Linked Lists

Fan-Hsun Tseng

Department of Computer Science and Information Engineering

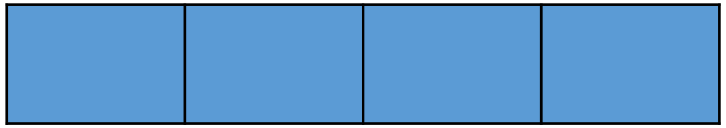
National Cheng Kung University

Outline

- Singly Linked Lists and Chains
- Representing Chains in C++
- The Template Class Chain
- Circular Lists
- Linked Stacks and Queues
- Polynomials
- Equivalence Classes
- Sparse Matrices
- Doubly Linked Lists

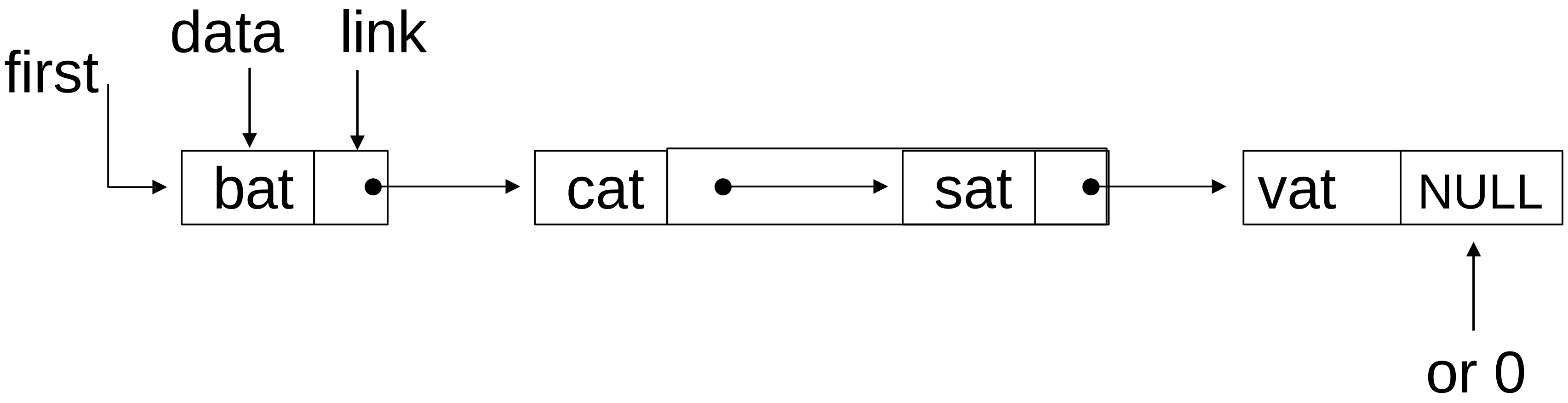
Introduction

- Array
  - Successive items locate a fixed distance
- Disadvantage
  - Data movements during insertion and deletion
  - Waste space in storing n ordered lists of varying size, i.e. $O(n)$
- Possible solution
  - Linked list



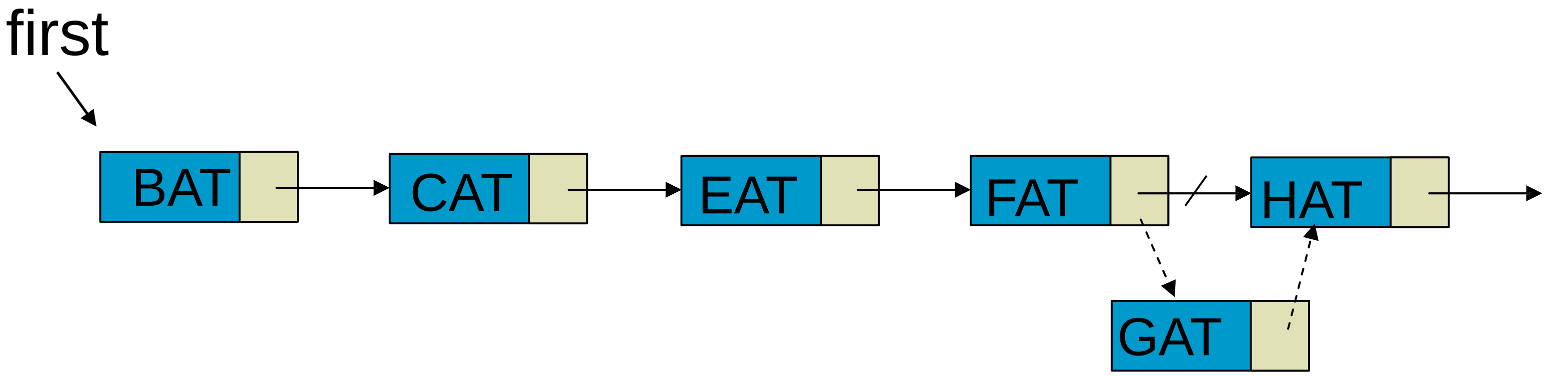
Singly Linked Lists

- **Figure 4.2:** Usual way to draw a linked list



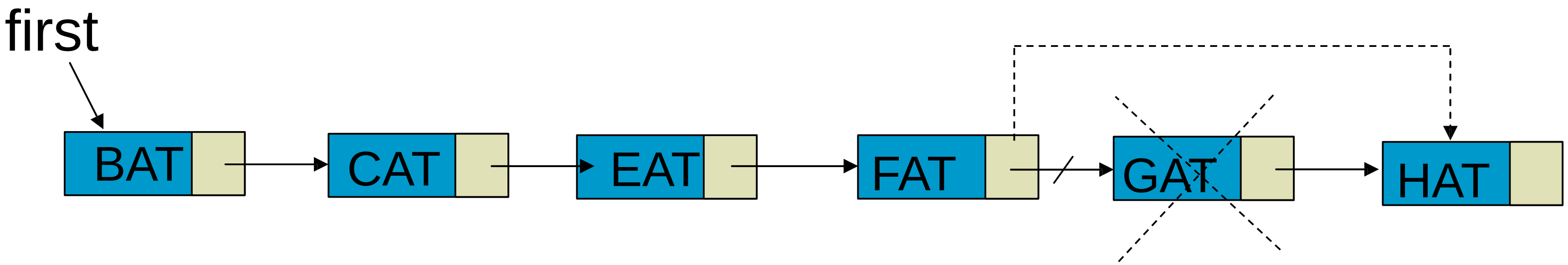
Insertion

- Insert GAT after FAT



Deletion

- Delete GAT from list



Outline

- Singly Linked Lists and Chains
- Representing Chains in C++
- The Template Class Chain
- Circular Lists
- Linked Stacks and Queues
- Polynomials
- Equivalence Classes
- Sparse Matrices
- Doubly Linked Lists

Defining a List Node in C++

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

Designing a List in C++ (Cont’d)

- Design Attempt 1:
  - Use a global variable first which is a pointer of *ThreeLetterNode*.
  - Unable to access to private data members: data and link.
  - A popular approach in C

```
ThreeLetterNode * first;
first->data, first->link
```

Designing a List in C++ (Cont'd)

- Design Attempt 2:
  - Make data members public or define public member functions GetLink(), SetLink() and GetData()
  - Defeat the purpose of data encapsulation
    - We should not know how the list is implemented
- An ideal solution should
  - Only grant those functions that perform list manipulation operations (i.e., inserting a node or deleting a node) access to data members

Designing a List in C++ (Cont'd)

- Design Attempt 3:
  - Use of two classes.
  - Create a class that represents the linked list.
  - The class contains the items of another objects of another class.
- A data object of Type A HAS-A data object of Type B if A conceptually contains B or B is a part of A
  - *Computer* HAS-A *Processor*, or *Book* HAS-A *Page*

Program 4.1: Composite Classes

```
class ThreeLetterList; // forward declaration

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

class ThreeLetterList {
public:
    // List Manipulation operations
.
.
private:
    ThreeLetterNode *first;
};
```

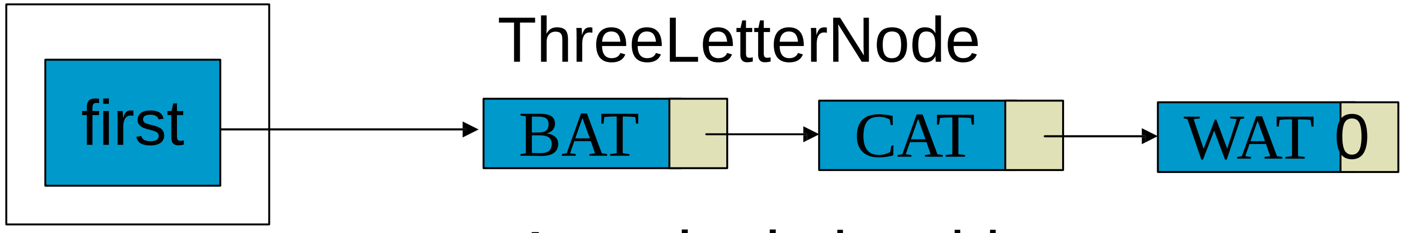
-NodeData

-List Operations

Program 4.1: Composite Classes (Cont'd)

