CHAPTER 4
LINKED LISTS

Iris Hui-Ru Jiang

Fall 2008

Linked Lists

- **Contents**
  - Array vs. list
  - Singly linked lists
  - Doubly linked lists
  - Applications
- **Readings**
  - Chapter 4
  - C++ STL
    - list
    - iterators
      - iterator, const_iterator
      - reverse_iterator, const_reverse_iterator
      - iterator category
      - input/output ← forward ← bidirectional ← random access

Review of Sequential Representations (1/2)

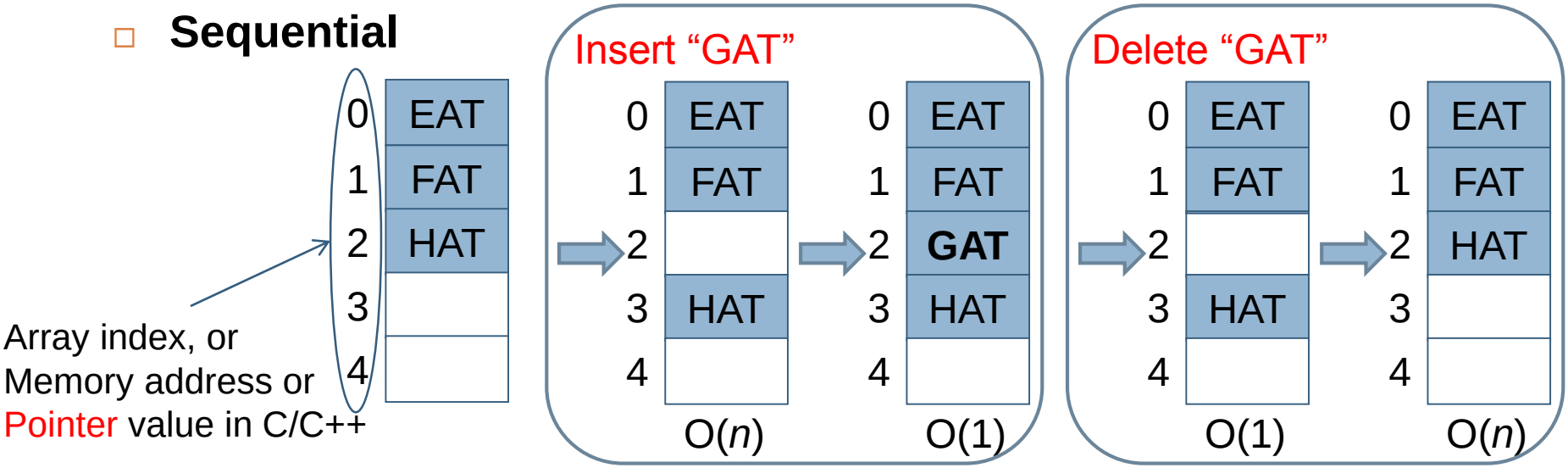
- **Consider an array + a sequential mapping**
  - Elements are stored in sequential in memory
- **Array**
  - If a_i is stored at location L_i, a_{i+1} is stored at location L_i+1
- **Queue**
  - If a_i is stored at location L_i, a_{i+1} is stored at location $(L_i+1)\%n$
- **Stack**
  - If *top* is stored at location L_T, one beneath it is stored at L_T -1
- **Query an arbitrary element?**
  - In O(1) time 

Review of Sequential Representations (2/2)

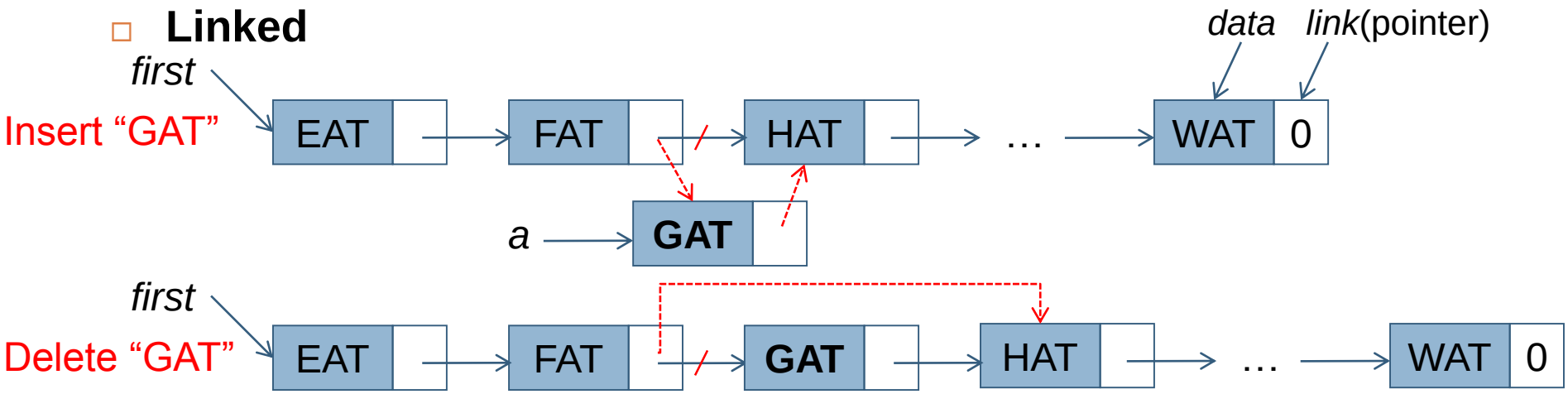
- **Array**
  - Append at the end
- **Stack**
  - Insert and delete only at one end
- **Queue**
  - Insert at one end; delete at the other
- **Consider ordered lists**
  - Drawback 1: What if we want to insert and delete an arbitrary element (in the middle of it)?
    - In O(*n*) time 
  - Drawback 2: What if we want to handle multiple sizes of ordered lists?
    - Space inefficiency (MaxSize, MaxSize=?)
  - $\Rightarrow$ Access in O(1) time and allow various sizes? How?

Sequential vs. Linked

- **Sequential**



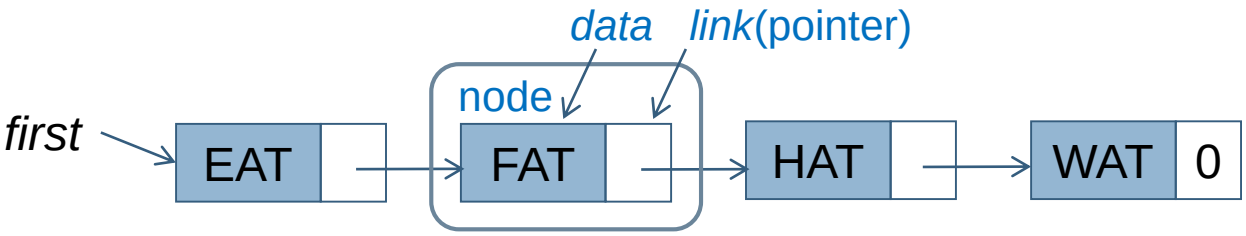
- **Linked**



Singly Linked Lists

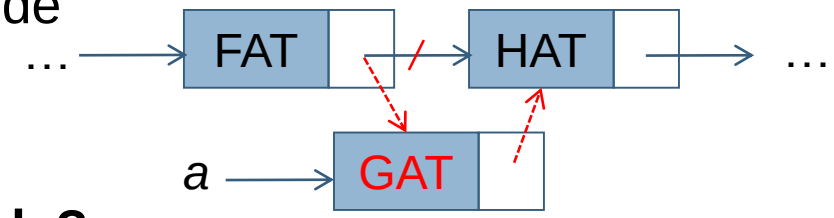
A Singly Linked List (Chain)

- **A node *i* has two types of fields**
  - *data*[*i*]: one or more values
  - *link*[*i*]: pointer to next element ⇒ linked together
- **A head pointer points to the first element**
- **The end element points to 0**



	data	link
1	HAT	6
2		
3	EAT	7
4		
5		
6	WAT	0
7	FAT	1

- **Insert a node**
  - New a node
  - Set data
  - Set links
- **Delete a node?**

