EECS 204002
Data Structures 資料結構
Prof. REN-SONG TSAY 蔡仁松 教授
NTHU

CH. 4
LINKED LISTS

Array Reviews

- Store an ordered list using **sequential** mapping
  - Element(node) a_i is stored in the location L_i of the array
  - Next node is at the location L_i+1
- Pros:
  - Suitable for **random access**
  - **Efficient** to insert/delete from the end
  - Adequate for special data structures, **Stack** and **Queue**.
- Cons:
  - **Difficult** to insert/delete nodes at **arbitrary** location

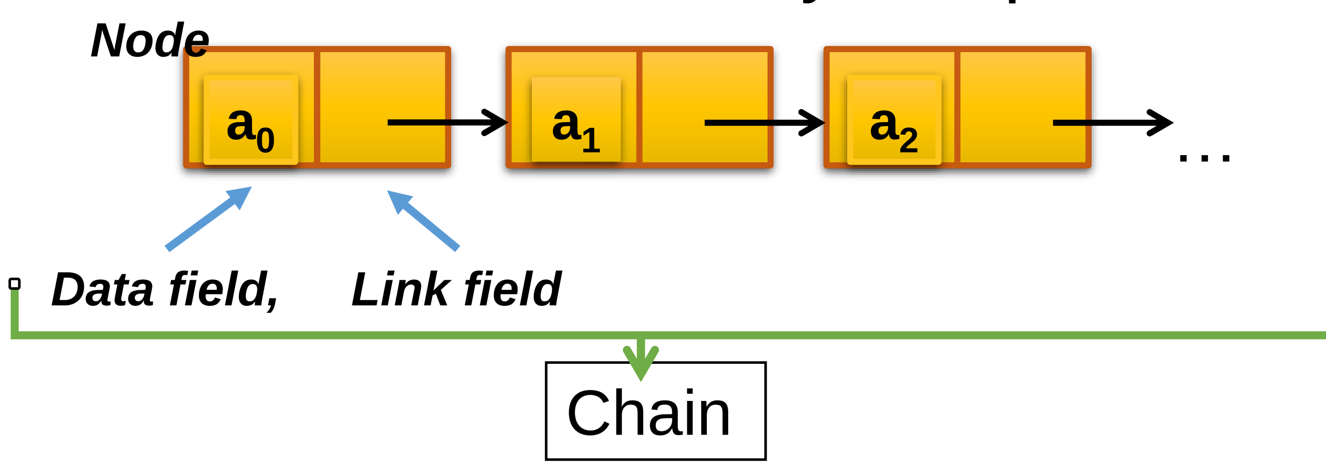
4.1

Singly Linked Lists and Chains

4.1

Linked Representation

- Nodes are **no longer placed continuously** in the memory space
- Each node stores the **address or location** of the next one
- Singly Linked List (SLL)
  - each node has exactly one pointer field

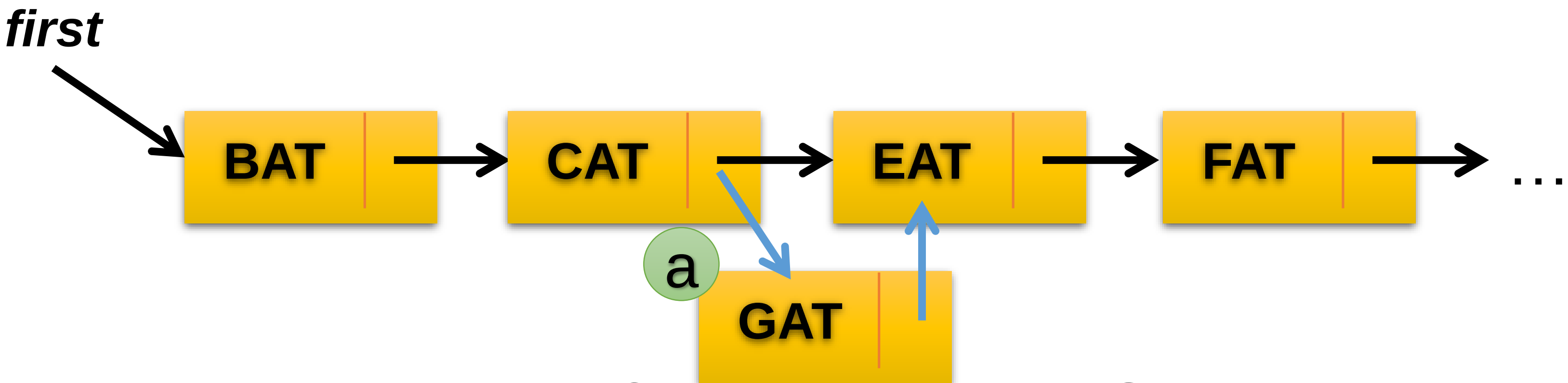


4.2

Representing Chains in C++

4.2

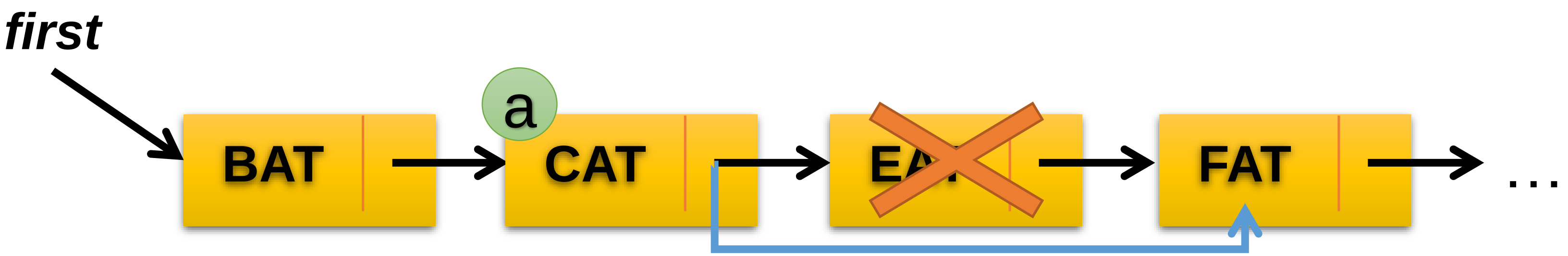
SLL Operation: Insert



- Procedures for inserting ”GAT” in between “CAT” and “EAT” nodes
  - Create a new node “a” and set data field to “GAT”
  - Set the link field of “a” to “EAT” node
  - Set the link field of “CAT” node to “a”

You do not need to move or shift any node!