•

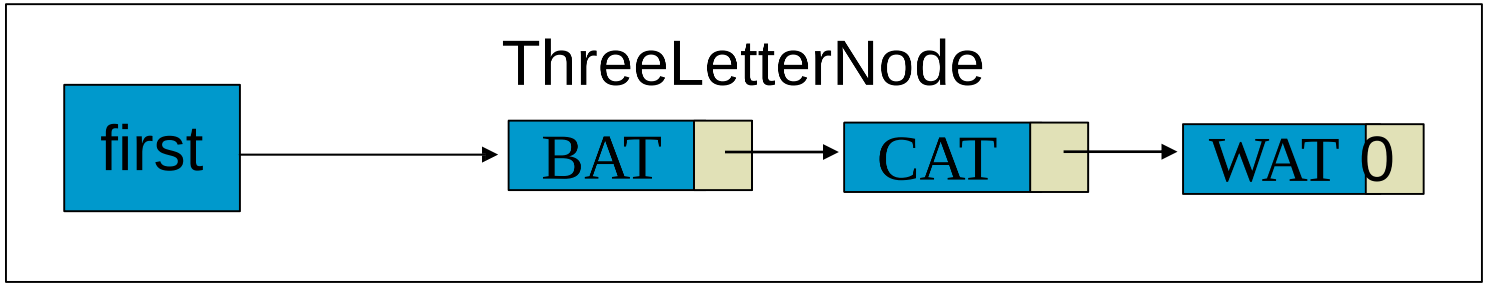
ThreeLetterChain



Actual relationship

•

ThreeLetterChain



Ideal relationship

Program 4.2: Nested Classes

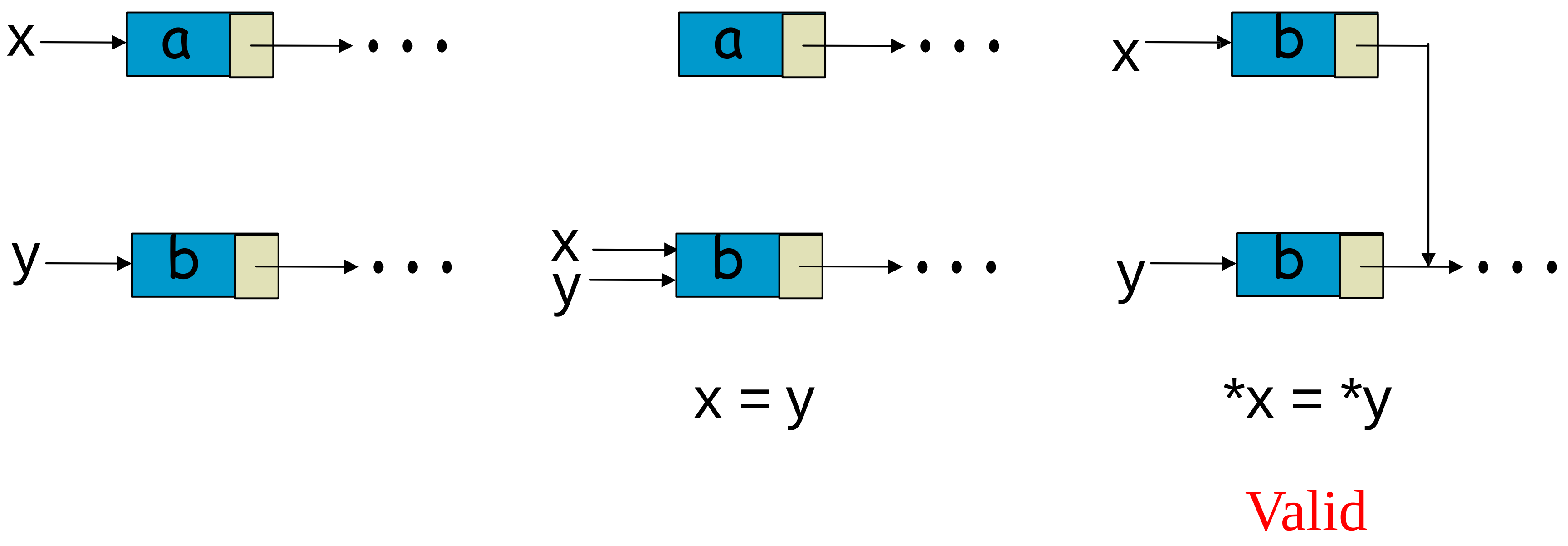
- Nested classes
  - One class is defined inside the definition of another.
- Class *ThreeLetterNode* is defined inside the private portion of the definition of class *ThreeLetterList*
  - This ensures that ThreeLetterNode objects cannot be accessed outside class ThreeLetterList

Program 4.2: Nested Classes (Cont'd)

```
class ThreeLetterList {
public:
// List Manipulation operations
.
.
private:
    class ThreeLetterNode {// nested class
    public:
        char data[3];
        ThreeLetterNode *link;
    };
    ThreeLetterNode *first;
};
```

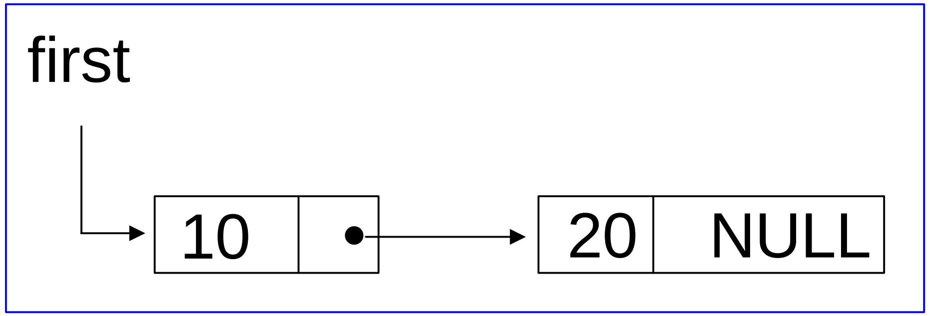
Pointer Manipulation in C++

- Two pointer variables of the same type can be compared.
  - The expressions x == y, x != y, x == 0 are all valid



Program 4.3: Create a Two-Node List

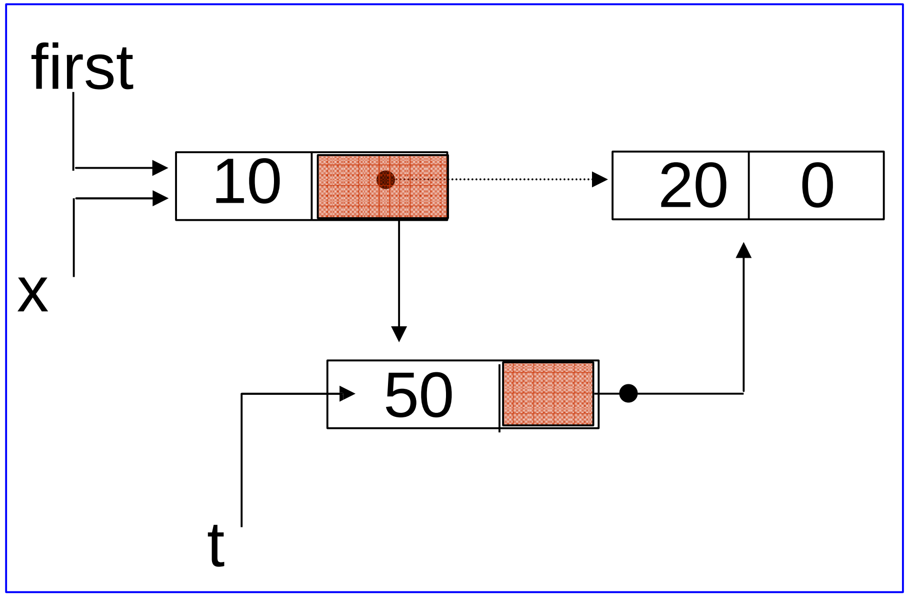
```
void List::Create2( )
{
    // create a linked list with two nodes
    first = new ListNode(10);
    first->link=new ListNode(20);
}
ListNode::ListNode(int element=0)
{
    data=element;
    link=0;
}
```



List Insertion

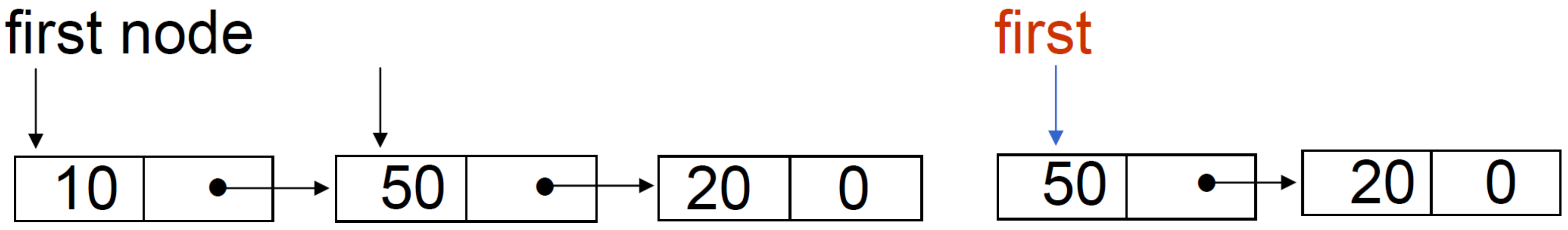
- Insert a node after a specific node (after the first node 10 in the previous case)

```
void List::Insert50 (ListNode *x)
{
/* insert a new node with data = 50 into the list */
   ListNode *t = new ListNode(50);
   if (!first)
   {
     first=t;  return;
   }
   //insert after x
   t->link=x->link;
   x->link=t;
}
```