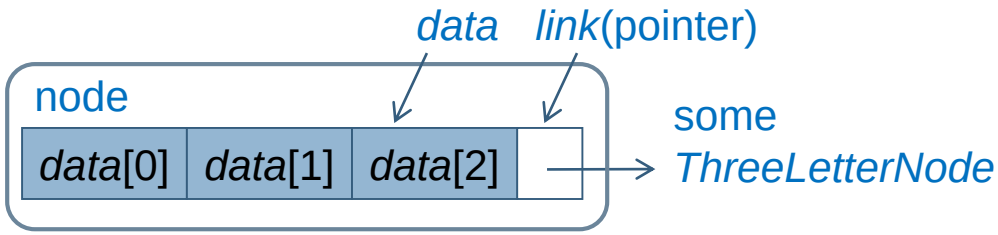


	data	link
1	HAT	6
2		
3	EAT	7
4	GAT	1
5		
6	WAT	0
7	FAT	4

Defining a Node in C++

- **Need to know the type of each of its fields**
  - e.g., char, array, ..., pointer
- **Example: three-letter words**

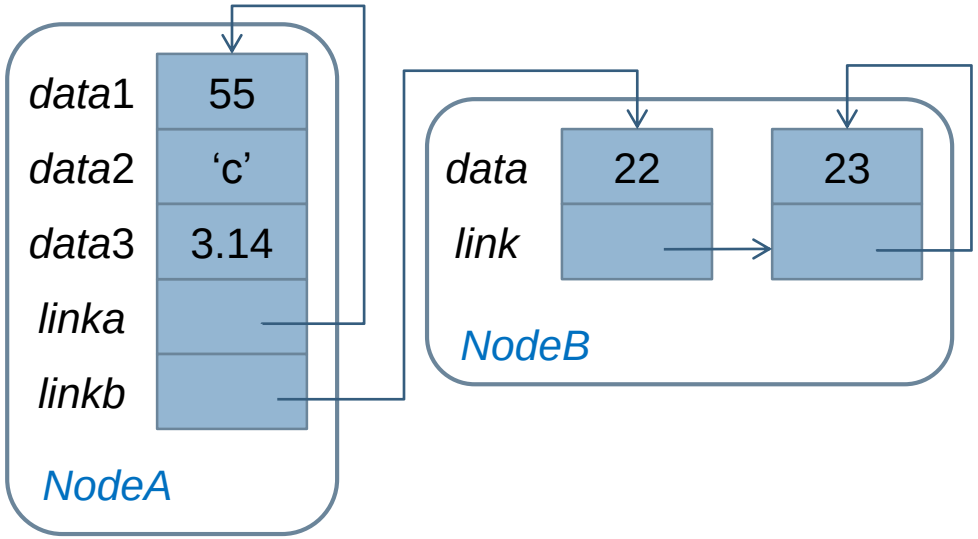
```
class ThreeLetterNode {
private:
    char data[3];
    ThreeLetterNode *link;
};
```



- **Example: a complex list structure**

```
class NodeA {
private:
    int data1;
    char data2;
    float data3;
    NodeA *linka;
    NodeB *linkb;
};
```

```
class NodeB {
private:
    int data;
    NodeB *link;
};
```



HAS-A Relationship in C++

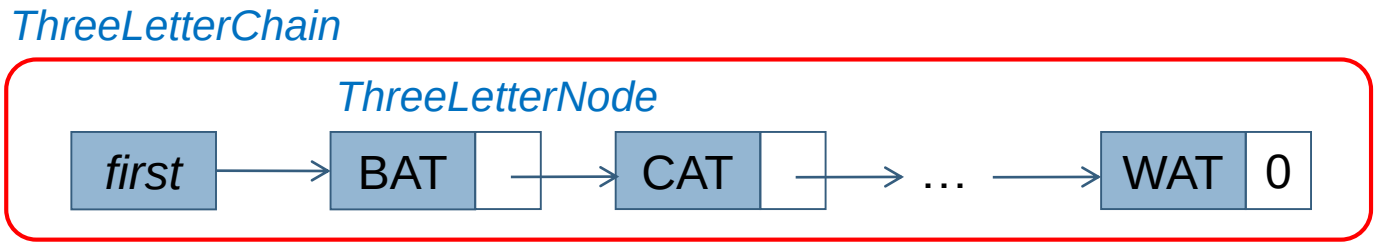
- **Definition: an object *A* HAS-A object *B* if**
  - *A* conceptually contains *B* or *B* is a part of *A*
  - ⇒ in C++, object *B* is a data member of object *A*

```
class PC {
    processor a;
    memory b;
    disk c;
    …;
};
```

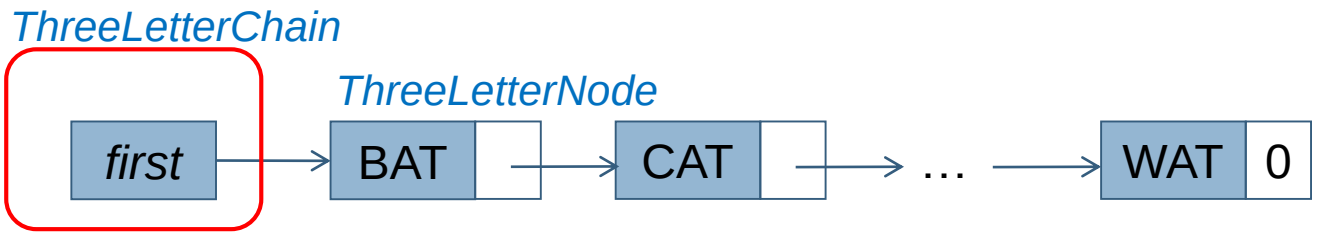
```
class hi_tech_firm {
    CEO golf;
    Manager hi_pay[10];
    Engineer Karoshi[10000];
    …;
};
```

A Chain of Three-Letter Words
-- Implemention (1/3)

- **An *ThreeLetterChain* consists of 0 or more *ThreeLetterNode*'s**
  - *ThreeLetterChain* HAS-A *ThreeLetterNode*
  - Conceptual relationship between the two classes



  - Actual relationship between the two classes



A Chain of Three-Letter Words
-- Implemention (2/3)

- **Use a composite of *ThreeLetterChain* and *ThreeLetterNode***
  - *ThreeLetterChain* manipulates list operations
  - *ThreeLetterNode* is reusable

```
class ThreeLetterChain; // forward declaration
class ThreeLetterNode {
    friend class ThreeLetterChain;  // has been declared
    private:
        char data[3];
        ThreeLetterNode *link;
}

class ThreeLetterChain { // true definition
    public:
        // Chain manipulation operations (discussed later)
    .
    .
    private:
        ThreeLetterNode *first;  // has a TreeLetterNode pointer
};
```

ThreeLetterChain can access the private part of *ThreeLetterNode*

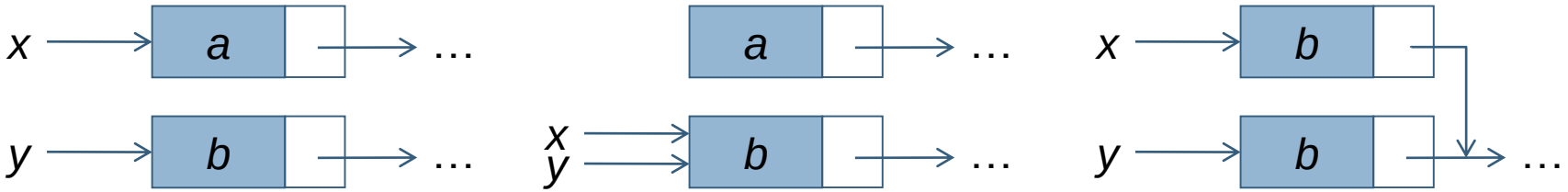
A Chain of Three-Letter Words
-- Implemention (3/3)

