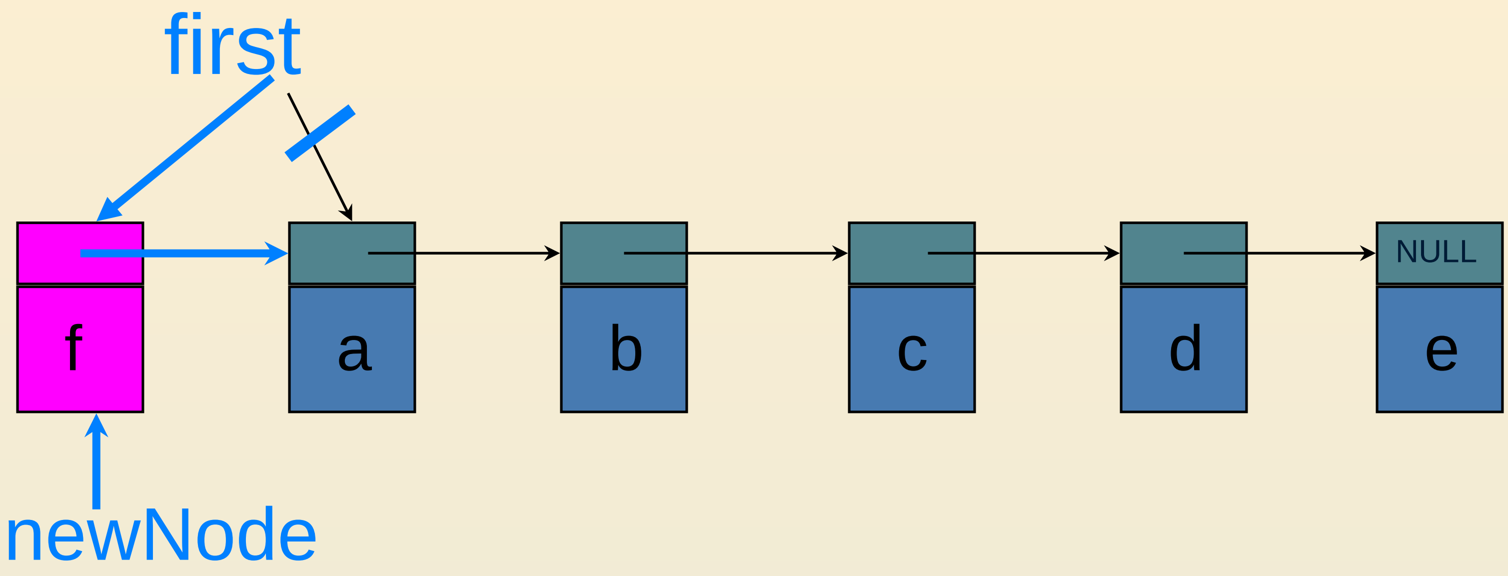
Insert(0,’f’) (cont.)



Step 2: update first

first = newNode;

One-step Insert(0,'f')



In summary,

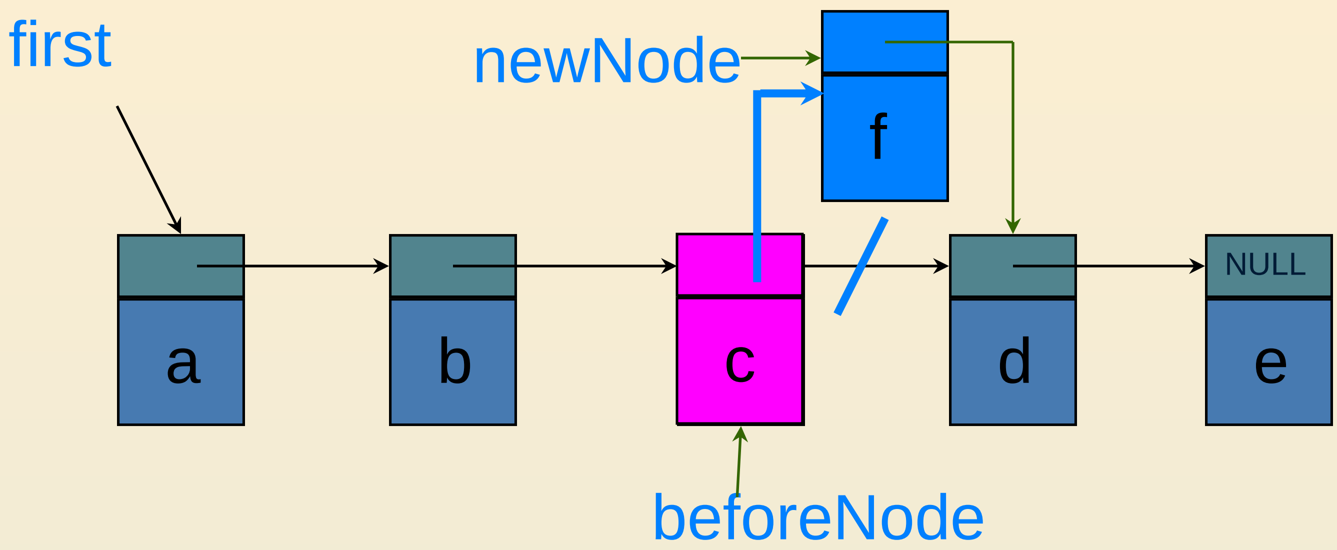
first = new chainNode<char>('f', first);

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);
```

Insert(3,'f')



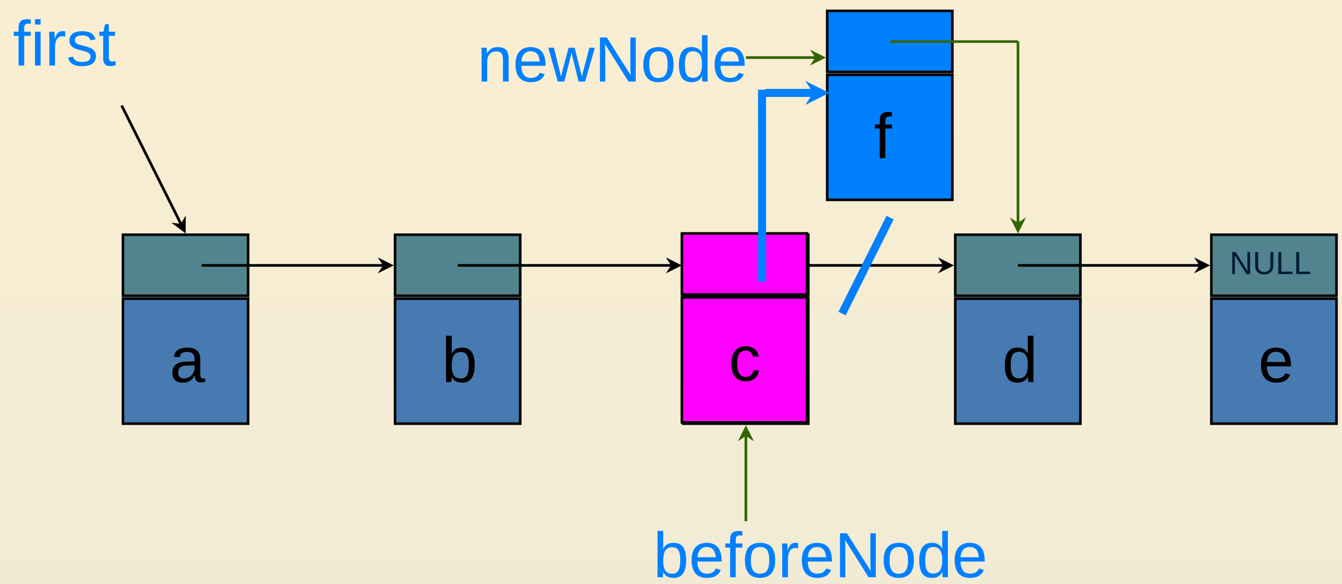
- First find the node with index 2 ➔ beforeNode
- Next create a new node linking to node d (index 3)

```
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);

Inserting an Element

```
else
{  // use p as beforeNode
   ChainNode<T>* p = first;
   for (int i = 0; i < theIndex - 1; i++)
   {
      p = p->link;
   }
   // insert after p
   p->link = new ChainNode<T>
                    (theElement, p->link);
}
}
```

Iterators

- Iterator: an object permits you to traverse all the elements of a container class (such as list class).