SLL Operation: Delete

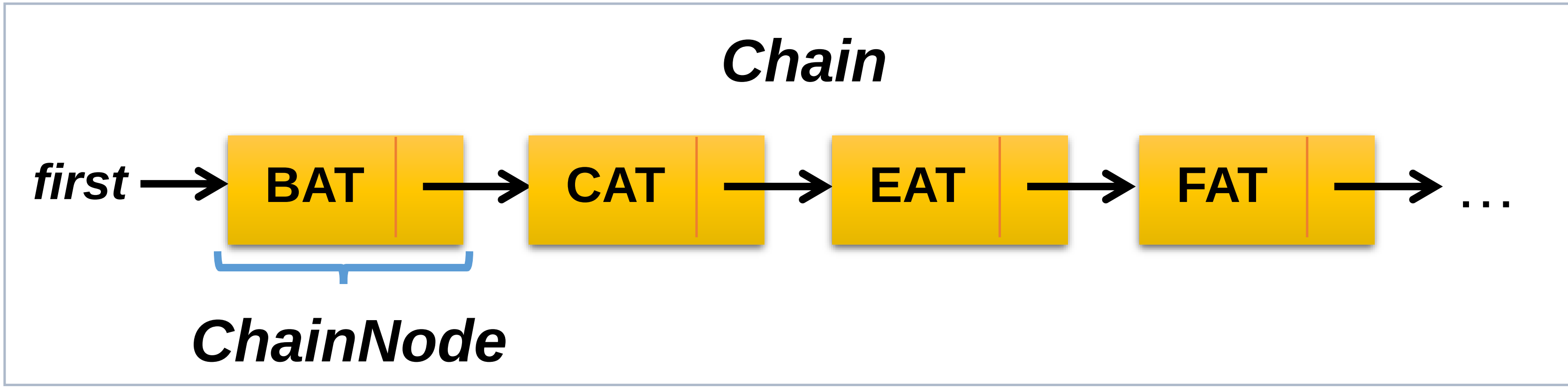


- Steps to do when we want to delete the "EAT" node from the list
  - Locate the node "a" precedes the "EAT" node
  - Set the link field of "a" to node next to "EAT" node
  - Delete the "EAT" node

You do not need to move or shift any node!

Conceptual Design

- Defining a "ChainNode" class
  - Data field
  - Link field
- Designing a "Chain" class
  - Support various operation on ChainNodes



4.2.1 ChainNode & Chain Classes

- Composite class

```
class ChainNode
{
friend class Chain;
public:
 // Constructor
 ChainNode(int
value=0, ChainNode*
next=NULL){
   data = value;
   link = next;
   }
private:
   int data;
   ChainNode *link;
};
```

```
class Chain
{
public:
     // Create a chain with two nodes
     void Create2();

     // Insert a node with data=50
     void Insert50(ChainNode *x);

     // Delete a node
     void Delete(ChainNode *x, ChainNode *y);

private:
     ChainNode *first;
};
```

ChainNode & Chain Classes

- Nested class

```
class Chain
{
public:
     // Create a chain with two nodes
     void Create2();

     // Insert a node with data=50
     void Insert50(ChainNode *x);

     // Delete a node
     void Delete(ChainNode *x, ChainNode *y);

private:
      class ChainNode{
        public:
          int data;
          ChainNode *link;
     }
     ChainNode *first;
};
```

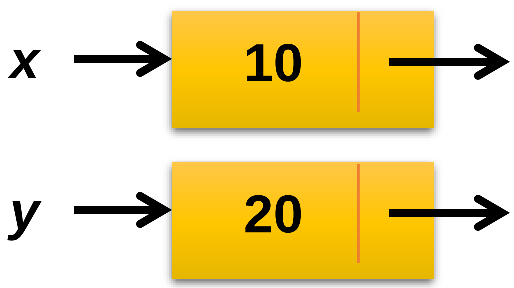
4.2.3

Review Pointer Manipulation

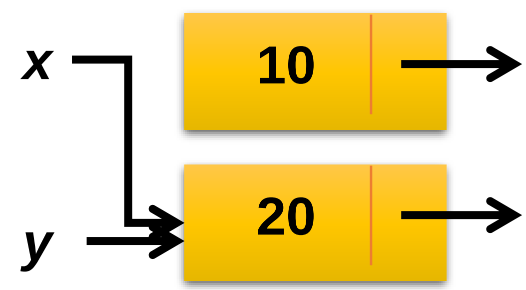
- **Declaration**
  - NodeA *a1=NULL, *a2=NULL;
- **Allocate memory**
  - a1 = new NodeA;
  - a2 = new NodeA[10];
- **Delete memory**
  - delete a1; a1=NULL;
  - delete [] a2; a2=NULL;
- **Dereference**
  - NodeA &a1Ref = (*a1);
- **Access members**
  - a1->memData;
  - a1->memFunc();
  - (*a1).memData;
  - (*a1).memFunc();

Pointer Assignment

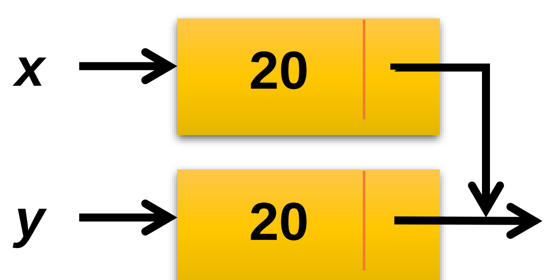
- ChainNode *x, *y;



- x = y;



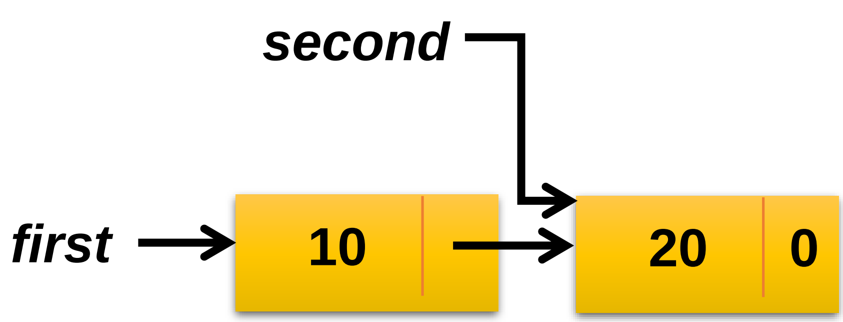
- *x = *y;