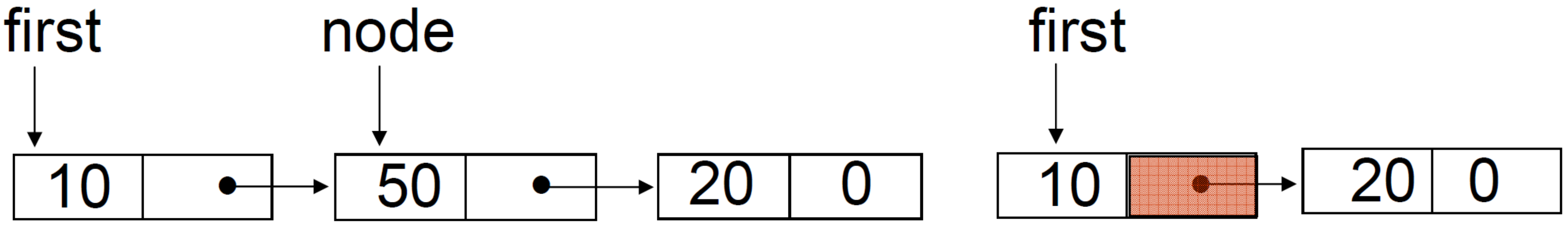
List Deletion

- Delete the first node



(a) before deletion (b) after deletion

- Delete node other than the first node



Outline

- Singly Linked Lists and Chains
- Representing Chains in C++
- The Template Class Chain
- Circular Lists
- Linked Stacks and Queues
- Polynomials
- Equivalence Classes
- Sparse Matrices
- Doubly Linked Lists

Program 4.6: Template Definition of Chains

```
enum Boolean {FALSE, TRUE};
template <class Type> class List;
template <class Type> class ListIterator;
template <class Type> class ListNode {
  friend class List<Type>;
  friend class ListIterator <Type>;
  private:
    Type data;
    ListNode *link;
};
template <class Type> class List {
friend class ListIterator <Type>;
public:
  List() {first = 0;};
  // List manipulation operations
   .
   .
private:
  ListNode <Type> *first;
};
```

Program 4.6: Template Definition of Chains (Cont’d)

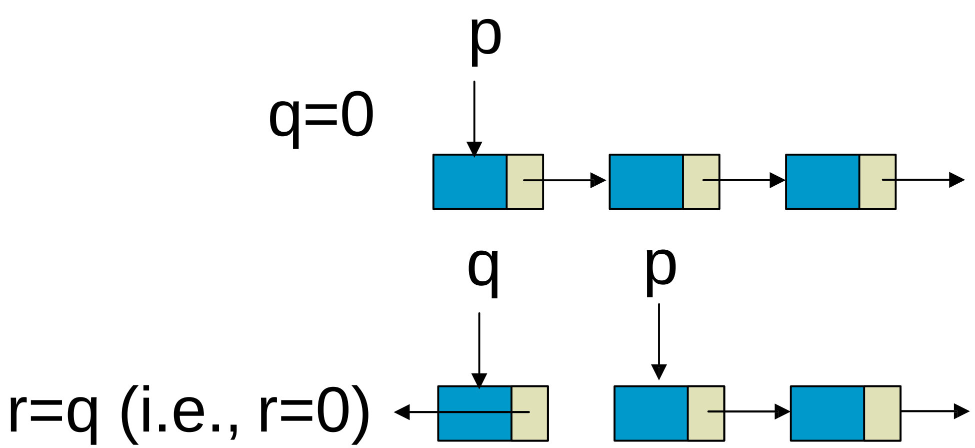
```
template <class Type> class ListIterator {
public:
   ListIterator(const List<Type> &l): list(l), current(l.first)
   {};
   Boolean NotNull();
   Boolean NextNotNull();
   Type * First();
   Type * Next();
Private:
   const List<Type>& list; // refers to an existing list
   ListNode<Type>* current; // points to a node in list
};
```

Program 4.11: Attaching a Node to the End of a List

```
Template <class Type>
Void List<Type>::Attach(Type k)
{
   ListNode<Type>*newnode = new ListNode<Type>(k);
   if (first == 0) first = last =newnode;
   else {
       last->link = newnode;
       last = newnode;
   }
};
```

Program 4.12: Inverting a list

```
template <class Type>  void List<Type>:: Invert()
// A chain x is inverted so that if x=(a1,…an)
// then, after execution, x=(an,…,a1)
{
    ListNode<Type>*p = first,*q=0; //q trails p
    while (p)
    {
       ListNode<Type> *r=q;
       q=p; //r trails q
       p=p->link; //p moves to next node
       q->link=r; //link q to preceding node
    }
    first=q;
};
```

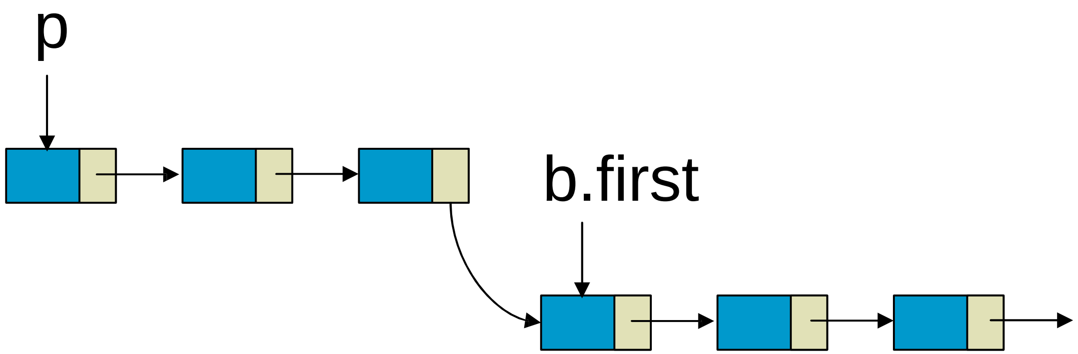


Program 4.13: Concatenating Two Chains

```
Template <class Type>
void List<Type>:: Concatenate(List<Type> b)
// this = (a1, …, am) and b = (b1, …, bn) m, n ≥ ,
// produces the new chain z = (a1, …, am, b1, bn) in this.
{
    if(!first) {
    first = b.first;
    return;
    }
    if (b.first) {
      for (ListNode<Type> *p = first; p->link; p = p->link);
      // no body
      p->link = b.first;
    }
}
```

first==null

first!=null && b!=null



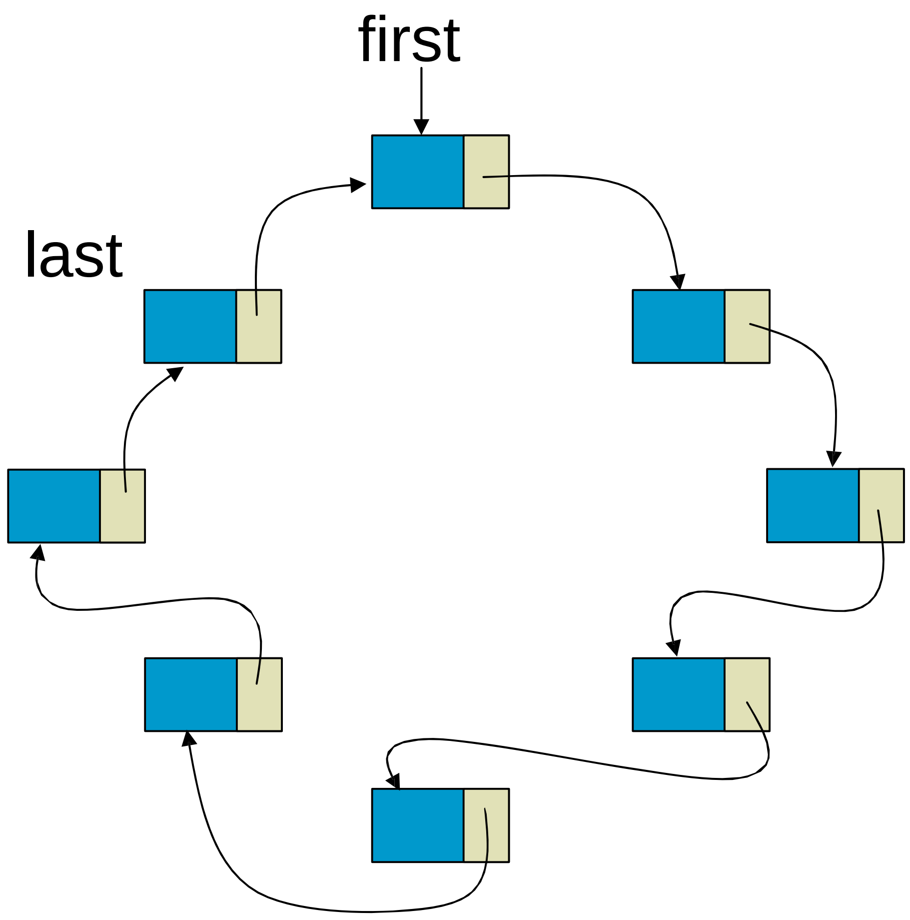
List Destructor

```
Template <class Type>
List<Type>::~List()
// Free all nodes in the chain
{
    ListNode<Type>* next;
    for ( ; first; first = next) {
        next = first->link;
        delete first;
    }
}
```