- **Use nested classes**
  - One class is defined inside another
  - *ThreeLetterNode* is defined inside the private portion of *ThreeLetterChain*

```
class ThreeLetterChain { // true definition
    public:
        // Chain manipulation operations
    .
    .
    private:
        class ThreeLetterNode { // nested class
            public:
            char data[3];
            ThreeLetterNode *link;
        };
        ThreeLetterNode *first;
};
```

Pointer Manipulation in C++

- **Dynamic memory allocation and de-allocation**
  - *f* = **new** *ThreeLetterNode*;
    - **f* denotes the node of type *ThreeLetterNode* that is created
  - **delete** *f*;
- **The null pointer in C++ is constant 0 (NULL)**
- **Two pointer variables of the same type can be compared**
- **The effect of assignments on pointers**



(a) Initially, x and y are pointers of the same class

(b) after *x* = *y*

(c) after **x* = **y*

Chain Manipulation Operations

-- a Chain of **int**s


```
class Chain; // forward declaration

class ChainNode {
friend class Chain;
public:
   ChainNode (int element = 0, ChainNode* next = 0)
      // 0 is the default value for element and next
      {data = element; link = next;}
private:
   int data;
   ChainNode *link;
};

class Chain { // true definition
   ChainNode *first; // has a ChainNode pointer
public:
   void Create2(); // create 2 nodes
   void Insert50(ChainNode *x); // insert a node
   void Delete(ChainNode *x, ChainNode *y); // delete a node
};
```

Chain Manipulation Operations

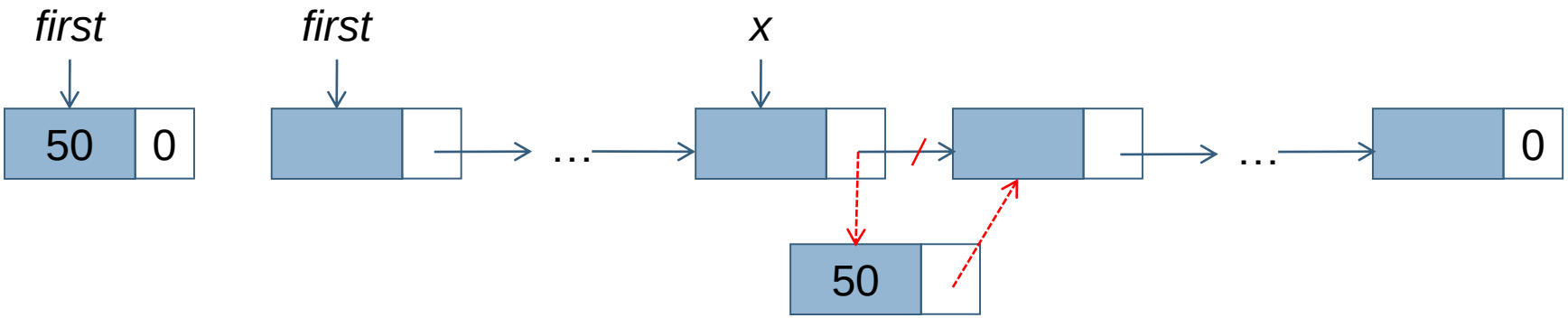
-- Creating a 2-Node List


```
void Chain::Create2() {
    // create and set fields of second node
    ChainNode *second = new ChainNode (20, 0);
    // create and set fields of first node
    first = new ChainNode (10, second);
};
```

Chain Manipulation Operations

-- Inserting a Node


```
void Chain::Insert50(ChainNode *x) {
    if (first) // insert after x
        x->link = new ChainNode (50, x->link);
    else // insert into empty list
        first = new ChainNode (50, 0);
};
```

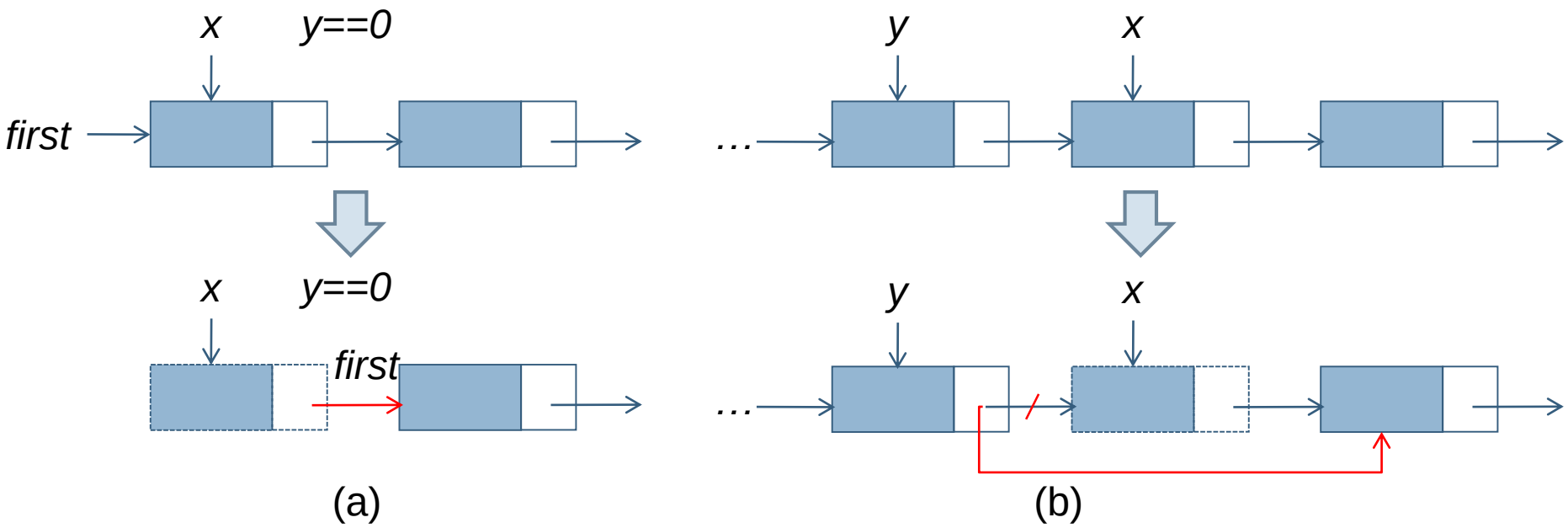


(a) (b)

Chain Manipulation Operations

-- Deleting a Node


```
void Chain::Delete(ChainNode *x, ChainNode *y) {
    // Let y point to the node (if any) that precedes x,
    // and let y == 0 iff x == first
    if (x == first) first = first->link;
    else y->link = x->link;
    delete x;
};
```



The Template Class Chain


```
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 { // definition here
public:
   Chain() {first = 0;}  // ctor, initialize first to 0
   // Chain manipulation operations
   .
   .
private:
   ChainNode <T> *first;
};
```

Usage:
Chain <**int**> *intlist*;
Chain <*Rectangle*> *reclist*;

Chain Iterators

- **Definition: An iterator is an object**
  - Used to access the elements of a container class one by one
- **Motivation: Consider an integer container *C*, if we want to…**
  - Output all integers in *C*
  - Obtain the max, min, mean, median or mode of all integers in *C*
  - Obtain the sum, product, or sum of squares of all integers in *C*
  - …
- **To solve each of these operations, we have to examine all elements of the container class**

Chain Iterators

-- Example: to Find the max Element


```
1.  int x = std::numeric_limits<int>::min();
     // initialization, must include <limits>
2.  for each item in C
3.  {
4.      currentItem = current item of C;
5.      x = max(currentItem, x); // do something with currentItem
6.  }
7.  return x; // postprocessing
```

- **What if an array *a* of size *n*?**

```
2.  for (int i = 0; i < n; i++)
4.      currentItem = a[i];
```

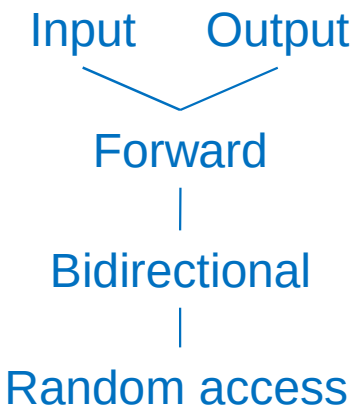
- **What if a nonempty chain of integers?**

```
2.  for (ChainNode<int> *ptr = first; ptr !=0; ptr = ptr->link)
4.      currentItem = ptr->data;
```