4.2.4

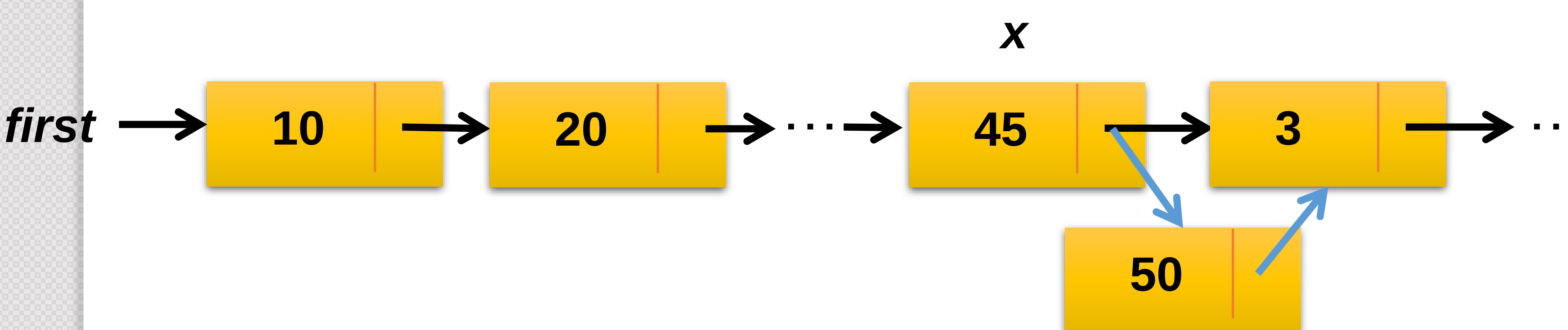
Chain Operations

```
void Chain::Create2()
{
    // Create and set the fields of 2nd node
    ChainNode* second = new ChainNode(20,0);

    // Create and set the fields of 1st node
    first = new ChainNode(10,second);
}
```



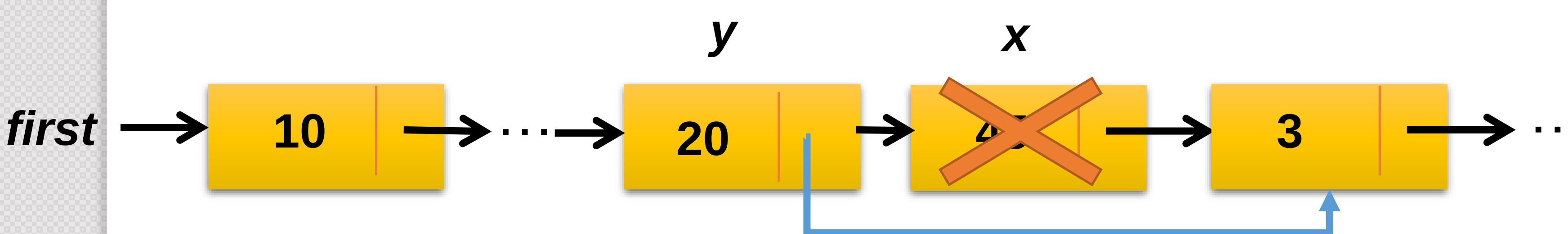
Chain Operations

```
void Chain::Insert50(ChainNode *x)
{
    if( x ) // Insert after x
        x→link = new ChainNode(50, x→link);
    else // Insert into empty list
        first = new ChainNode(50);
}
```



Chain Operations

```
void Chain::Delete(ChainNode *x, ChainNode *y)
{  // x is the node to be deleted and y is the node
   // preceding x
   if( !x || !y ) throw "cannot delete NULL nodes!";
   if(x==first) first = first->link;
   else y->link = x->link;
   delete x; x=NULL;
}
```



4.3

The Template Class Chain

4.3

Template Chain Class

```
Template < class T > class Chain;  // Forward declaration

template < class T >
class ChainNode {
friend class Chain <T>;
private:
        T data;
        ChainNode<T>* link;
};

template <class T>
class Chain {
public:
        // Constructor
        Chain(void) {first = last = NULL;}

        // Chain operations…

private:
       ChainNode<T> *first;
       ChainNode<T> *last;
};
```

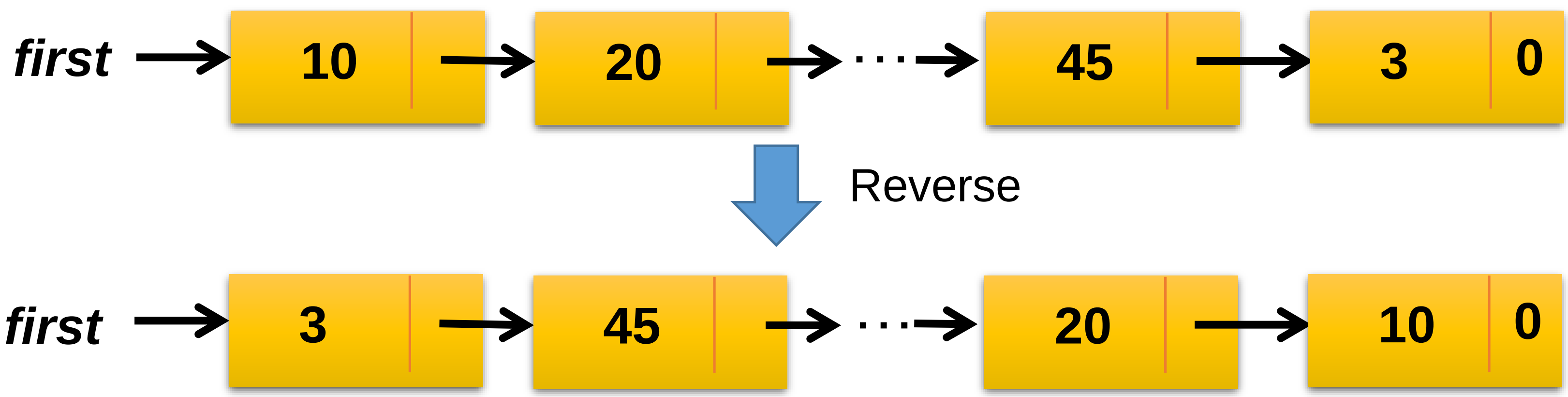
Chain Operations

```
template < class T >
void Chain<T>::InsertBack(const T& e)
{
    if(first) {// Non-empty chain
        last->link = new ChainNode<T>(e);
        last = last->link;
    }
    else // Insert into an empty chain
        first = last = new ChainNode<T>(e);
}
```

```
template < class T >
void Chain<T>::Concatenate(Chain<T>& b)
{   // b is concatenated to the end of *this
    if ( first ) { last->link = b.first; last = b.last; }
    else { first = b.first; last = b.last; }
    b.first = b.last = 0;
}
```