- C++ iterators, e.g.
  - 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 ->
- May be advanced using the increment 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 ->
- May be advanced using 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);
```

Be careful: there is no “int i=0” here because the iterator objects are not like simple variables

vs

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

- Assume Size returns number of elements in the chain and Get returns the *i*-th element.
- Since the code by iterators looks more sophisticated, then why still need it? See next.

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
- So "Iterators" provide a uniform way to sequence through the elements of a data structure

A Forward Iterator for Chain

```
class ChainIterator {
public:
    // The constructor comes here; (see next page)
    // operations, e.g., dereferencing operators * &
    // ->, ++, -- , etc. come here
private:
  ChainNode<T> *current;
```

Constructor

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

Increment

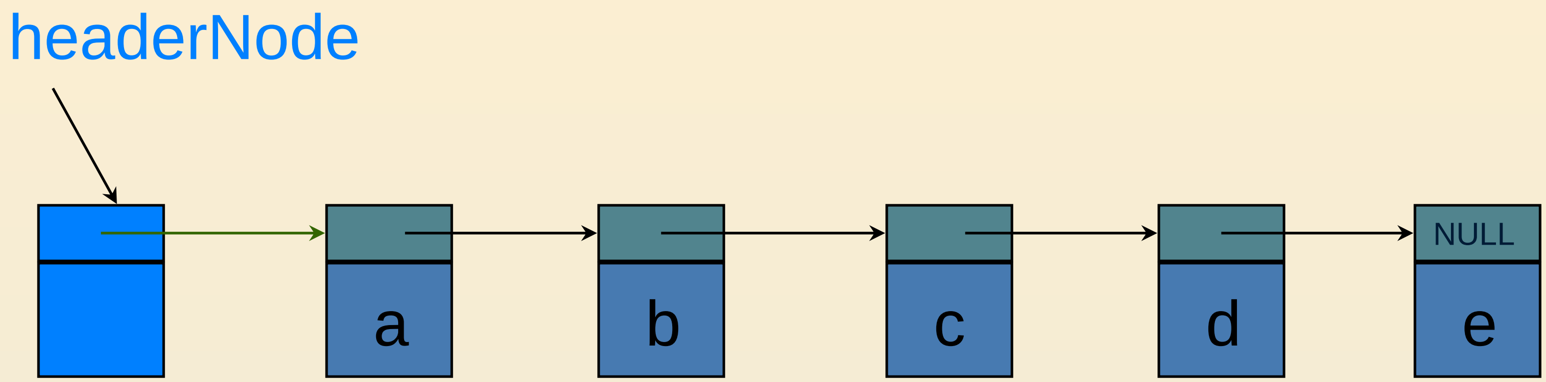
```
ChainIterator& operator++() // preincrement
  {current = current->link;
   return *this;}
```

Move to the next element first; then retreive the element

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

Retrieve the element first; then move to the next element

Chain With Header Node

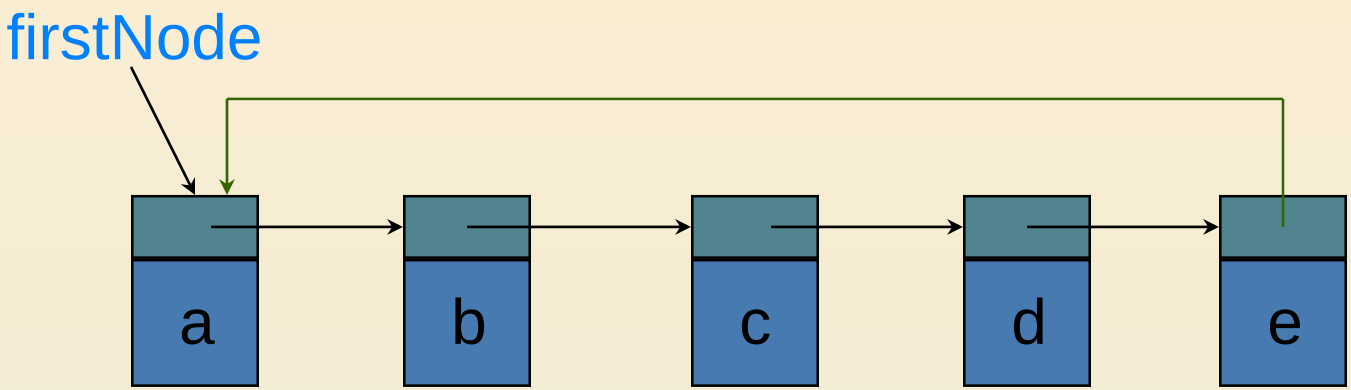


- Header nodes are like head pointers:
  - are used to get to a list

- Now insert/erase at left end (i.e., index = 0) are no different from any other insert/delete. So insert/delete code is simplified

- This is the difference between head pointers and head nodes.