- **Can we standardize these operations? (systematic access)**

C++ Iterators

- **In C++, an iterator**
  - Is a pointer to an element of an object
  - Supports equality operators: ==, !=
  - Supports dereference operators: *
- **C++ STL defines 5 categories of iterators**
  - Input: additionally provides read access to the elements pointed at and supports pre- and post-increment operator
  - Output: provides write access to the elements and permits iterator advancement via ++
  - Forward: may be ++
  - Bidirectional: may be ++ and --
  - Random access: jumps by arbitrary amounts



A Forward Iterator for *Chain* (1/2)

Public nested member class of *Chain*

```
class ChainIterator { . . .
public:
   // typedefs required by C++ for a forward iterator omitted
   ChainIterator (ChainNode <T>* startNode = 0) {current = startNode;} // ctor

   // dereferencing operators
   T& operator *() const {return current->data;}
   T* operator ->() const {return &current->data;}
   // increment
   ChainIterator& operator ++() { // preincrement
      current = current->link; // increment iterator
      return *this;} // reference return
   ChainIterator operator ++(int)  { // postincrement
      ChainIterator old = *this; // hold current state of object
      current = current->link; // increment iterator
      return old;} // value return; return unincremented/saved object

   // equality testing
   bool operator!=(const ChainIterator right) const
      {return current != right.current;}
   bool operator==(const ChainIterator right) const
      {return current == right.current;}
private:
   ChainNode <T>* current; };
```

Compiler transforms ++*x* to operator++(*x*)

Compiler transforms *x*++ to operator++(0) (argument 0 is dummy and used to distinguish ++*x* and *x*++)

A Forward Iterator for *Chain* (2/2)


```
class Chain {
    public:
        .
        .
        ChainIterator begin() {return ChainIterator(first);}
        ChainIterator end() {return ChainIterator(0);}
        .
        .
};
```

Usage:

```
Chain <int>::ChainIterator yi = y.begin();
sum = accumulate(y.begin(), y.end(), 0);
```

Chain Operations
-- Insert & Concatenate

- **Insert at the back of a list**
  - with a private data member *last*, pointing to the last node of the chain

```
template <class T>
void Chain<T>::InsertBack(const T& e)
{ // Insert e at the back of *this
   if (first) { // nonempty chain
      last->link = new ChainNode <T>(e);
      last = last->link;
   } else first = last = new ChainNode <T>(e);
}
```

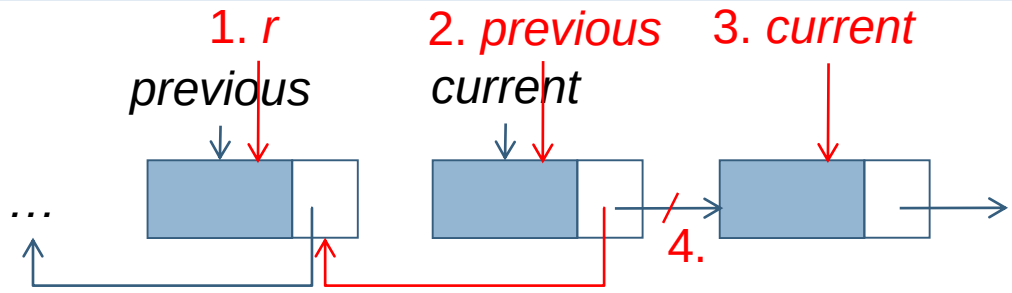
- **Concatenate two chains**

```
template <class T>
void Chain<T>::Concatenate(Chain <T>& b)
{ // Concatenate b 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

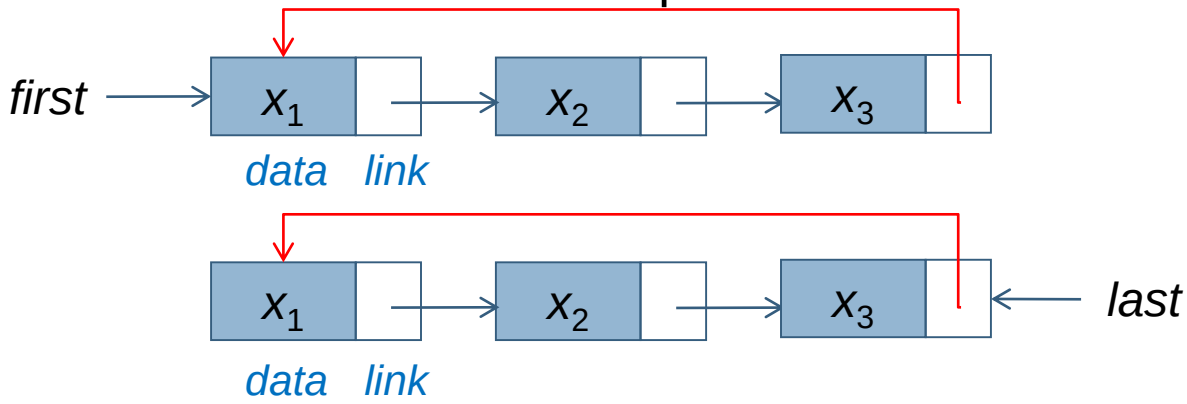
- **Reverse in-place**
  - No element is physically moved
  - Only pointers are changed

```
template <class T>
void Chain<T>::Reverse()
{ // A chain is reversed so that (a1, …, an) becomes (an, …, a1)
   ChainNode <T> *current = first,
                 *previous = 0; // previous trails current
   while (current) {
   1. ChainNode <T> *r = previous;
   2. previous = current; // r trails previous
   3. current = current->link; // current moves to next node
   4. previous->link = r; // link previous to preceding node
   }
   first = previous;
}
```



Circular Lists

- **A circular list is a chain that**
  - the *link* field of the last node points to the first node



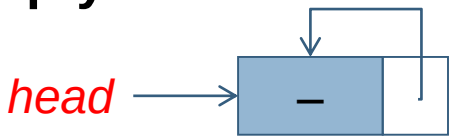
  - What is the advantage to record *last*?
- **Insert at the front of a circular list**