Outline

- Singly Linked Lists and Chains
- Representing Chains in C++
- The Template Class Chain
- Circular Lists
- Linked Stacks and Queues
- Polynomials
- Equivalence Classes
- Sparse Matrices
- Doubly Linked Lists

Diagram of a Circular List



Last->link = first

Inserting at the Front of a Circular List

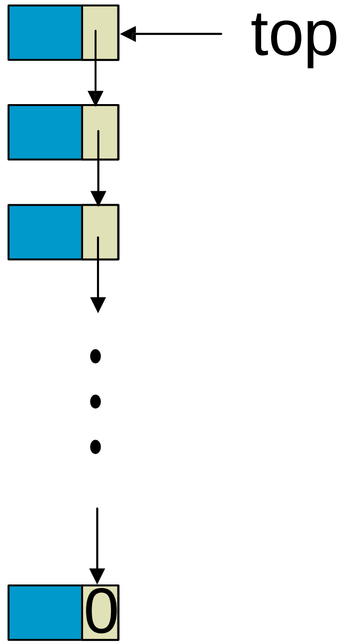
```
// Insert the element e at the "front" of the circular
// list where last points to the last node in the list

 if (!first)
    {
      last = newNode;
      newNode→link = newNode;
    }
   newNode→link = last→link;
   last→link = newNode;
```

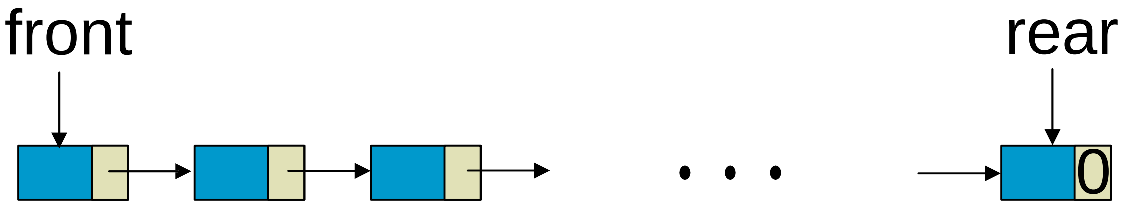
Outline

- Singly Linked Lists and Chains
- Representing Chains in C++
- The Template Class Chain
- Circular Lists
- Linked Stacks and Queues
- Polynomials
- Equivalence Classes
- Sparse Matrices
- Doubly Linked Lists

Linked Stacks and Queues



<u>Linked Stack</u>



<u>Linked Queue</u>

Adding and Deleting to a Linked Stack

- Adding

```
top = new ChainNode<T>(e,top);
```

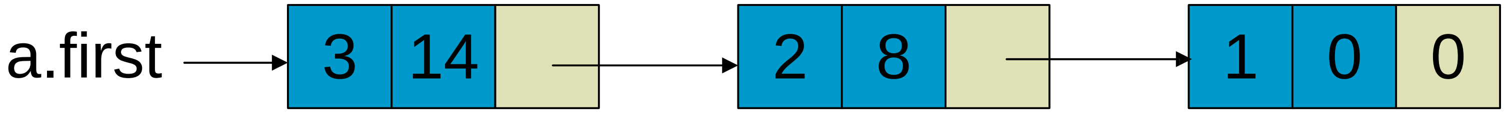
- Deleting

```
if (!empty)
{
   // Delete top node from the stack
   top = top→link; // remove top node
   delete delNode; // free the node
}
throw "Stack is empty. Cannot delete."
```

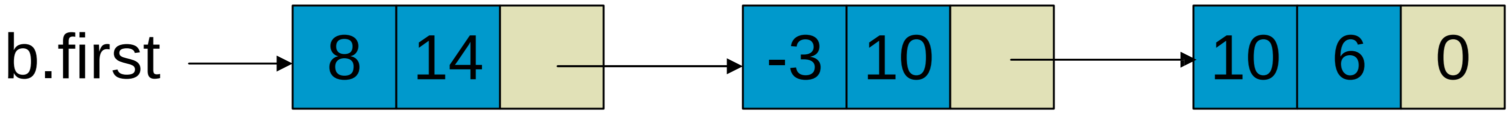
Outline

- Singly Linked Lists and Chains
- Representing Chains in C++
- The Template Class Chain
- Circular Lists
- Linked Stacks and Queues
- Polynomials
- Equivalence Classes
- Sparse Matrices
- Doubly Linked Lists

Revisit Polynomials



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



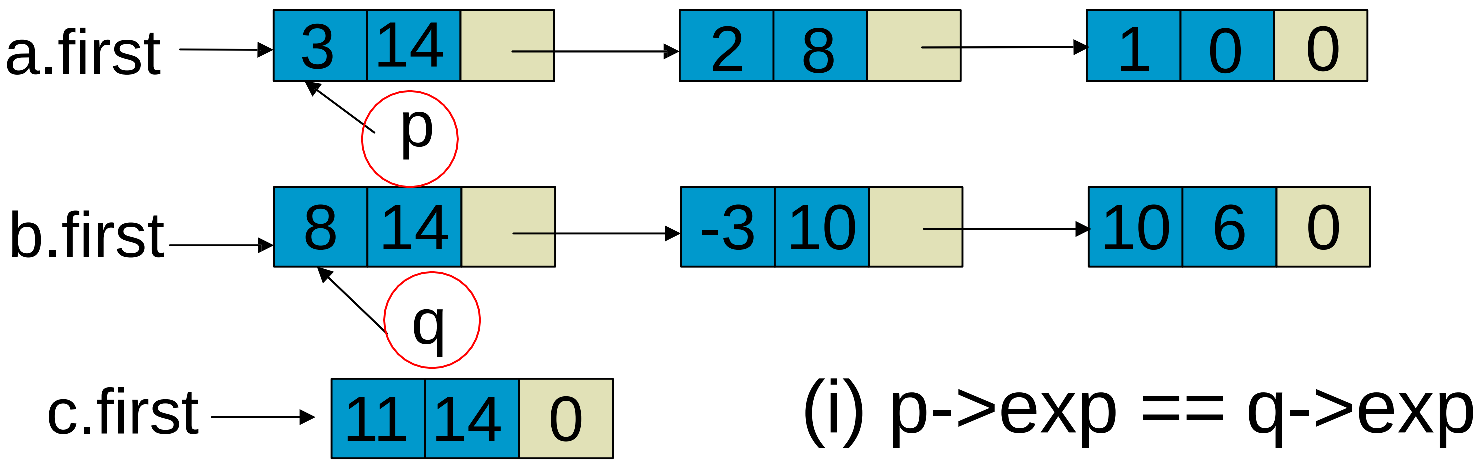
$$b = 8\,x^{14} - 3x^{10} + 10x^{6}$$

Program 4.20: Polynomial Class Definition

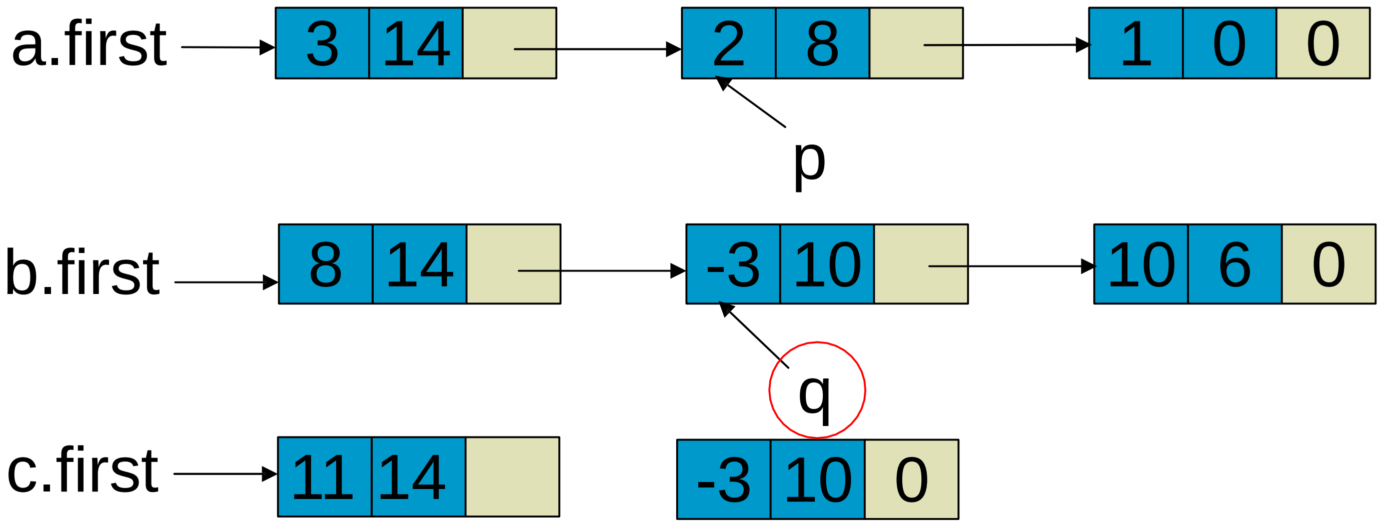
```
struct Term
/* all members of Terms are public by default */
{
    int coef;       // coefficient
    int exp;        // exponent
    void Init(int c, int e)
    {coef = c; exp = e;};
};
class Polynomial
{
    friend Polynomial operator+(const  Polynomial&, const Polynomial&);
    private:
       List<Term> poly;
};
```