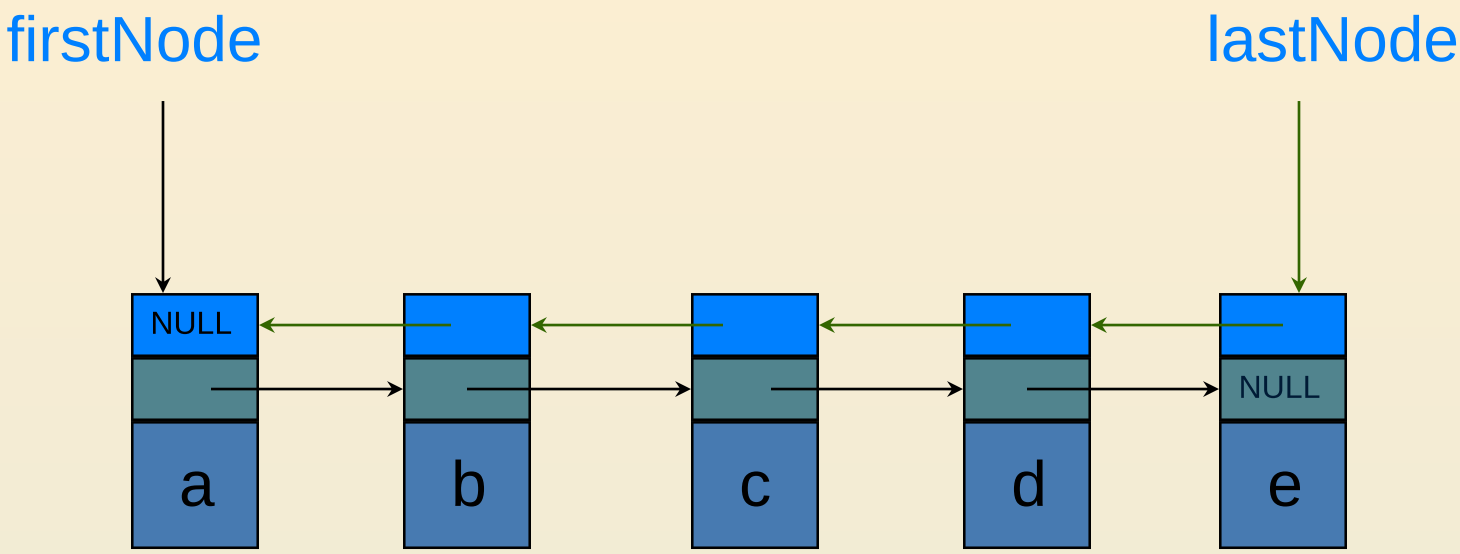
Circular List



- Natural option to represent arrays that are naturally circular (e.g., circular queues)

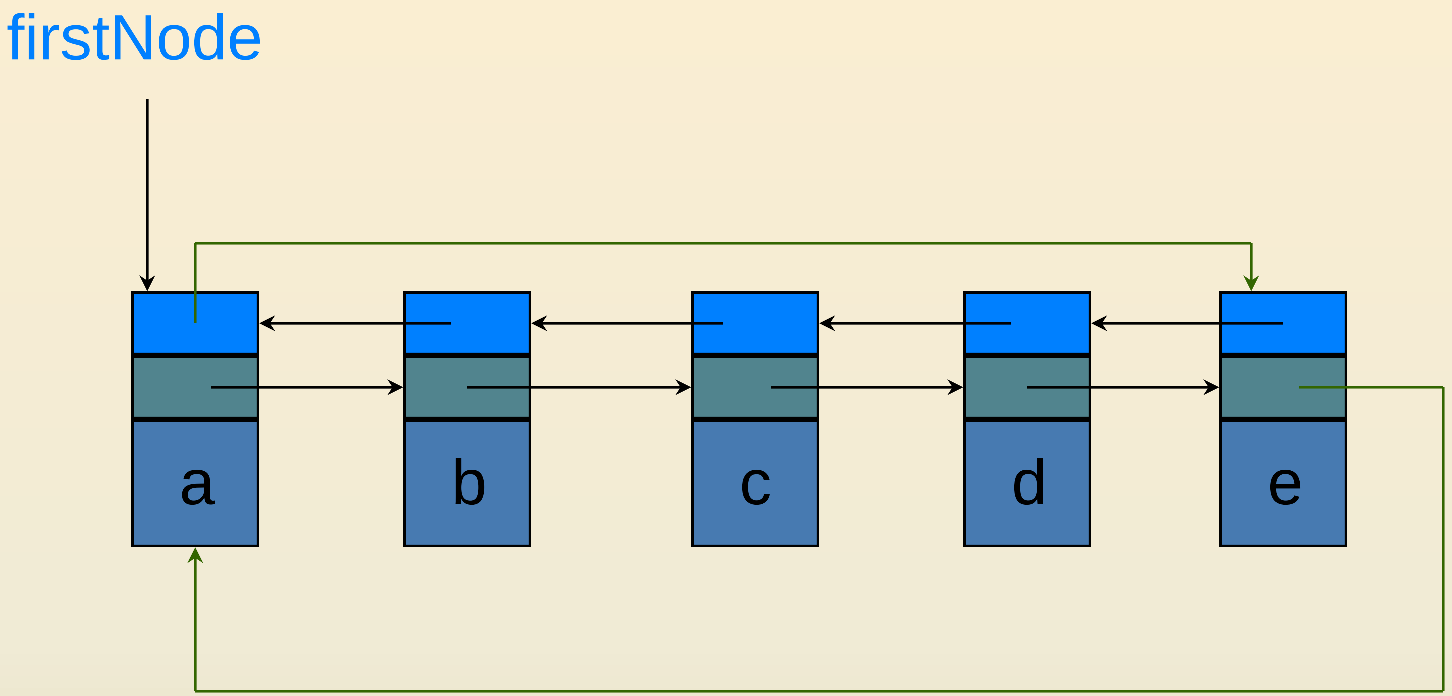
- Can also be applied to sparse matrix representation. Talk about this later.

Doubly Linked List

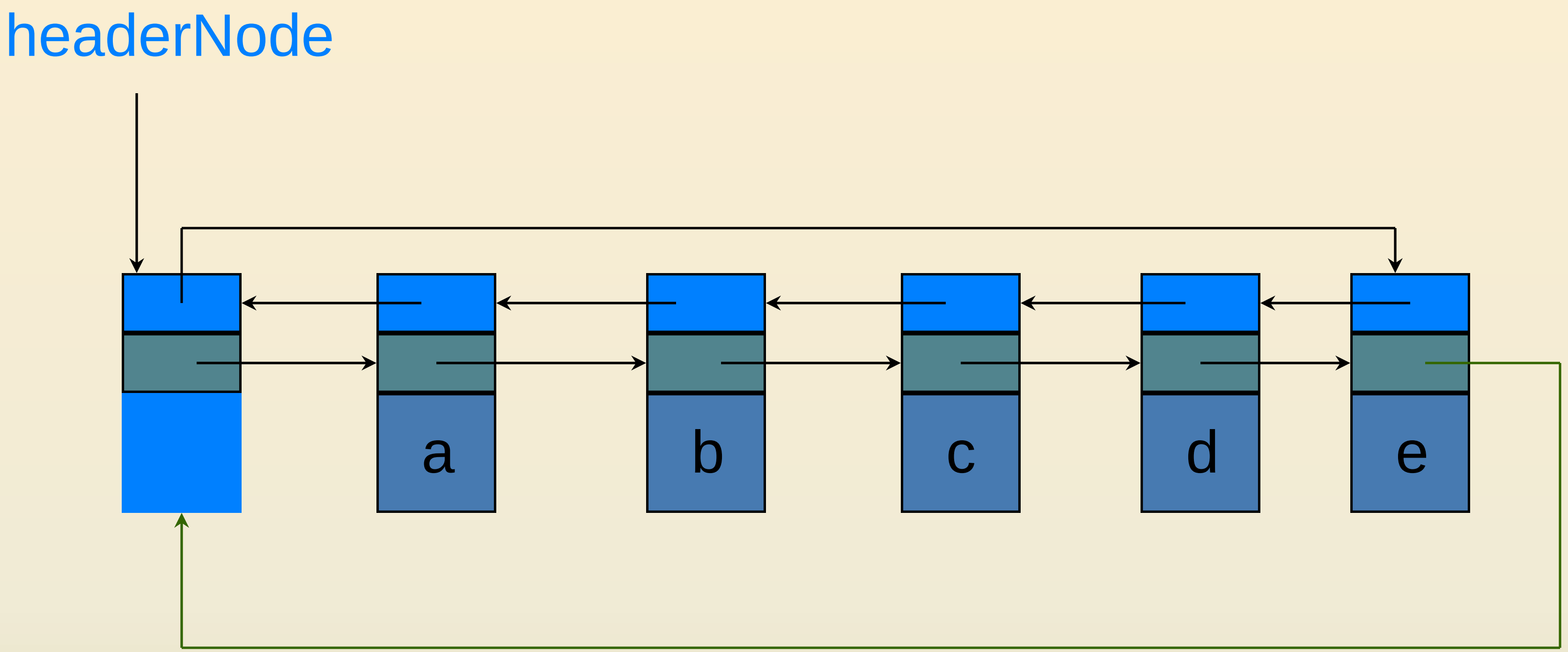


- Reduce worst-case run time for getting a particular node of a linked list by half
  - If index <= listSize/2, start at left end; otherwise, start at right end

Doubly Linked Circular List