```
template <class T>
void CircularList<T>::InsertFront(const T& e)
{ // Insert e at the front of *this
   ChainNode <T> *newNode = new ChainNode <T>(e);
   if (last) { // nonempty chain
      newNode->link = last->link; last->link = newNode;
   } else { last = newNode; newNode->link = newNode;}
}
```

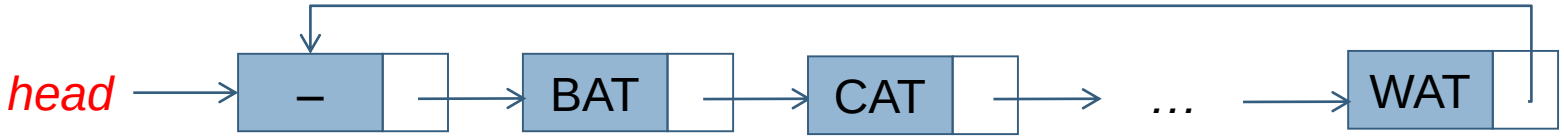
What if inserting at the back?
Add *last = newNode*;
into the **if** clause of *InsertFront*

Circular Lists with a Header Node

- **Introduce a dummy node, called the header node**
- **Empty circular list**



- **A circular list with a header node**



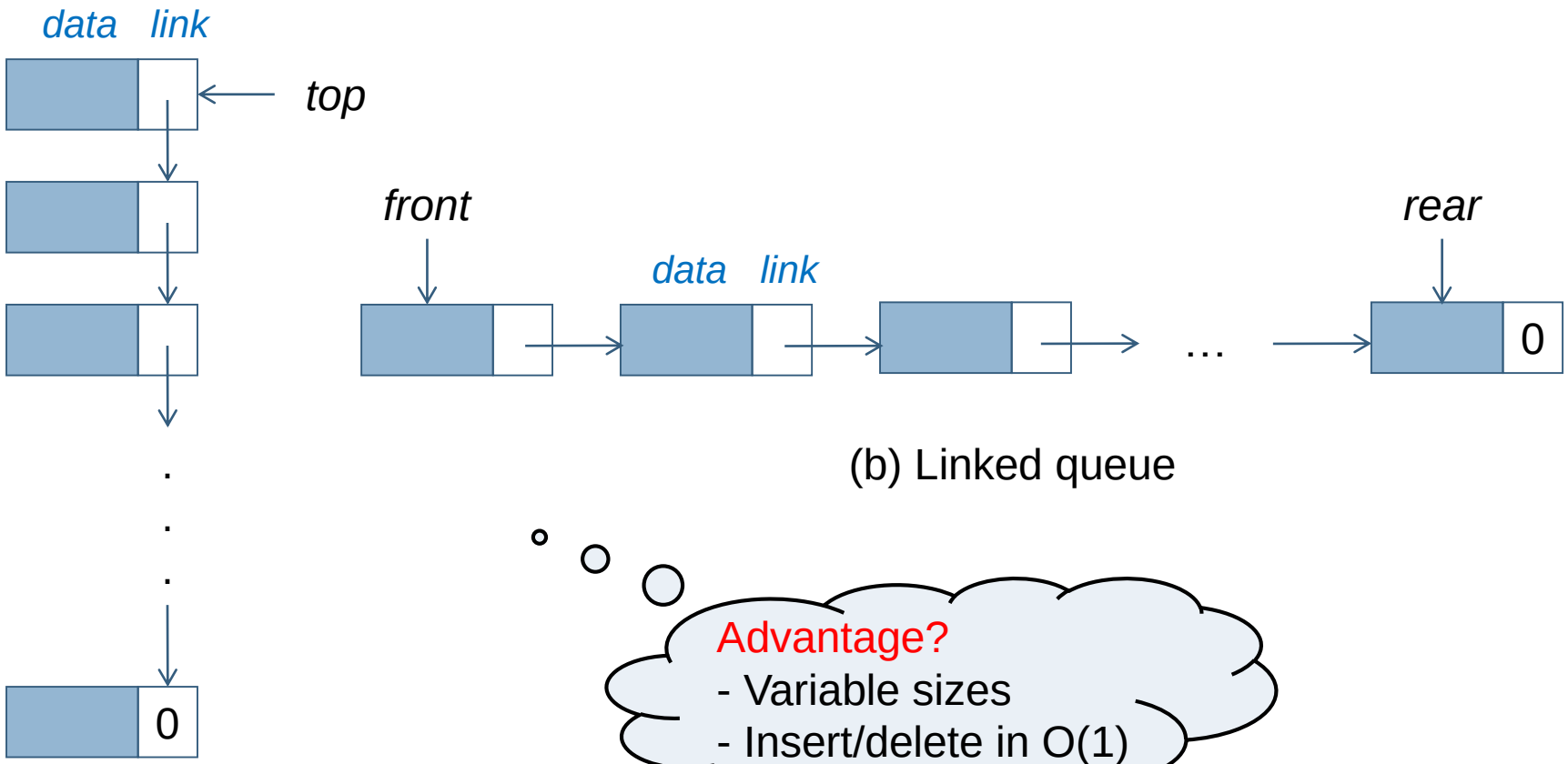
- **No need to handle an empty list as a special case**
  - e.g., no if-else in *InsertFront*

```
template <class T>
void CircularListWithHeader<T>::InsertFront(const T& e)
{ // Insert e at the front of *this
   ChainNode <T> *newNode = new ChainNode <T>(e);
   newNode->link = head->link;
   head->link = newNode;
}
```

Linked Stacks and Queues

Application

Linked Stacks and Queues



(a) Linked stack

(b) Linked queue

Advantage?
- Variable sizes
- Insert/delete in O(1)

Linked Stacks Operations

Assume ctor *LinkedStack*()
sets *top* = 0 (NULL)

- **Push**

```
template <class T>
void LinkedStack<T>::Push(const T& e) {
    top = new ChainNode <T>(e, top);
}
```

- **Pop**

```
template <class T>
void LinkedStack<T>::Pop() {
    if (IsEmpty()) throw "Stack is empty. Cannot delete.";
    ChainNode <T> *delNode = top;
    top = top->link; // remove top node
    delete delNode; // free the node
}
```

- **What if push/pop in linked queues?**

Polynomials

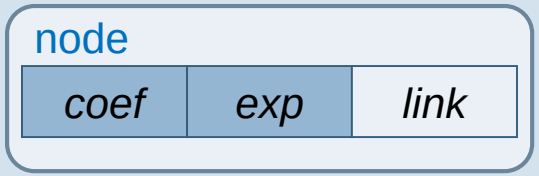
Application

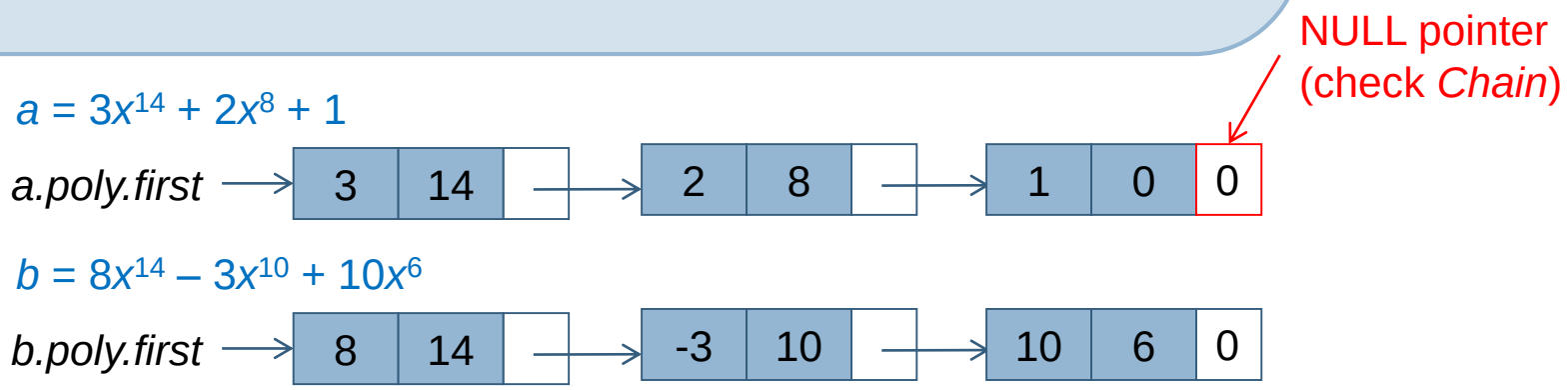
IS-IMPLEMENTED-BY

- **Definition: a data object of type *A* IS-IMPLEMENTED-IN-TERMS-OF a data object of type *B* if**
  - The type *B* object is central to the implementation of the type *A* object
  - $\Rightarrow$ declare *B* as a data member of *A*
  - e.g., *Polynomial* IS-IMPLEMENTED-BY *Chain*
    - Easier insertion/deletion

Polynomials Revisited

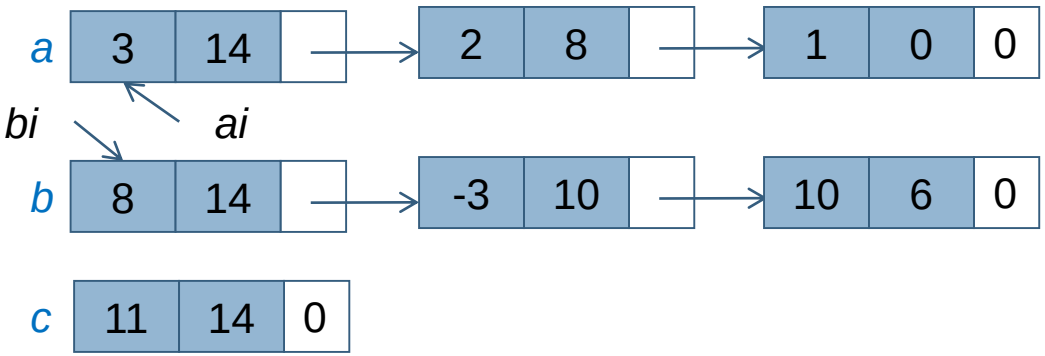