Operating on Polynomials

- With linked lists, it is much easier to perform operations on polynomials such as adding and deleting.
  - E.g., adding two polynomials *a* and *b*

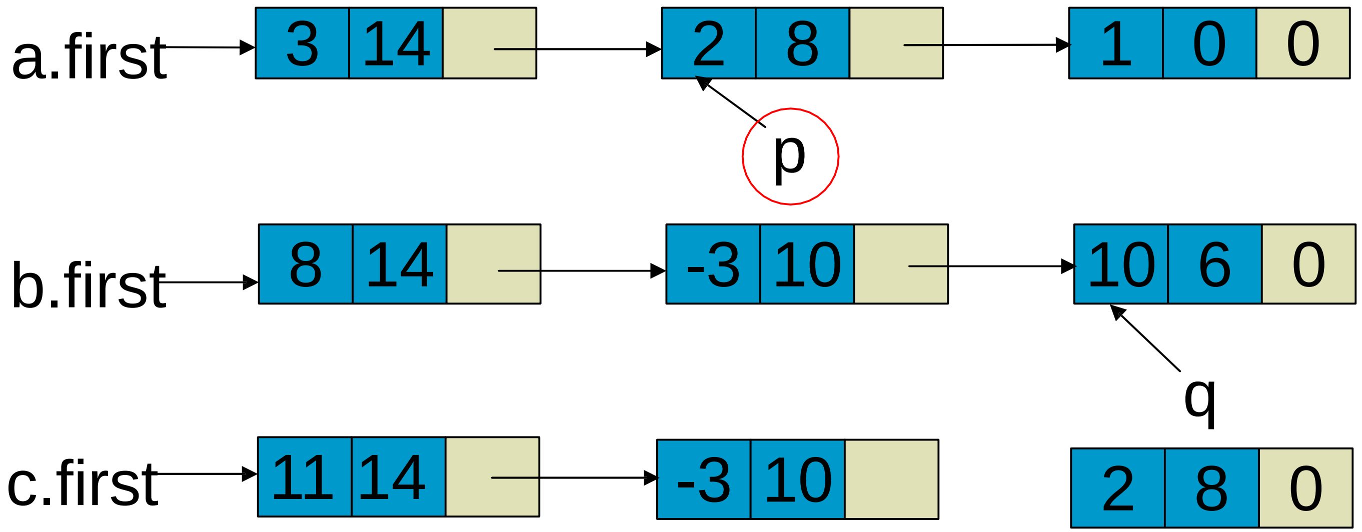


Operating on Polynomials (Cont'd)



(ii) p->exp < q->exp

Operating on Polynomials (Cont'd)



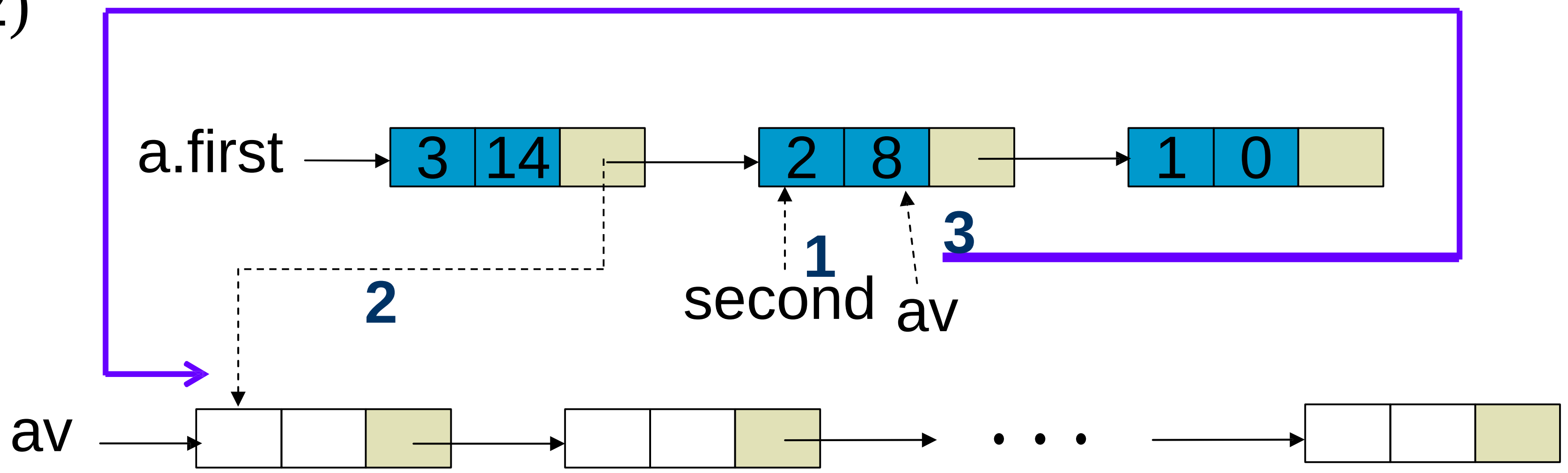
(iii) p->exp > q->exp

Free Pool

- When items are created and deleted constantly, it is more efficient to have a circular list to contain all available items.
  - To reduce the times of creating and deleting objects
- When an item is needed, the free pool is checked to see if there is any item available.
  - If yes, then an item is retrieved and assigned for use.
- If the list is empty, then either we stop allocating new items or use **new** to create more items for use.

Program 4.25 with some modifications

```
template <class KeyType>
void CircList<Type>::~CirList()
// Erase the circular list pointed to by first
{
   if( first )
   {
     ListNode* second=first->link; //(1)
     first->link=av; //(2)
     av=second; //(3)
     first=0; //(4)
   }
}
```



Program 4.23

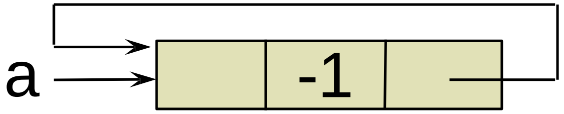
```
template <class Type>
ListNode <Type>* CircList::GetNode()
// Provide a node for use
{
    ListNode <Type> *x;
    if( !av )
        x = new ListNode<Type>;
    else
    {
        x = av;
        av = av -> link;
    }
    return x;
}
```

Program 4.24

```
template <class Type>
void
    CircList<Type>::RetNode( ListNode<  Type> *x
    //Free the node pointed to by x
    {
       x -> link = av;
       av = x;
    }
```

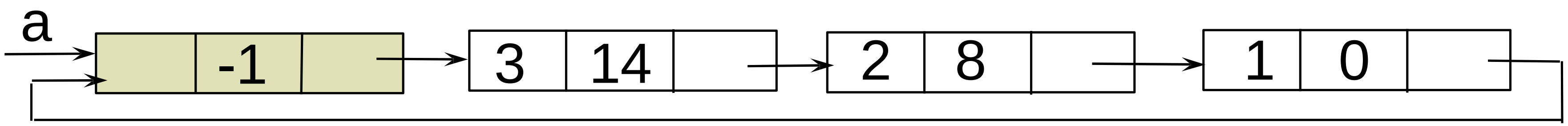
Head Node

- Represent polynomial as circular list
  - Zero



Zero polynomial

  - Others



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

Outline

- Singly Linked Lists and Chains
- Representing Chains in C++
- The Template Class Chain
- Circular Lists
- Linked Stacks and Queues
- Polynomials
- Equivalence Classes
- Sparse Matrices
- Doubly Linked Lists

Equivalence Relations

- For an arbitrary relation by the symbol $\equiv$ (equivalent to)
- Reflexive
  - If $x \equiv x$
- Symmetric
  - If $x \equiv y$, then $y \equiv x$
- Transitive
  - If $x \equiv y$ and $y \equiv z$, then $x \equiv z$
- A relation over a set *S*, is said to be an equivalence relation over *S* iff it is symmetric, reflexive, and transitive over *S*.

Examples

- The "equal to" (=) relationship is an equivalence relation since
  - x=x
  - x=y implies y=x
  - x=y and y=z implies x=z
- An equivalence relation is to partition the set *S* into equivalence classes such that two members x and y of *S* are in the same equivalence iff $x \equiv y$
  - $0 \equiv 4$, $3 \equiv 1$, $6 \equiv 10$, $8 \equiv 9$, $7 \equiv 4$, $6 \equiv 8$, $3 \equiv 5$, $2 \equiv 11$, $11 \equiv 0$
  - Three equivalent classes are {0, 2, 4, 7, 11}; {1, 3, 5}; {6, 8, 9, 10}

A Rough Algorithm to Find Equivalence Classes

Phase 1

Phase 2

```
{
    initialize;
    while (there are more pairs) {
        read the next pair <i,j>;
        process this pair;
    }
    initialize the output;
    do {
        output a new equivalence class;
    } while (not done);
}
```

What kinds of data structures are adopted?