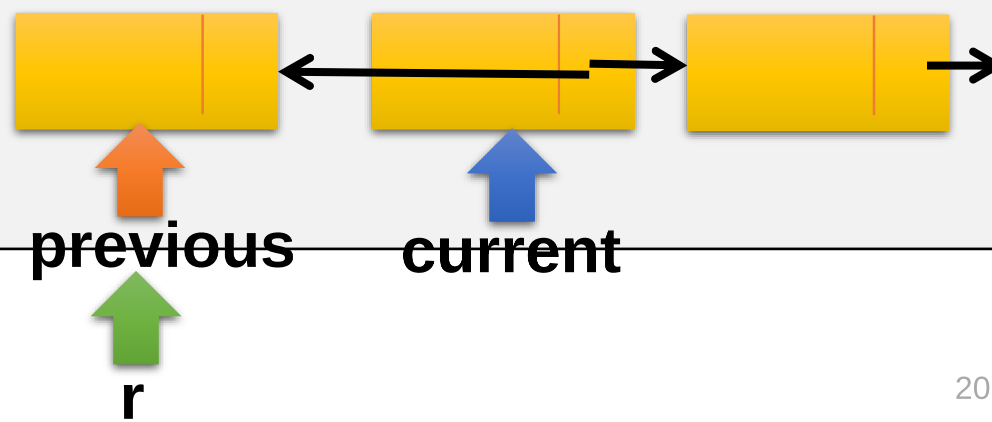
Chain Operations

- Reverse a chain, such that $(a_1, \ldots, a_n)$ turns into $(a_n, \ldots, a_1)$.



Chain Operations

```
template < class T >
void Chain<T>::Reverse(void)
{  // Turn a chain,(a1, …, an) into (an, …, a1)
   ChainNode<T> *current = first, *previous = NULL;
   while (current) {
      ChainNode<T> *r = previous;
      previous = current;     // r is behind the previous
      current = current->link; // move current to next node
      previous->link = r; // link previous to previous node
   }
   first = previous;
}
```



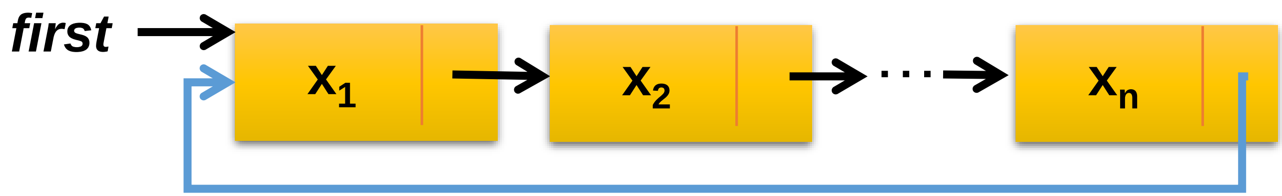
4.4

Circular Lists

4.4

Singly-linked Circular Lists

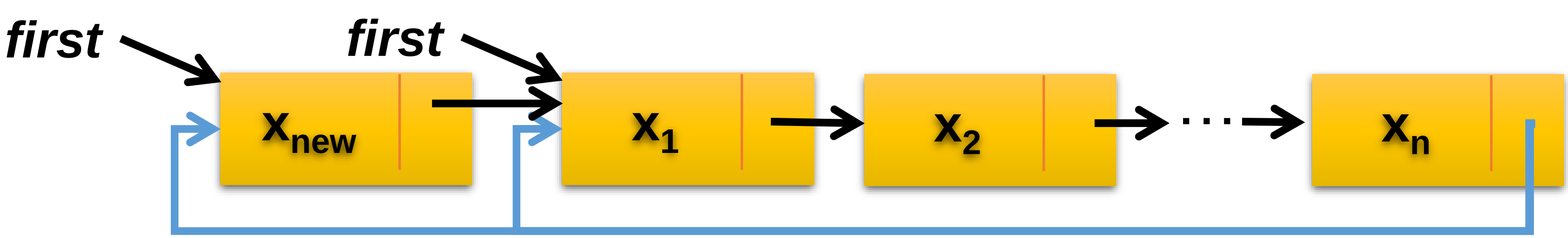
- Can visit a node from any position
- The link field of the last node points to the first node



- Check for the last node
  - if(current->link == ***first***)
- Easier to store the last node of a circular list and access the first node via last->link

Circular Lists : Insert

- Suppose we want to insert a new node at the front of list
- Set link field of new node to ***first*** and set ***first*** to new node
- Go to the last node and set the link field to new node

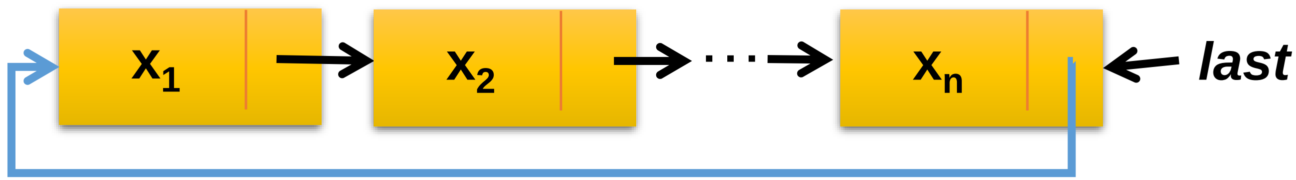


Circular Lists: Insert at Front

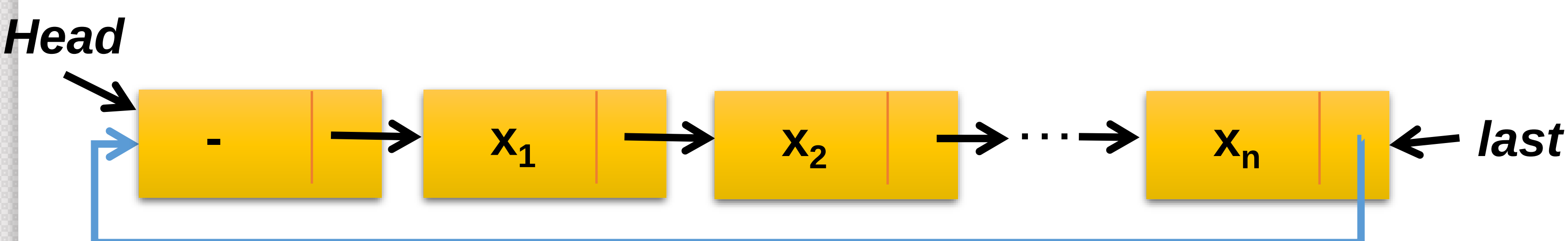
```
Template<class T>
void CircularList<T>::InsertFront(const T& e)
{
   ChainNode<T>* newNode = new ChainNode<T>(e);

   if(last){ // nonempty list
     newNode->link = last->link;
     last->link = newNode;
   }
   else{ // empty list
     last = newNode;
     newNode->link = newNode;
   }
}
```