```
struct Term {  // equivalent to class Term { public:
    int coef; // coefficient
    int exp; // exponent
    Term Set(int c, int e) { coef = c; exp = e; return *this;}; // inline
};

class Polynomial {
public:
    // public functions defined here
private:
    Chain <Term> poly;
};
```





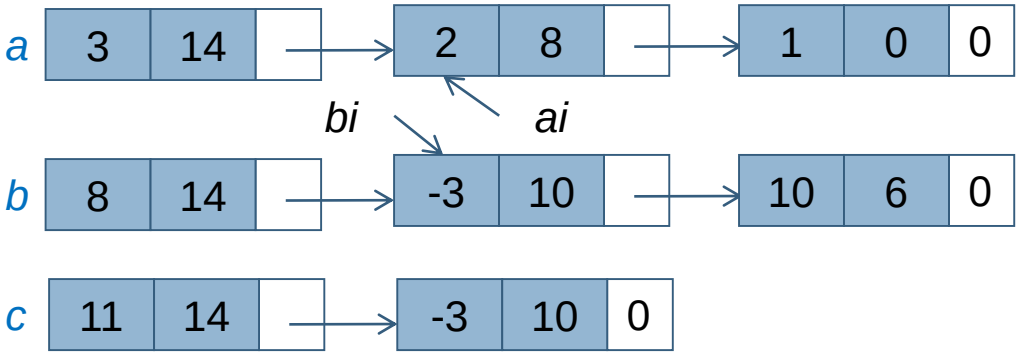
Adding Two Polynomials (1/3)

- $a = 3x^{14} + 2x^8 + 1$
- $b = 8x^{14} - 3x^{10} + 10x^6$
- **Iterators:**
  - *a: ai* : starting at *a.poly.begin*()
  - *b: bi* : starting at *b.poly.begin*()

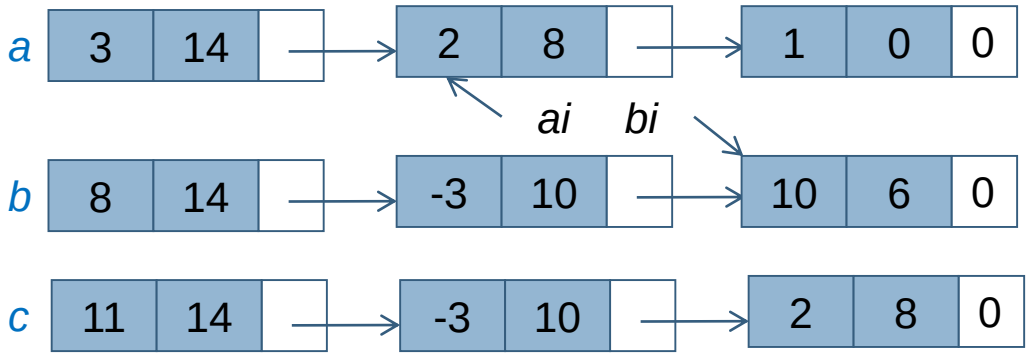


(a) *ai->exp == bi->exp* : sum

Adding Two Polynomials (2/3)



(b) *ai->exp* < *bi->exp* : copy *b*



(c) *ai->exp* > *bi->exp* : copy *a*

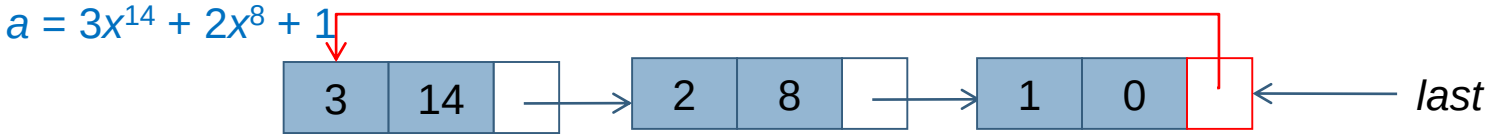
Time: O(*m*+*n*)
- *a* has *m* terms
- *b* has *n* terms

Adding Two Polynomials (3/3)

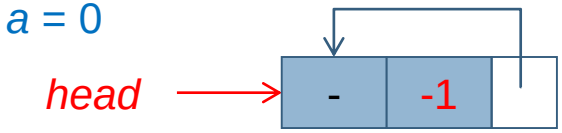
c = *this(*a*)+ *b*

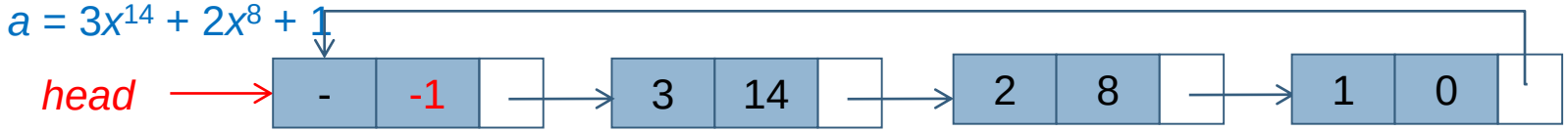
```
Polynomial Polynomial::operator+(const Polynomial& b) const {
    Term temp;
    Chain <Term>::ChainIterator ai = poly.begin(), bi = b.poly.begin();
    Polynomial c;
    while (ai && bi) { // current nodes are not null
        if (ai->exp ==bi->exp) { // sum
            int sum = ai->coef  + bi->coef;
            if (sum) c.poly.InsertBack(temp.Set(sum, ai->exp));
            ai++; bi++; // advance to next term
        } else if (ai->exp < bi->exp) { // copy b
            c.poly.InsertBack(temp.Set(bi->coef, bi->exp));
            bi++; // next term of b
        } else { // copy a
            c.poly.InsertBack(temp.Set(ai->coef, ai->exp));
            ai++; // next term of a
        } // end of if
    } // end of while
    while (ai) { // copy rest of *this (a)
        c.poly.InsertBack(temp.Set(ai->coef, ai->exp)); ai++; }
    while (bi) { // copy rest of b
        c.poly.InsertBack(temp.Set(bi->coef, bi->exp)); bi++; }
    return c;
}
```

Using Circular List Instead



- **What if a zero polynomial?**
  - Use a header node





Adding Two Polynomials
-- Circular List Representation

c = *this(*a*)+ *b*

```
Polynomial Polynomial::operator+(const Polynomial& b) const {
    Term temp;
    CircularListWithHeader <Term>::Iterator ai = poly.begin(), bi = b.poly.begin();
    Polynomial c; // assume ctor sets head->exp = -1
    while (1) {
        if (ai->exp ==bi->exp) { // sum
            if (ai->exp == -1) return c; // NULL or finished
            int sum = ai->coef  + bi->coef;
            if (sum) c.poly.InsertBack(temp.Set(sum, ai->exp));
            ai++; bi++; // advance to next term
        } else if (ai->exp < bi->exp) { // copy b
            c.poly.InsertBack(temp.Set(bi->coef, bi->exp));
            bi++; // next term of b
        } else { // copy a
            c.poly.InsertBack(temp.Set(ai->coef, ai->exp));
            ai++; // next term of a
        } // end of if
    } // end of while
}
```

no need to copy remaining terms

Available Space Lists

Recycle

Available Space Lists

- **When a node or chain is no more used…**
  - Do nothing (i.e., without dtor)
    - Memory leak: the memory it occupies is lost to the program and is not returned to the system. ⇒ system crashes!

Only *first* is freed. The chain of *ChainNode's* are not freed.

  - Free it