Program 4.27 Equivalence Algorithm

```
void equivalence()
{
  read n; // read in number of objects
  initialize seq to 0 and out to FASLE;
  while more pairs // input pairs
  {
     read the next pair (i,j);
     put j on seq[i] list;
     put i on seq[j] list;
  }
  for( i = 0; i < n; i++ )
     if( out[i] == FALSE){
       out[i] = TRUE;
       output the equivalence class that contains object i
     }
}// end of equivalence using transitivity
```

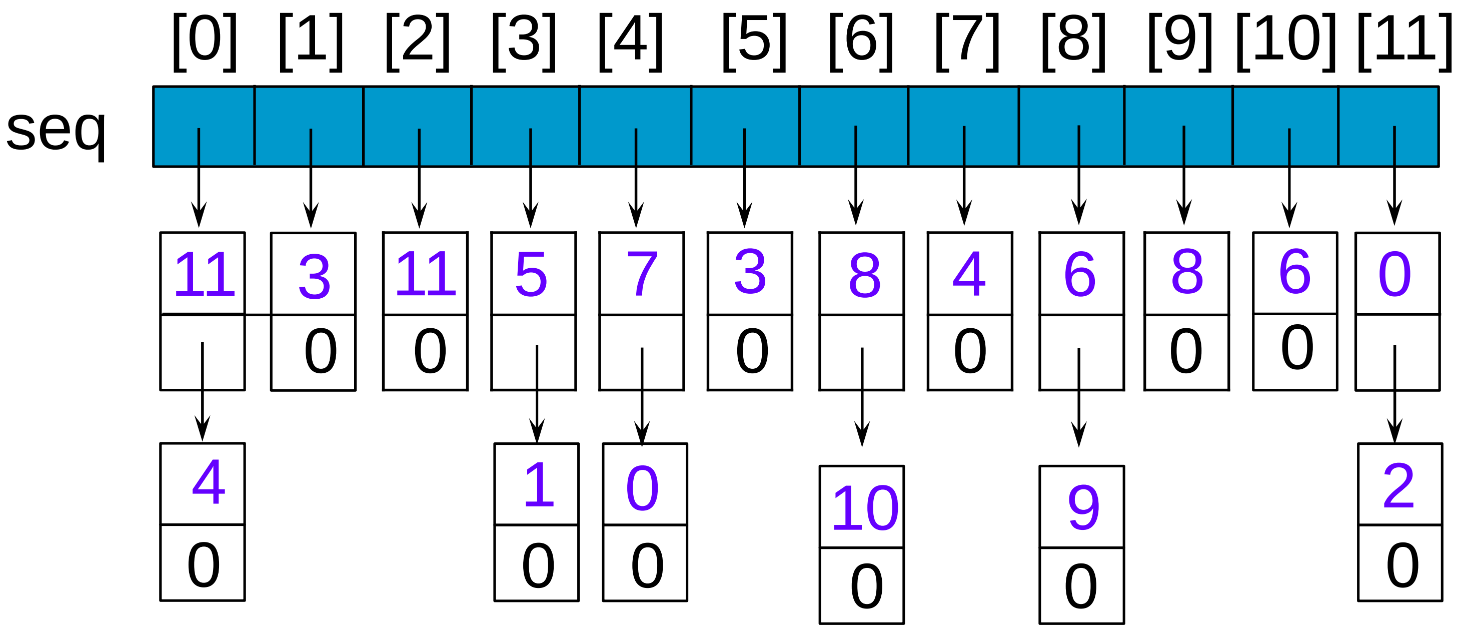
direct equivalence

Compute indirect equivalence using transitivity

Illustration

Lists after pairs have been input

0 ≡ 4
3 ≡ 1
6 ≡ 10
8 ≡ 9
7 ≡ 4
6 ≡ 8
3 ≡ 5
2 ≡ 11
11 ≡ 0



Program 4.28

```
enum Boolean { FALSE, TRUE };
class ListNode{
friend void equivalence();
private:
   int data;
   ListNode *link;
   ListNode(int);
};

typedef ListNode *ListNodePrt;
/* so we can create an array of pointers using new */
ListNode::ListNode(int d){
   data = d;
   link = 0;
}
```

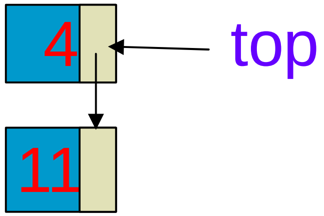
```
void equivalence()
/* Input the equivalence pairs an output the  equivalence classes */
{
   //”equiv.in” is the input file  ifstream inFile(“equiv.in”, ios::in);
   if(!inFile) {
        cerr << “Cannot open input file ” <<  endl;
   return;
}
   int i, j, n;
   inFile >> n; // read number of objects  // initialize seq and out
   ListNodePtr *seq = new ListNodePtr[n];  Boolean *out = new Boolean[n];
   for( i = 0; i < n ; i++){
       seq[i] = 0;
       out[i] = FALSE;
}
```

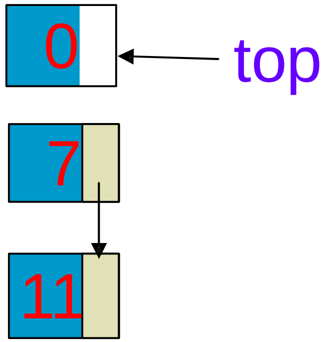
```
// Phase 1: input equivalence pairs
inFile >> i >> j ;
while( inFile.good()){ // check end of file
   ListNode *x = new ListNode(j);
   x->link = seq[i];
   seq[i] = x; // add j to seq[i]
   ListNode *y = new ListNode(i);
   y->link = seq[j];
   seq[j] = y; // add i to seq[i]
   inFile >> i >> j;
}
```

```
// Phase 2: output equivalence classes
for( i = 0; i < n; i++)
   if( out[i] == FALSE){ // needs to be output
   cout << endl << "A new class: " << i;
   out[i] = TRUE;
   ListNode *x = seq[i]; ListNode *top = 0;
   // init stack
      while(1){ // find rest of class
        while(x){ // process the list
          j = x->data;
          if( out[j] == FALSE ){
            cout << "," << j;
            out[j] = TRUE;
            ListNode *y = x -> link;
            x -> link = top;
            top = x;
            x = y;
          }
          else x = x->link;
        }// end of while(x)
        if( !top ) break;
        else{
         x = seq[top->data];
         top = top->link; // unstack
        }
      } // end of while(1)
   } // end of if( out[i] == FALSE)
   for( i = 0; i < n;  i++ )
      while( seq[i]){
        ListNode *delnode = seq[i];
        seq[i] = delnode->link;
        delete delnode;
      }
   }
   delete [] seq; delete [] out;
} // end of equivalence
```

push

pop





Outline

- Singly Linked Lists and Chains
- Representing Chains in C++
- The Template Class Chain
- Circular Lists
- Linked Stacks and Queues
- Polynomials
- Equivalence Classes
- Sparse Matrices
- Doubly Linked Lists