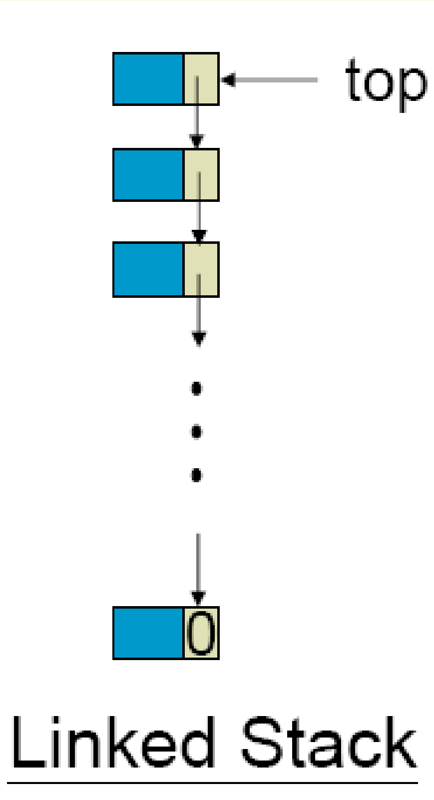
Doubly Linked Circular List With Header Node

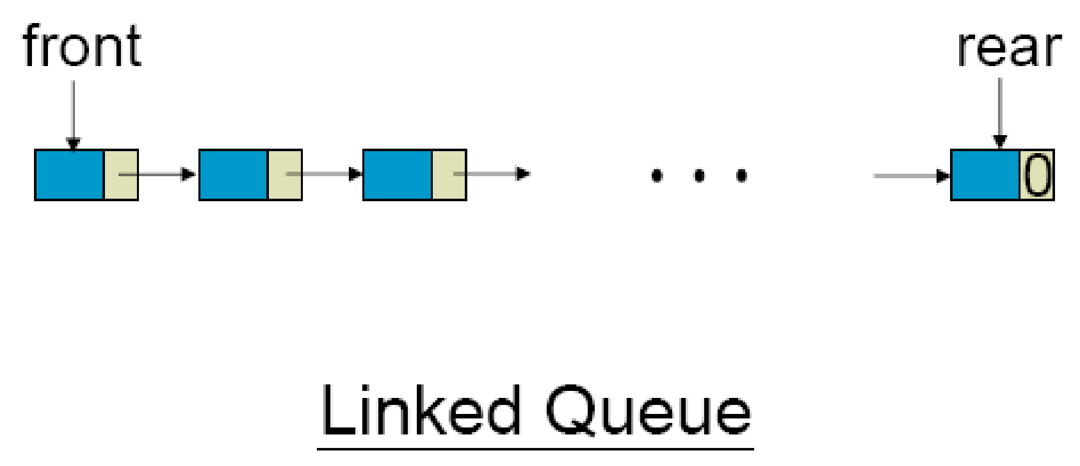


The STL Class list

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

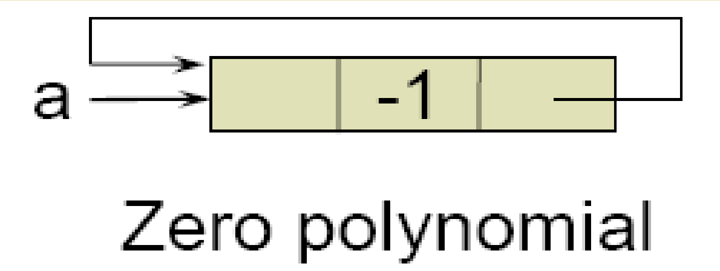
Linked Stacks and Queues



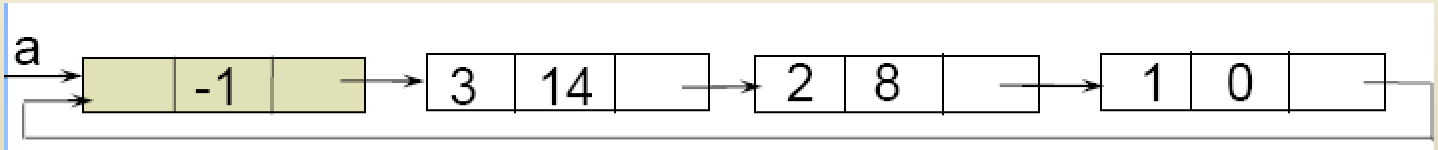


Polynomial with Head Node

- Represent a polynomial as circular list
  - Zero



  - Others



$$a = 3x^{14} + 2x^{8} + 1$$

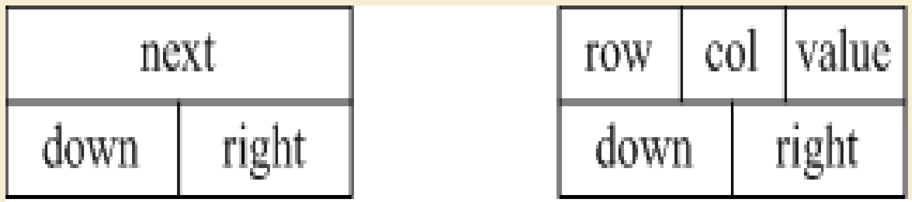
Revisit Sparse Matrices

- Inadequancy of sequential schemes