```
template <class T>
void Chain <T>::~Chain( ) { // no argument
// free all nodes in the chain
   ChainNode <T>* next;
   for (; first; first = next) {
      next = first->link;
      delete first; // free the node pointed by first
   }
}
```

Time: O(n)
Every time a Chain goes out of scope, the dtor is invoked to free all nodes in it

  - Recycle it for later use
    - Avoid expensive **new** and **delete**
    - Store a chain of deleted nodes into an available space list *av*
      - Time: O(1)

av for *CircularList* (1/3)

- **Example: Consider a *CircularList*, an available space list *av***
  - Points to the first node in the chain of deleted nodes
  - Initializes *av* = 0
  - Uses *CircularList*::*GetNode*() and *CircularList*::*RetNode*() instead of **new** and **delete**
  - Deletes a whole chain by ~*CircularList*() in O(1)

```
template <class T>
class CircularList {
   static ChainNode <T> *av; // available space list
public:
   // public members here
private:
   // other private members here
};
```

All objects of a data type share a single copy of static data member

av for *CircularList* (2/3)

- **Get a node**

```
template <class T>
ChainNode<T>* CircularList<T>::GetNode()
{ // Provide a node for use
    ChainNode <T>* x;
    if (av) {x = av; av = av->link;}
    else x = new ChainNode <T>; // new only if none in av
    return x;
}
```

- **Return a node**

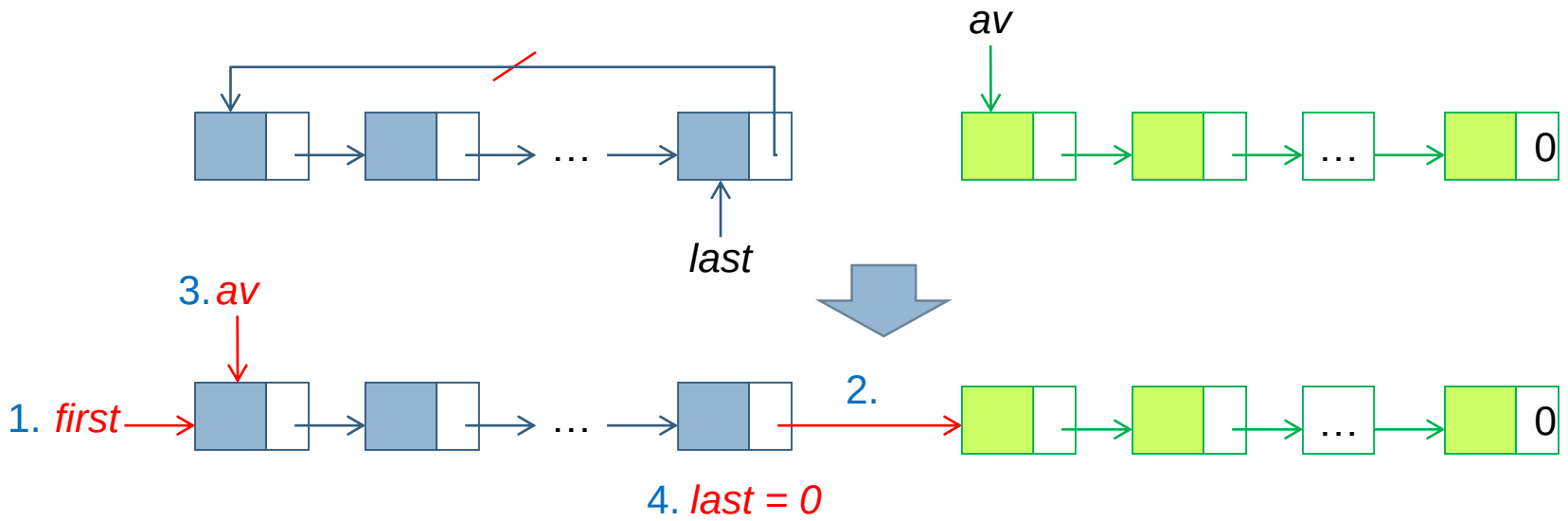
```
template <class T>
void CircularList<T>::RetNode(ChainNode<T>*& x)
{ // Free the node pointed by x
    x->link = av;
    av = x;
    x = 0;
}
```

av for *CircularList* (3/3)

- **Delete a whole circular list in time O(1)**

```
template <class T>
void CircularList<T>::~CircularList()
{ // Delete the whole circular list
   if (last) { // nonempty list
  1. ChainNode<T>* first = last->link;
  2. last->link = av; // last node linked to av
  3. av = first; // first node of list becomes front of av list
  4. last = 0;
   }
}
```

Assume *last* is available here.
What if we don't know where *last* is?



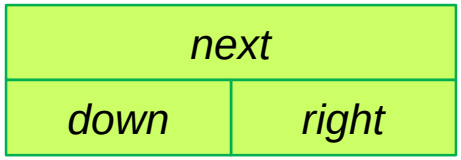
Sparse Matrix

Application

Example: A 5x4 Sparse Matrix

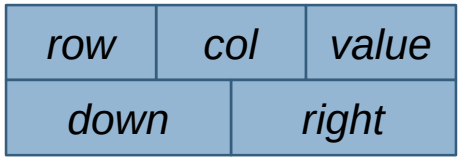
2	0	0	0
4	0	0	3
0	0	0	0
8	0	0	1
0	0	6	0

head = **true**

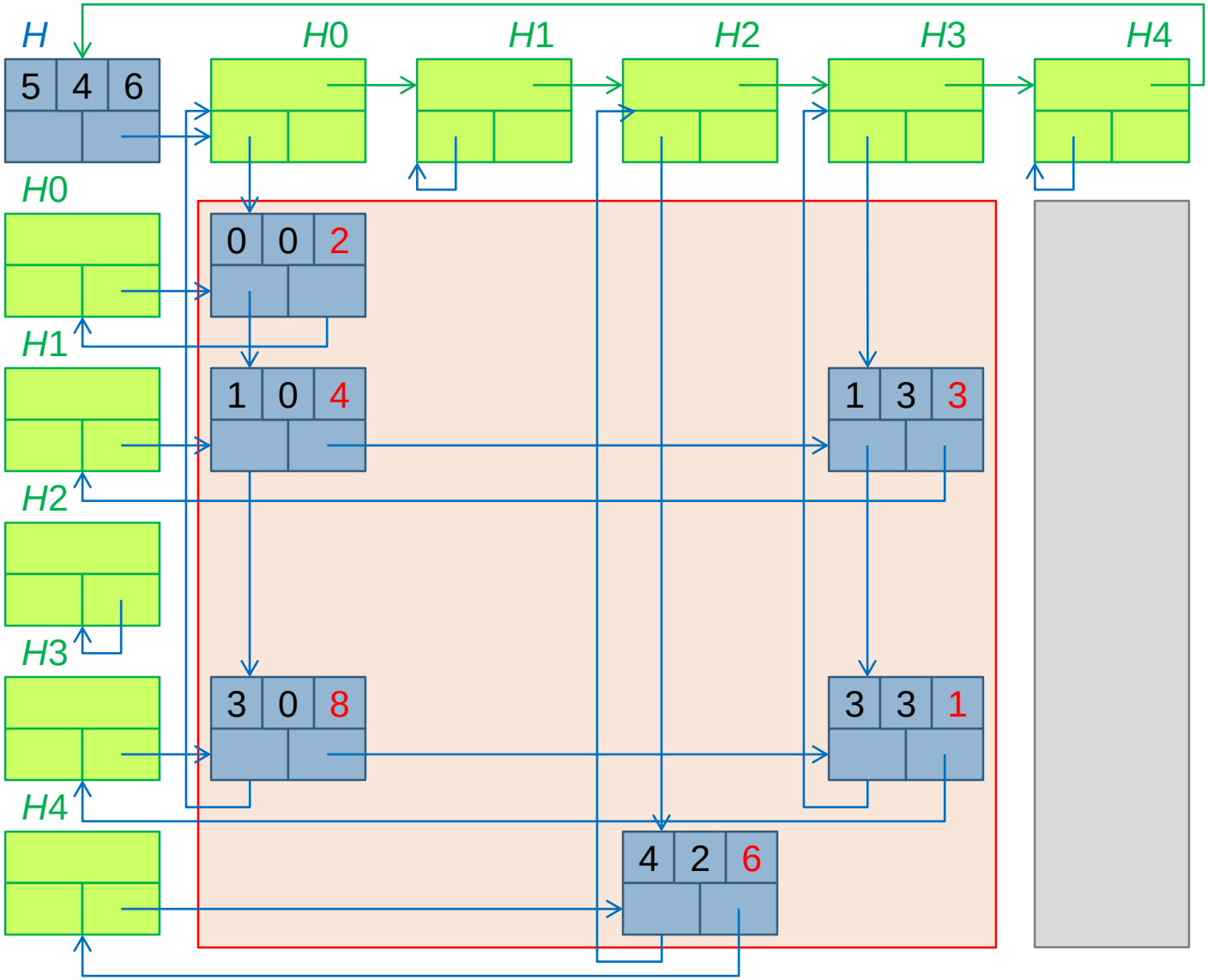


Header node

head = **false**



Element node



Sparse Matrix Representation

- **Represent a sparse matrix by storing only nonzero elements**
  - The sequential representation of Chapter 2
    - Access easily only by row
  - How about linked representation?
    - Access easily both by column or by row
    - Represent each row/column by a circular list
      - $\Rightarrow$ Each nonzero element is both in a row list and in a column list
    - Implementation
      - One header node *Hi* for each row/col *i*
        - Row *i* and col *i* share a header node
      - A header node *H* for the list of header nodes
      - Represent an *n*x*m* matrix with *r* nonzero elements by total max{*n*, *m*}+*r*+1 nodes (cf. *nm*)

Class Definition for a Sparse Matrix (1/2)

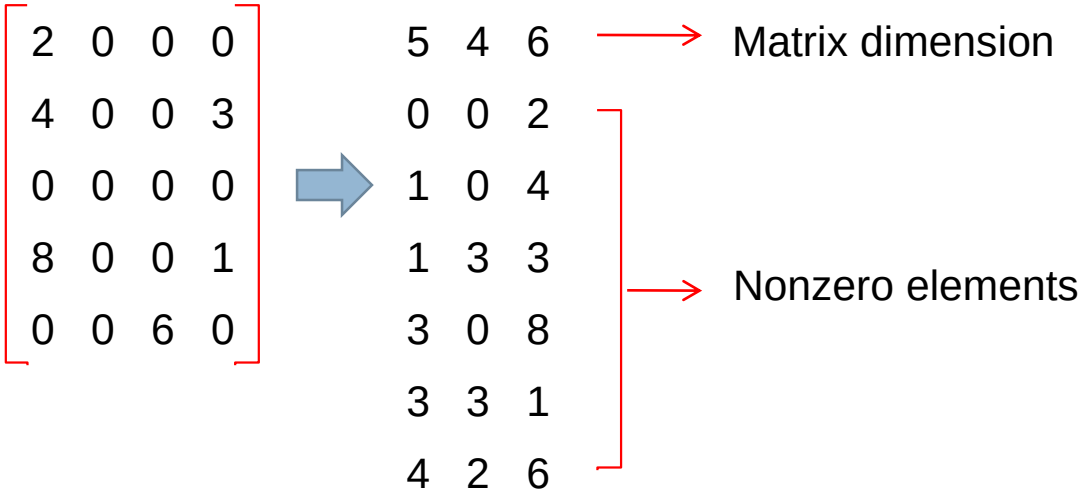