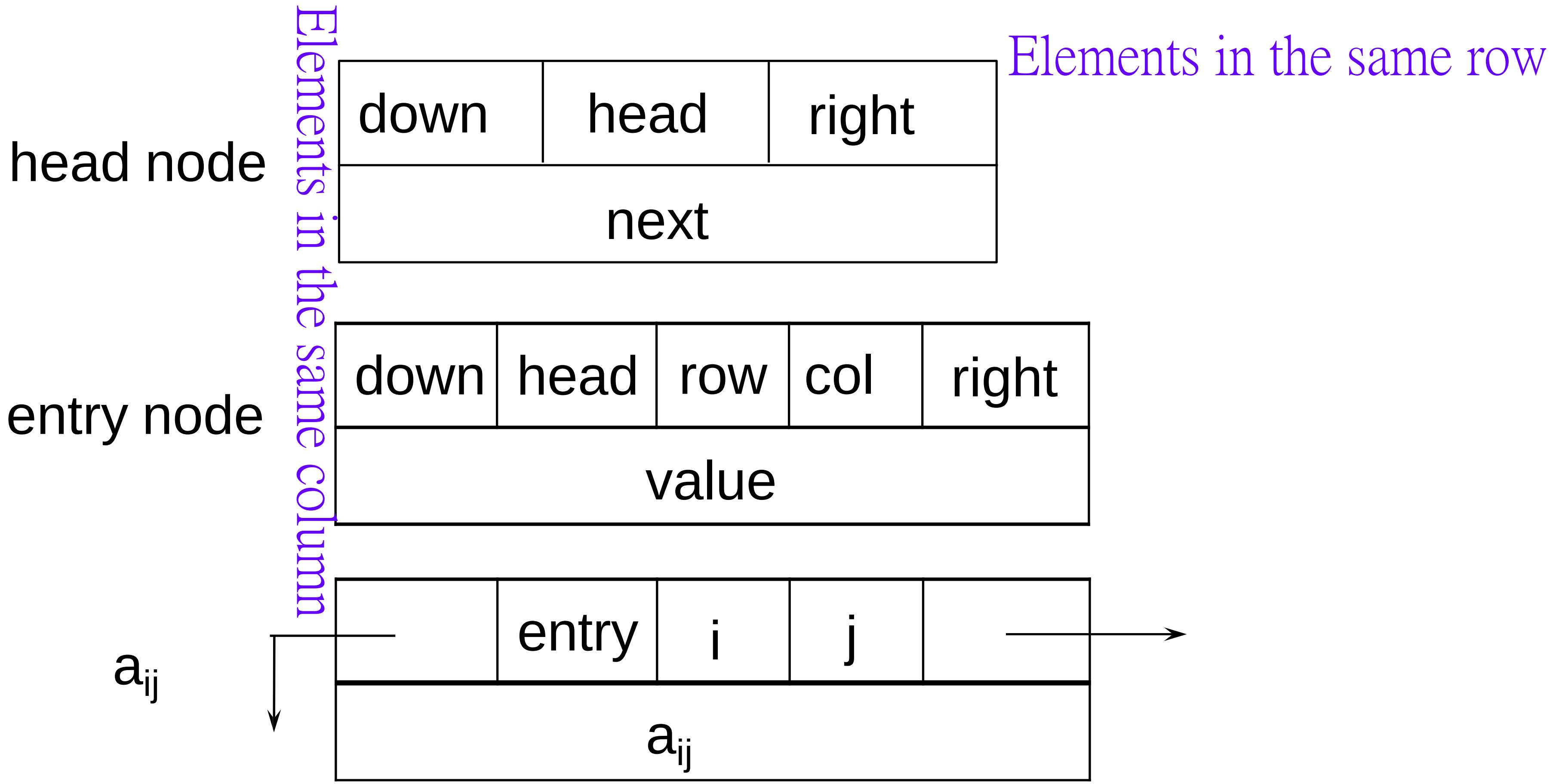
Sparse Matrices

- Inadequates of sequential schemes
  - Number 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

$$\begin{bmatrix} 0 & 0 & 11 & 0 \\ 12 & 0 & 0 & 0 \\ 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & 15 \end{bmatrix}$$

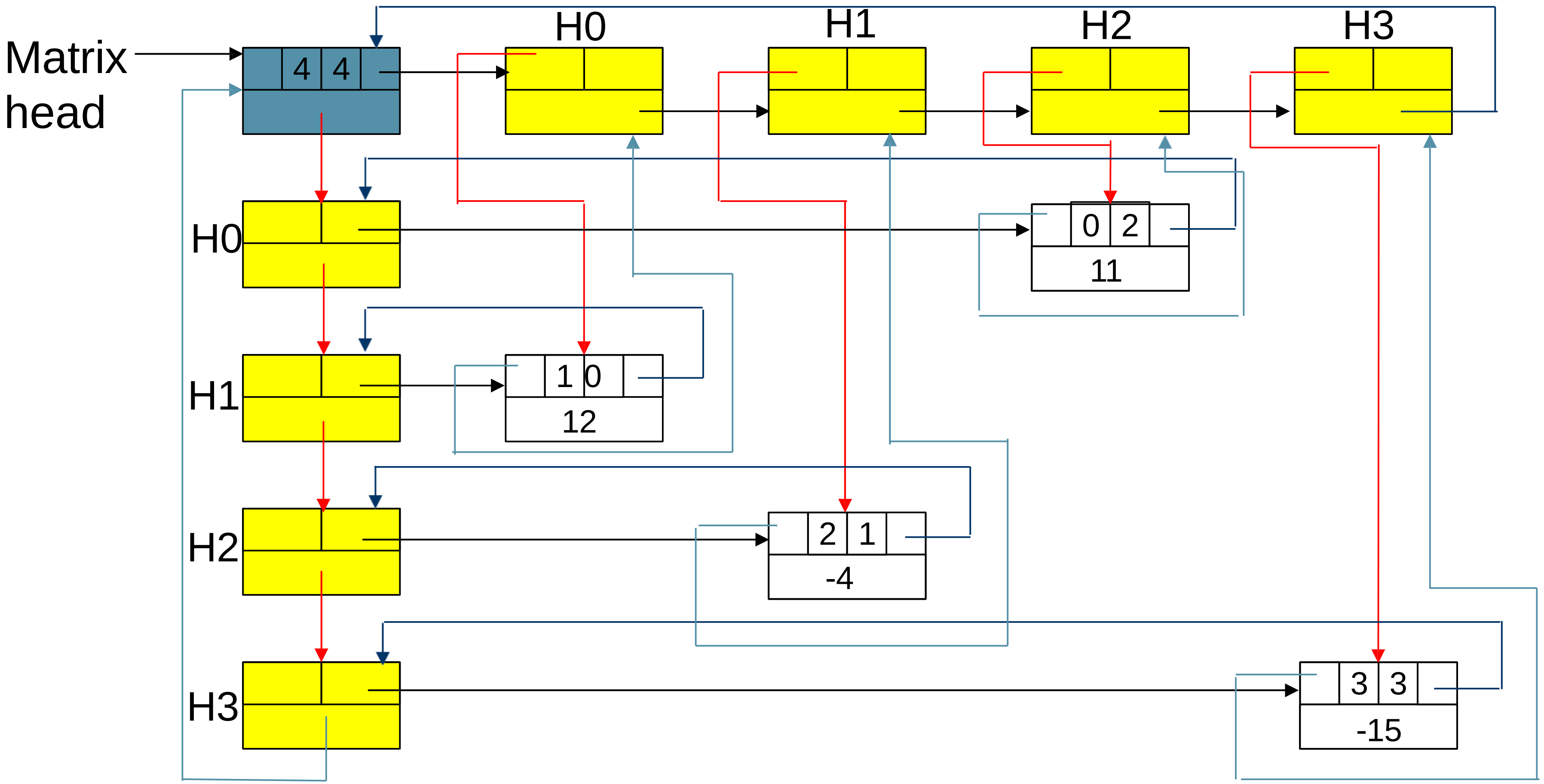
Revisit Sparse Matrices

- Number of head nodes = max{# of rows, # of columns}
- The field head is used to distinguish between head nodes and entry nodes



Linked Representation of A Sparse Matrix

- For an nxm sparse matrix with r nonzero terms, the number of nodes needed is max{n, m} + r + 1, total storage will be less than *nm* for sufficiently small *r*



Program 4.30

```
enum Boolean { FALSE, TRUE };
struct Triple { int value, row, col ; };
class Matrix ; //forward declaration  class MatrixNode {
    friend class Matrix ;
    //for reading in a matrix
    friend istream& operator>>(istream&, Matrix&) ;
    private:
        MatrixNode *down, *right ;
        Boolean head ;
        union { //anonymous union MatrixNode *next ;  Triple triple ;
        };
    MatrixNode(Boolean, Triple *) ;   //constructor
};
MatrixNode::MatrixNode(Boolean b, Triple *t)
  //constructor
{
    head = b ;
    if (b) { right = next = down = this;}   //row/column head node
    else triple = *t ;   //head node for list of  headnodes OR element node
}
```

```
typedef MatrixNode * MatrixNodePtr ;
//to allow subsequent creation of array of  pointers
class Matrix{
    friend istream& operator>>(istream&,  Matrix&) ;
    public:
        ~Matrix() ; //destructor
    private:
        MatrixNode *headnode ;
};
```

Outline

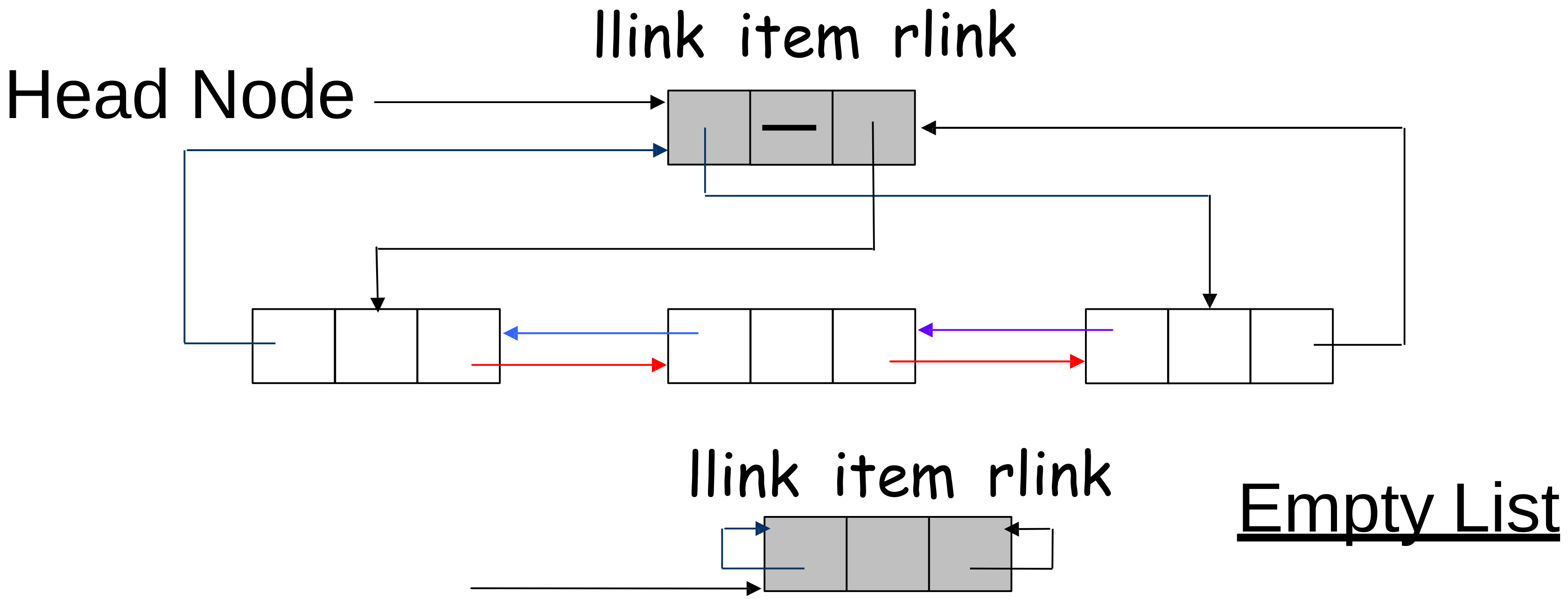
- Singly Linked Lists and Chains
- Representing Chains in C++
- The Template Class Chain
- Circular Lists
- Linked Stacks and Queues
- Polynomials
- Equivalence Classes
- Sparse Matrices
- Doubly Linked Lists