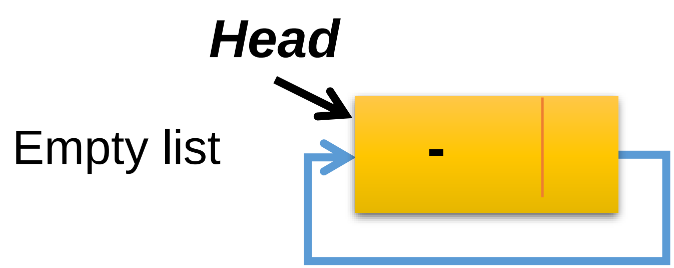
Circular Lists

- How to represent an "empty" list?



- Introducing a dummy node "Header"





4.9

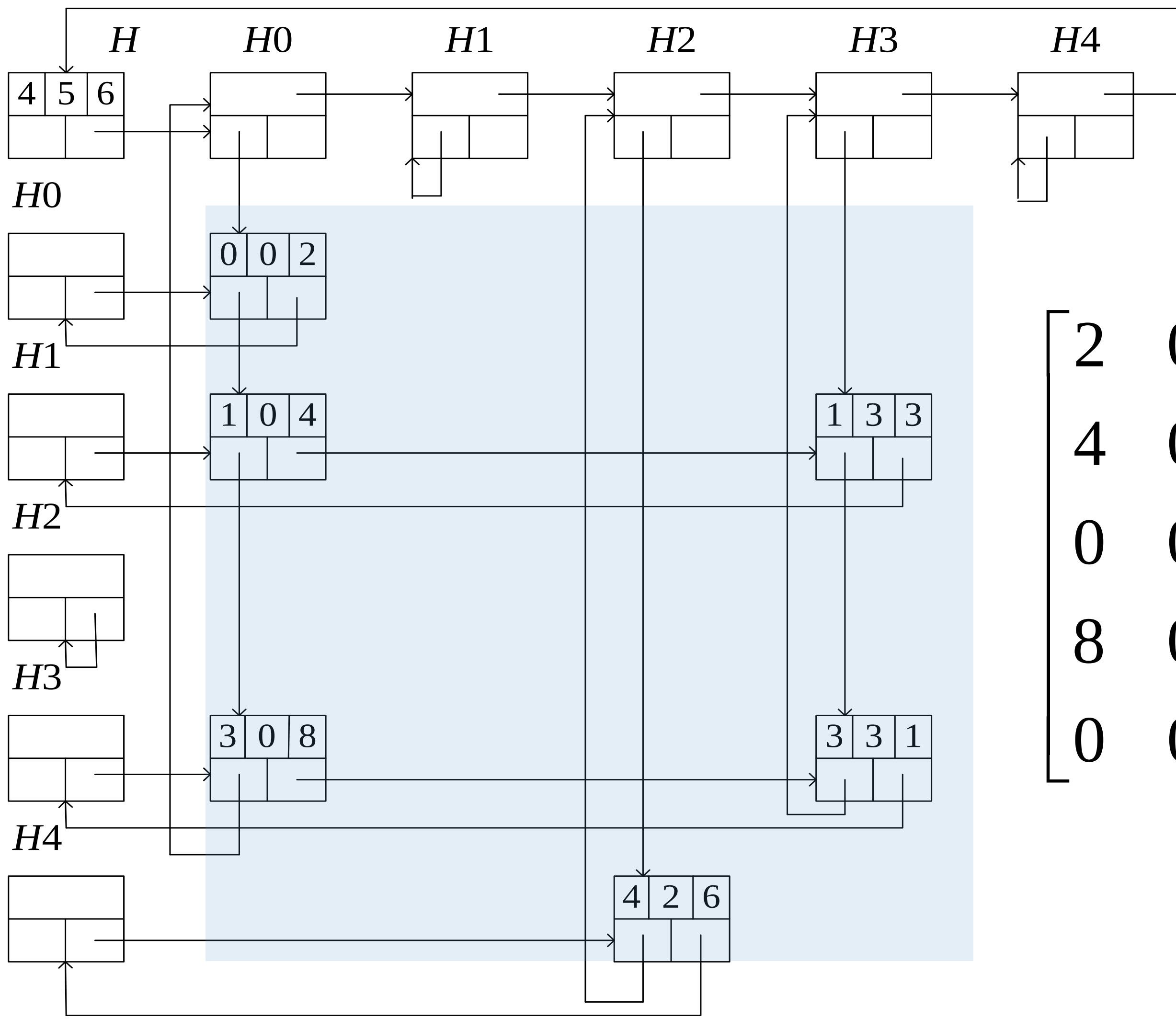
Sparse Matrices

Sparse Matrix

- A matrix has many **zero** elements.
- Devise a sequential array
  - store **non-zero** elements
  - **row-major** order.
- Access specific column is difficult!
- Using circular lists representation.