```
struct Triple {int row, col, value;};
class Matrix; // forward declaration
class MatrixNode {
friend class Matrix;
friend istream& operator>>(istream&, Matrix&); // for reading in a matrix
private:
    MatrixNode *down, *right;
    bool head;  // a head node or not
    union { // anonymous union
        MatrixNode *next;
        Triple triple; };
};

MatrixNode::MatrixNode(bool b, Triple *t) { // ctor
    head = b;
    if (b) {right = down = next = this;} // row/column header node
    else triple = *t; // element node or header node for list of header nodes
}
```

Class Definition for a Sparse Matrix (2/2)


```
class Matrix {
friend istream& operator>>(istream&, Matrix&);
public:
   ~Matrix();  // dtor
private:
   MatrixNode *headnode;
};

Matrix::~Matrix() { // dtor
// Return all nodes to the av list, linked via right field
// av is a static variable pointing to the first node of the av list
   if (!headnode) return; // no nodes to delete
   MatrixNode *x= headnode->right; // x points to H0
   headnode->right = av; av = headnode;  // return headnode
   while (x != headnode) { // erase by rows
      MatrixNode *y = x->right; // y points to headnode or 1st element
      x->right = av;
      av = y; // return an entire row
      x = x->next; // move to next row
   }
   headnode = 0;
}
```

Sparse Matrix Input

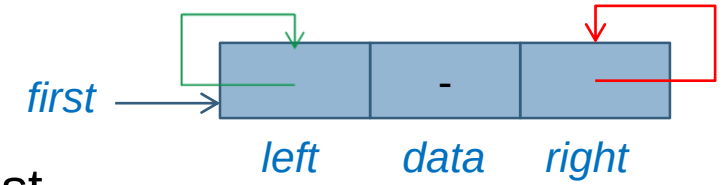
- **Read in a sparse matrix: Program 4.30, page 221**
  - *istream*& **operator**>>(*istream*& *is*, *Matrix*& *matrix*) {…}
  - Matrix dimension first and then nonzero elements in row major order
  - Read in an *n*x*m* matrix with *r* nonzero elements in O(*n*+*m*+*r*) time
- **e.g., a 5x4 matrix**



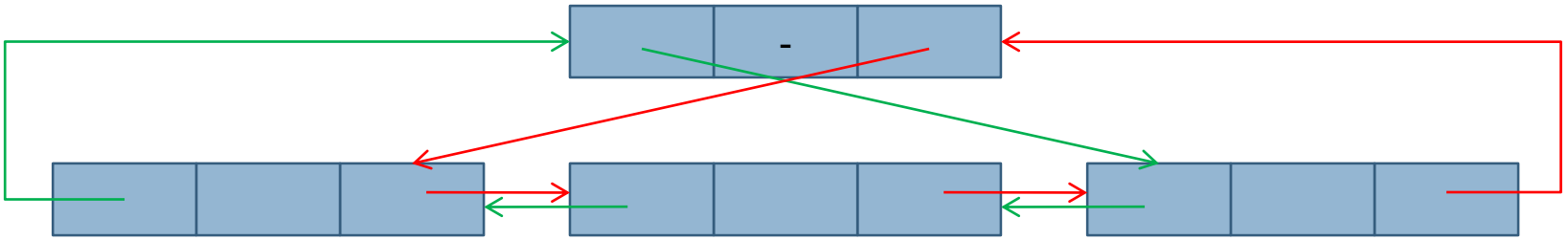
Doubly Linked Lists

Doubly Linked Lists

- **What's wrong with singly linked lists?**
  - Can only be traversed in one direction
    - Hard to find the preceding node of a specific node
    - Hard to delete an arbitrary node
    - …
- **Remedy: Doubly linked lists**
  - A node in a doubly linked list has *left* and *right* links
  - e.g., a doubly linked circular list with a header node
    - Empty list



    - Nonempty list



Class Definition of a Doubly Linked List


```
class DblList;

class DblListNode {
friend class DblList;
private:
    int data;
    DblListNode *left, *right;
};

class DblList {
public:
    // list manipulation operations
    // e.g., ctor, insert a node, delete a node, ...
private:
    DblListNode *first;
};
```

Doubly Linked List Operations (1/2)

- **ctor**

```
DblList::DblList() { // ctor
  first = new DblListNode;  // allocate the head node
  first->left = first;
  first->right = first;
}
```

- **Insertion**

```
void DblList::Insert(DblListNode *p, DblListNode *x) {
// insert node p to the right of node x
    p->left = x; p->right = x->right;
    x->right->left = p; x->right = p;
}
```

Doubly Linked List Operations (2/2)

- **Deletion**

```
void DblList::Delete(DblListNode *x) {
    if (x == first) throw "Please do not cut my head!";
    else {
        x->left->right= x->right;
        x->right->left = x->left;
        delete x;
    }
}
```