  - # of nonzero terms will vary after some matrix computation
  - Matrix just represents intermediate results

- New scheme
  - Each column (row): a circular linked list with a head node

Revisit Sparse Matrices

- Node representation:



Header node　　　Element node

- Each node also has a Boolean field “head” (but not shown here to avoid confusion on p.64’s figure of this PPT) to represent
  - header node
- # of header nodes = max{# of rows, # of columns}

Linked Representation of a Sparse Matrix

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$

- For an n*m sparse matrix with *r* nonzero elements, # of nodes needed = max{n, m} + r + 1 where
  - max{n, m}: # header nodes
  - r: # non-zero elements
  - 1: the head node for the whole matrix

Linked Representation of a Sparse Matrix

Matrix head

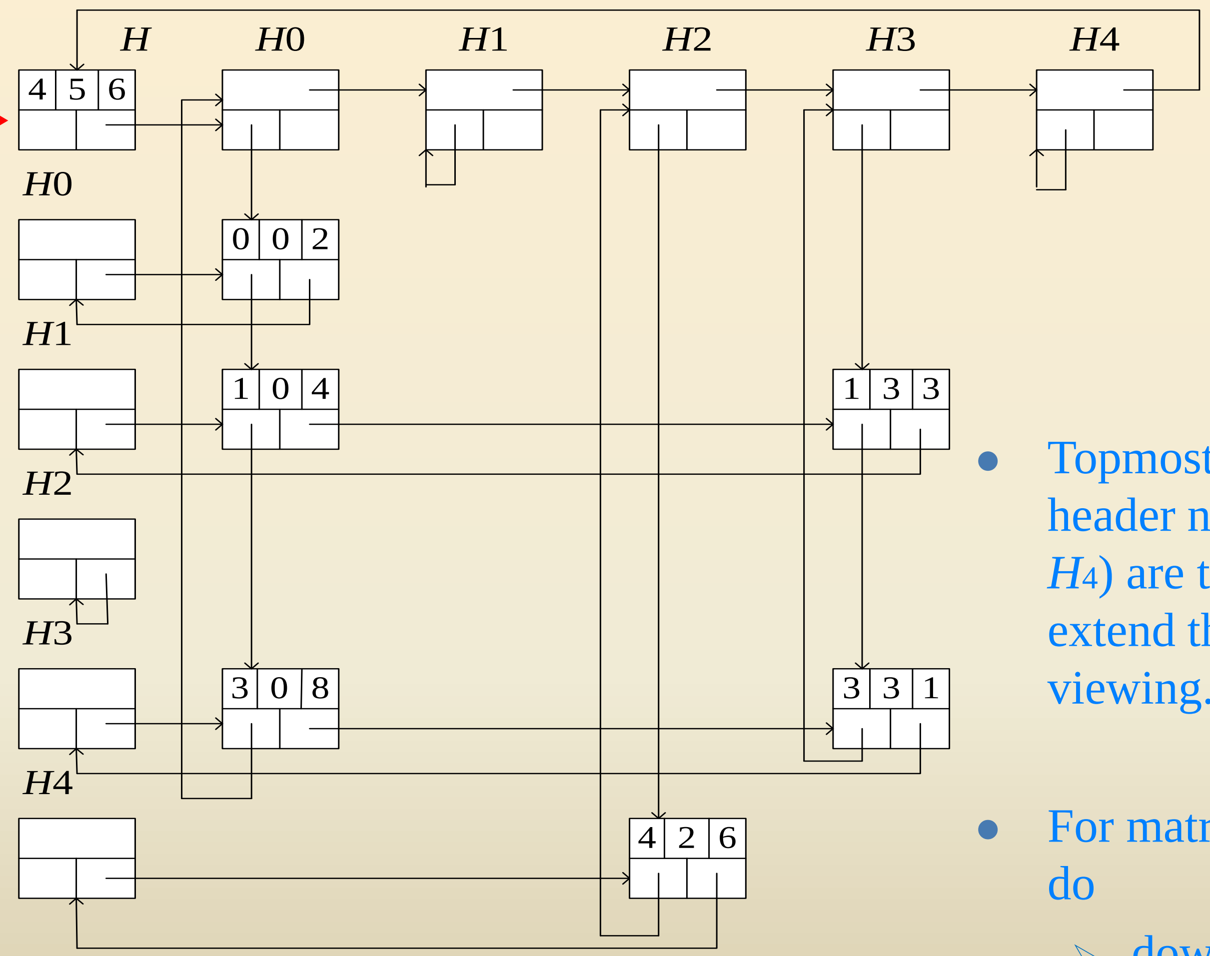
(Note:

the textbook didn't mention this node, but

4: # cols;

5: # rows;

6: # non-zero elements)



- Topmost and leftmost header nodes (H_0, … H_4) are the same. Just extend them for easy viewing.
- For matrix transpose, do
  - down → right
  - right → down

Difference between Linked-list and Array-based Representation

- Difference between (a) the linked-list and (b) 3-tuple array-based representation (in Chap 2):
  - Easier to use representation (b) to directly access the column data of the matrix.
  - By contrast, in representation (a), one has to dig into each row first, then access each column's data.
  - But # non-zeros elements may vary, (e.g., after matrix multiplication), so linked list is better in this case.

Homework

- Available on E-learning.
- Pay attention to the due day.