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

Linked Sparse Matrix



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

Linked Structure

- Header node: for each row or column

next	
down	right

  - **Down**: link to the 1st non-zero term in the column
  - **Right**: link to the 1st non-zero term in the row
  - **Next**: link to the next head node
  - The header node for row *i* is also the header node for column *i*

- Element node, each **non-zero** term that stores

row	col	value
down	right	

  - Data of **row, col, and value**
  - A **down** field to link to the next non-zero term in the same **column**
  - A **right** field to link to the next non-zero term in the same **row**

- The header of header nodes (a circular list)
  - Store dimension of the matrix

Create a Sparse Matrix

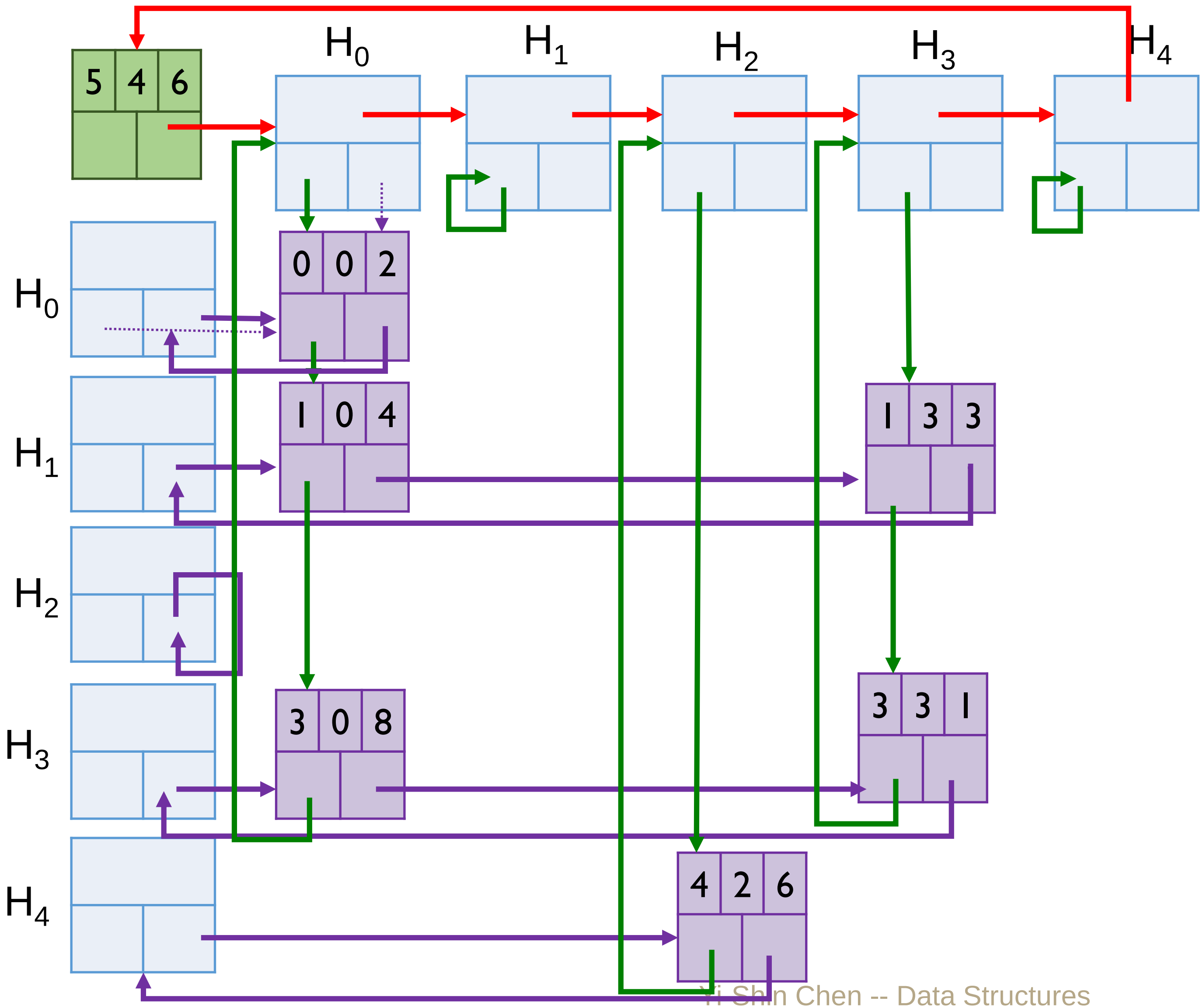
- Given an $n \cdot m$ sparse matrix with r non-zero terms
  - the total number of required nodes are $\max\{n, m\} + r + 1$
- Input format:
  - The 1st line gives the dimension of matrix and # of non-zero terms.
  - Each subsequent input line is a triple of the form (i, j, a_{ij}).
    - Triples are ordered by rows and within rows by columns.

Input
5,4,6;
0,0,2;
1,0,4;
1,3,3;
3,0,8;
3,3,1;
4,2,6;

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

Sparse Matrix in Linked Structure

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



Performance analysis: Create a Sparse Matrix

- Performance analysis
  - Set up header nodes, $O(\max\{n, m\})$
  - Set up non-zero nodes, $O(r)$
  - Close row, column lists, $O(\max\{n, m\})$
  - Link header nodes, $O(\max\{n, m\})$
- Total complexity:
  $O(\max\{n, m\} + r) = O(n + m + r)$

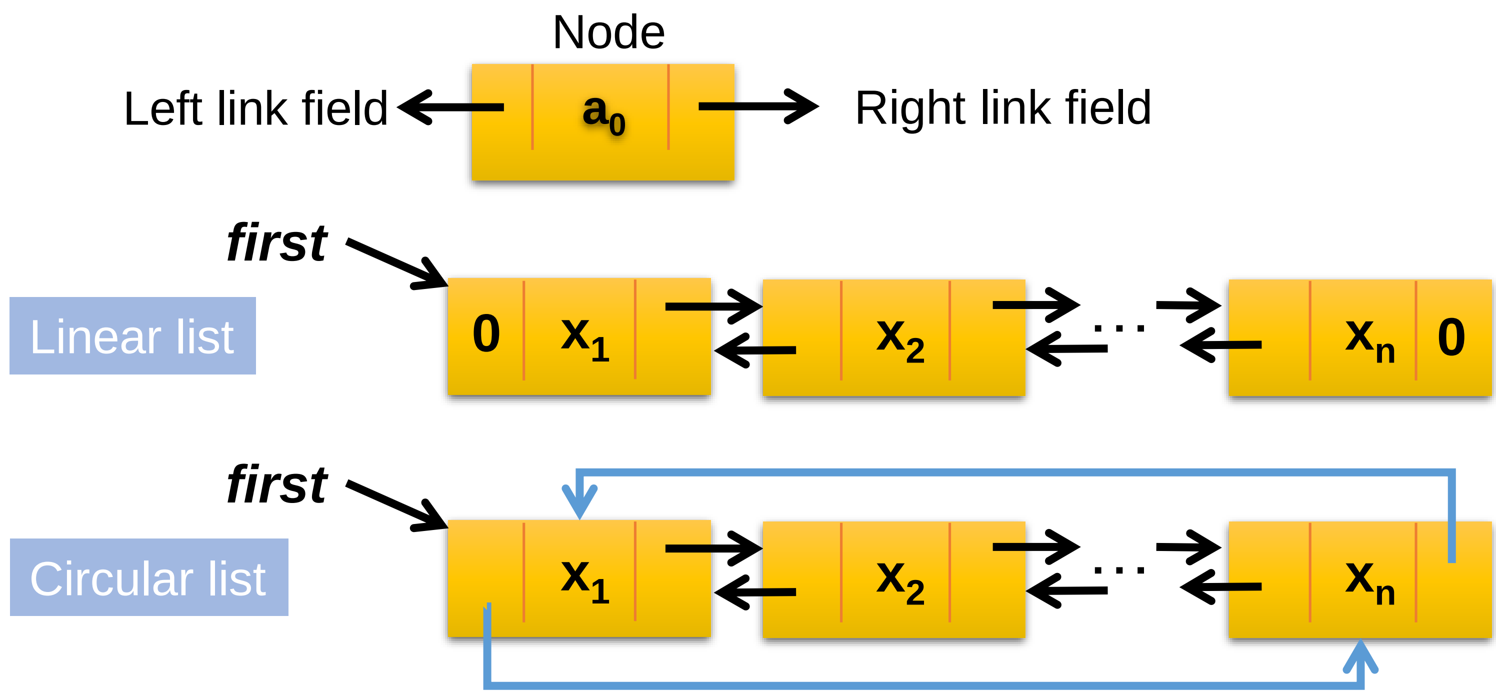
4.10

Doubly Linked Lists

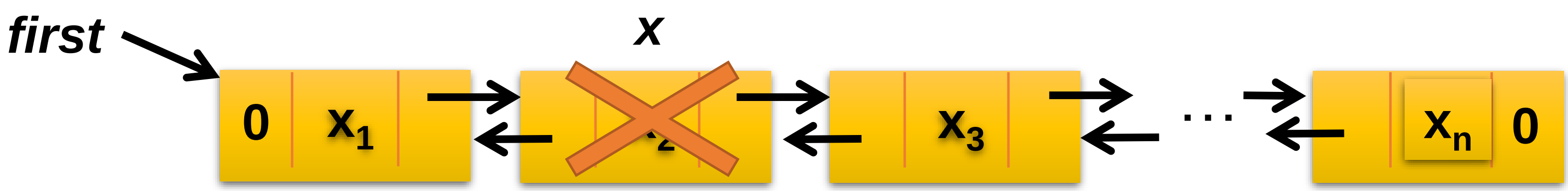
4.10

Double Linked Lists

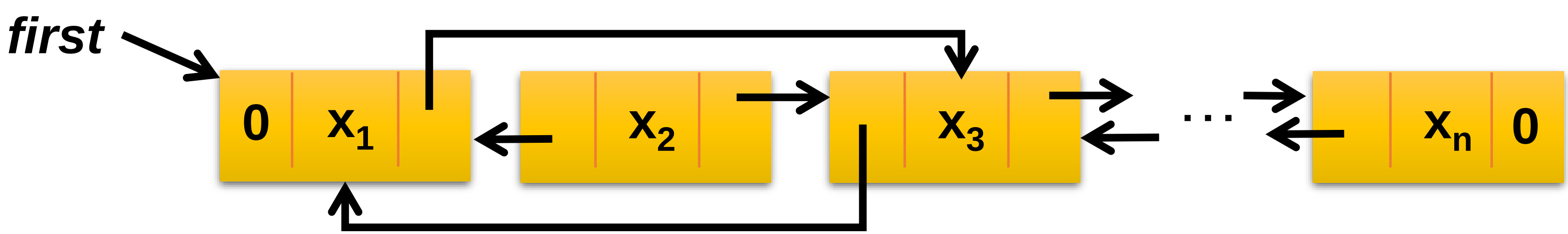
- Each node has **TWO** link fields
- Could move in **TWO directions** to visit nodes



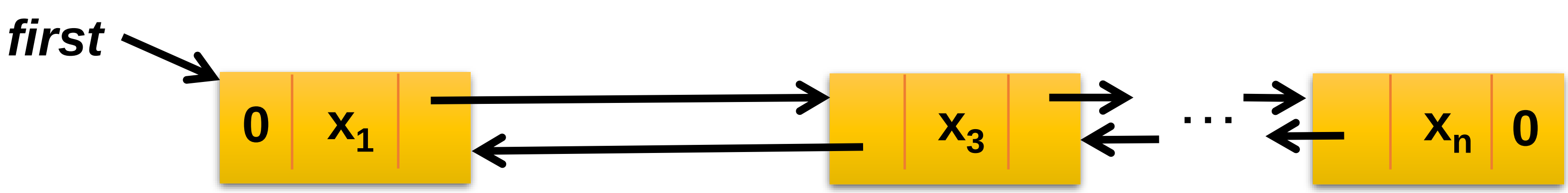
Double Linked Lists: Delete



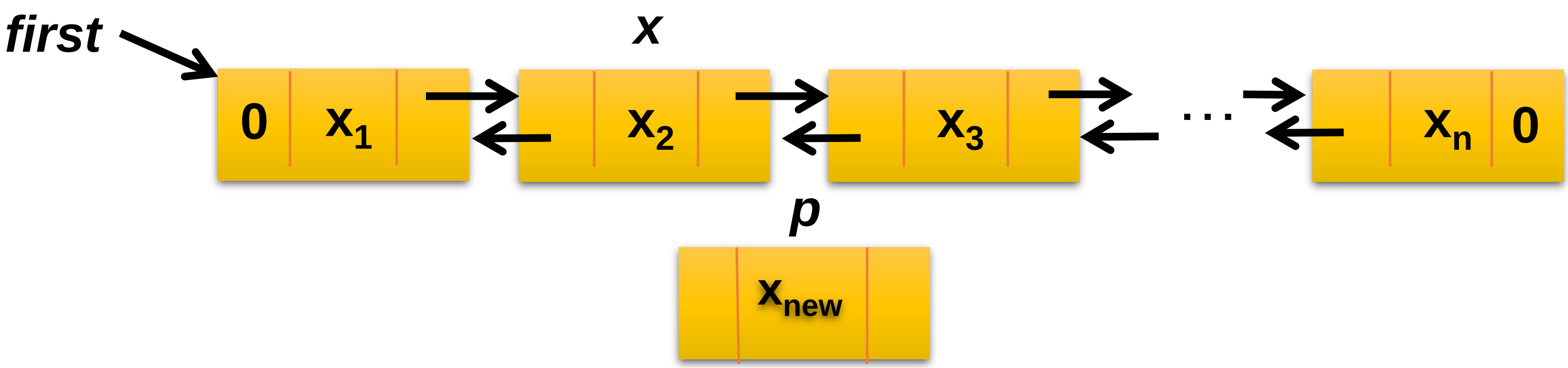

```
x->left->right = x->right;  x->right->left = x->left;
```




```
delete x;
```

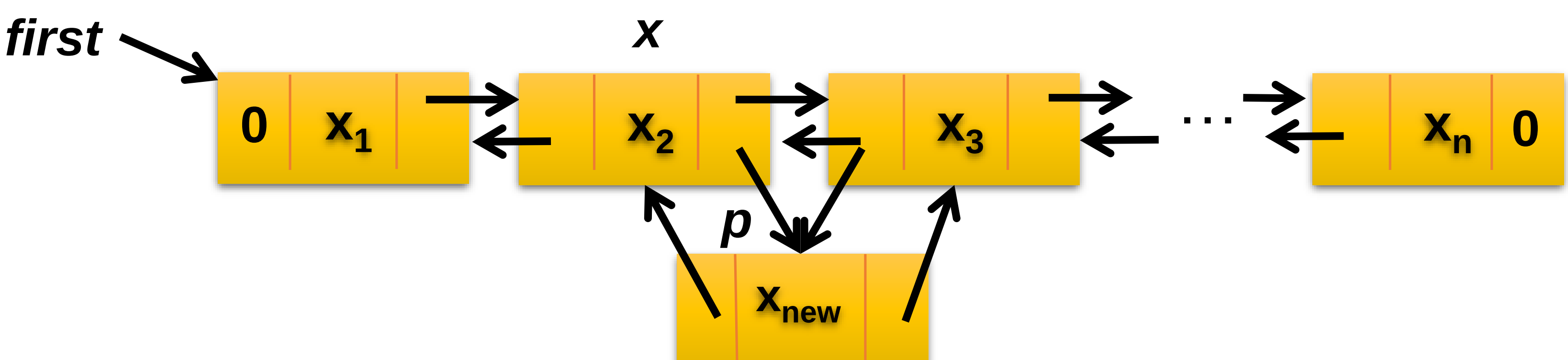


Double Linked Lists: Insert



p→left = x;

p→right=x→right



x→right→left = p;

x→right = p;

Self-Study Topics

- 4.5 Available Space Lists
- 4.6 Linked Stacks and Queues
- 4.7 Polynomial using linked lists
- 4.8 Equivalence Classes
- 4.11 Generalized Lists

Visit Elements in a Container

- Suppose we have a chain C of datatype Chain<int>.
  - Output all integers in C
  - Obtain the maximum, minimum or mean of all integers in C
  - Obtain the sum, product, or sum of squares of all integers in C
- All operations require to visit every element in the chain C!

Visit Elements in an Array

```
For each item in C
{
    currentItem = current item in C;
    do something with currentItem;
}
```

- In an array representation

```
for (int i = 0; i < n; i++)
{
    int currentItem = a[i];
    // do something with currentItem;
}
```

Visit Elements in a Linked List

```
For each item in C
{
    currentItem = current item in C;
    do something with currentItem;
}
```

- In a linked list representation

```
for (ChainNode<int> *ptr=first; ptr!=0; ptr=ptr->link)
{
    int currentItem = ptr->data;
    // do something with currentItem;
}
```

Visiting a Container using Iterator

- A powerful mechanism to visit a container with arbitrary data type.
- Guarantee runtime range safety.
- Applicable to all STL algorithms.
- Suitable for team development.
- Might scarify some amount of

```
// Possible implementation of STL copy algorithm
template < class Iterator >
void copy(Iterator start, Iterator end, Iterator to)
{ // copy from src[start, end) to dst[to, to+end-start)
    while (start != end)
    { *to = *start ; start++ ; to++; }
}
```

What is an Iterator ?

```
void main()
{
   int x [3] = {0,1,2};
   for (int* y = x; y != x+3; y++)
      cout << *y << endl;
}
```

```
void main()
{
   for (Iterator y = start; y != end; y++)
      cout << *y << endl;
}
```

- An ***iterator*** is a pointer to an element in a container.
- Using dereferencing operator (*) to access an element
- Support pre- or post- increment operator (++)

C++ Iterators

- **Input** iterator
  - Read access, pre- and post- “++” operators.
- **Output** iterator
  - Write access, pre- and post- “++” operators.
- **Forward** iterator
  - pre- and post- “++” operators.
- **Bidirectional** iterator
  - pre- and post- “++” and “--” operators.
- **Random access** iterator
  - Permit pointer jumps by arbitrary amounts.
- All iterators supports “==”, “!=” and “*” operators

Forward Iterator for Chain

```
template <class T>
class Chain {
public:
   // Constructor
   Chain(void) {first = last = NULL;}

   // Chain operations…
   class ChainIterator{…};

   // Get the first element
   ChainIterator begin() {return ChainIterator(first);}

   // Get the end of the list
   ChainIterator end() {return ChainIterator(0);}
private:
   ChainNode<T> *first;
   ChainNode<T> *last;
};
```

Forward Iterator for Chain

- General usage

```
void main()
{
    Chain<int> myChain;
    // do operations on myChain here…

    // print out every element in myChain
    Chain<int>::ChainIterator my_it;
    for (my_it = myChain.begin(); myChain!=myChain.end(); ++my_it)
        cout << *my_it << endl;

    // Use STL algorithm to calculate the sum of myChain
    int sum = std::accumulate(myChain.begin(), myChain.end(),0);
}
```

```
Class ChainIterator{ // A nested class within Chain
public:
  // Constructor
  ChainIterator(ChainNode<T>* startNode = 0)
               {current = startNode;}

  // Dereferencing operator
  T& operator*() const {return current->data;}
  T* operator->() const {return &current->data;}
  // Increment operator
  ChainIterator& operator++()  // pre-"++"
  { current = current->link ; return *this; }

  ChainIterator operator++(int)// post- "++"
  {
    ChainIterator old = *this;
    current = current->link;
    return old;
  }

  // Equality operators
  bool operator!=(const ChainIterator right) const
  { return current != right.current; }

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

private:
   ChainNode<T>* current;
};
```