Program 4.30

```
class DblList ;
class DblListNode {
   friend class DblList ;
   private:
      int data ;
      DblListNode *llink, *rlink ;
};
class DblList {
   public:
      //List manipulation operations
   private:
      //points to head node  DblListNode *first ;
};
```

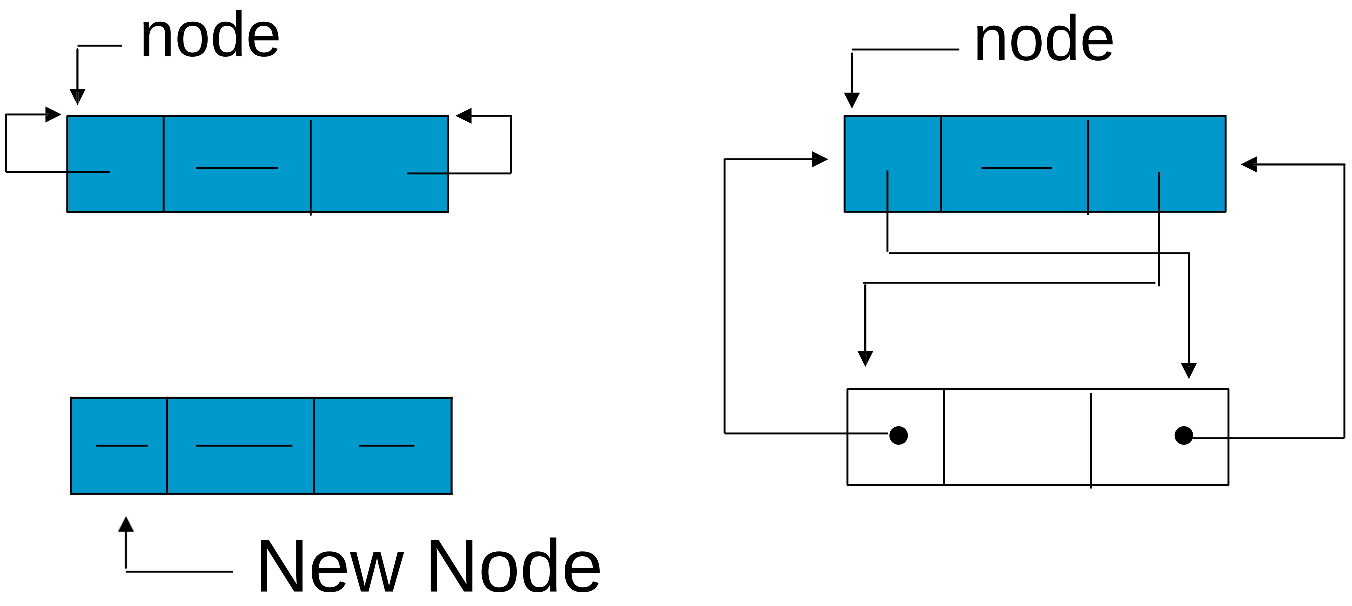
Doubly Linked List (contd.)

- A head node is also used in a doubly linked list to allow us to implement our operations more easily.



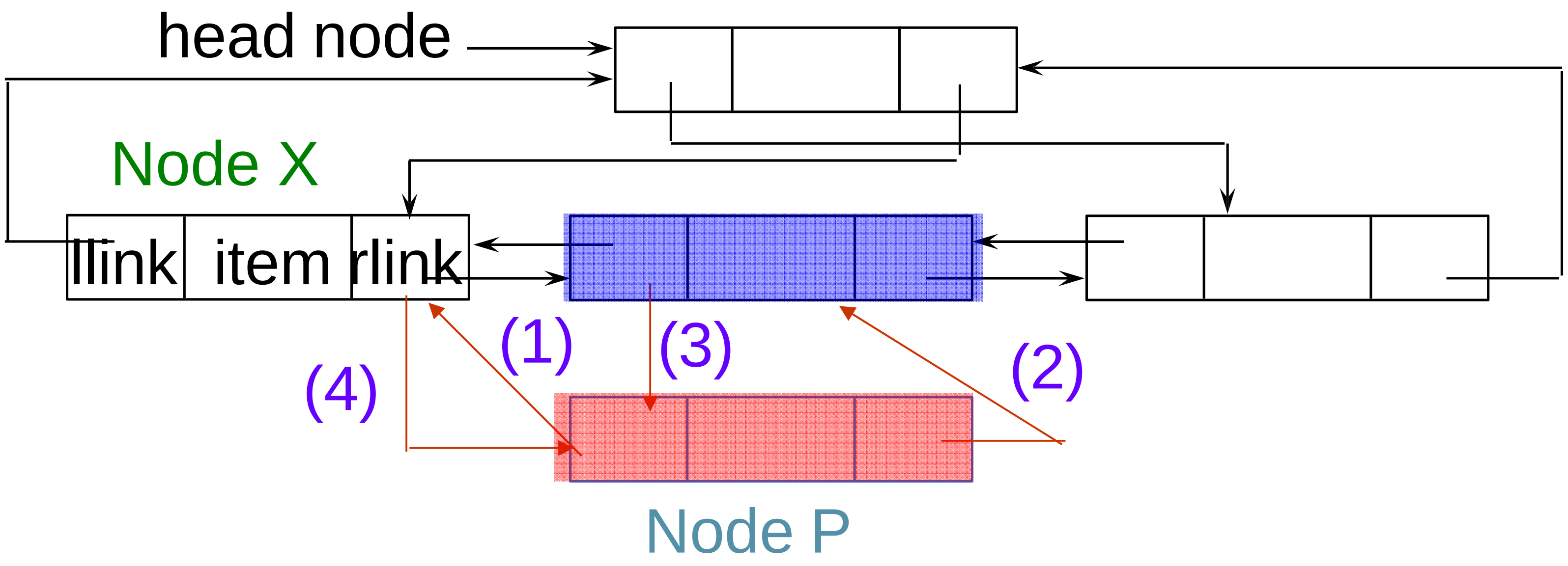
Doubly Linked List (contd.)

- Insertion into an empty doubly linked circular list



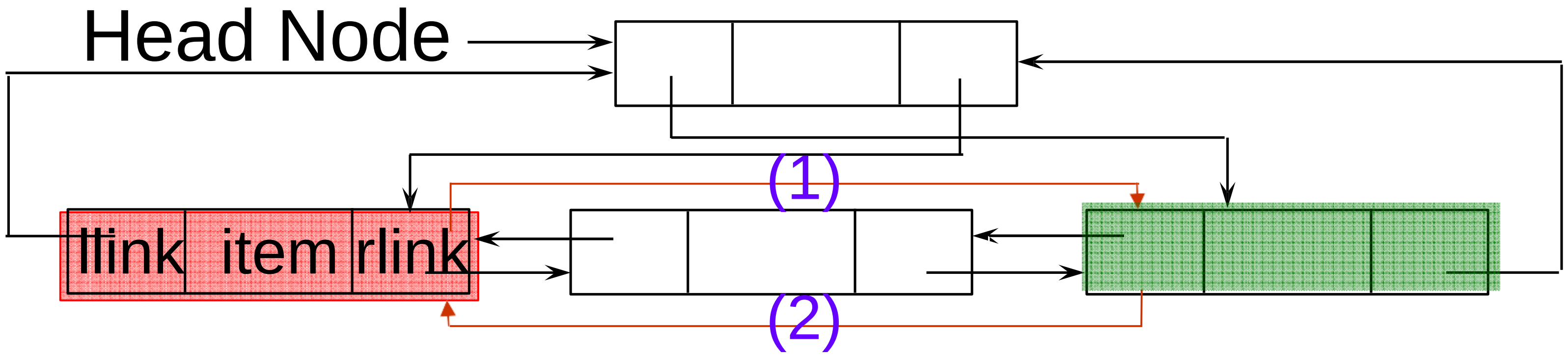
Insertion

```
void DblList::Insert(DblListNode *p,  DblListNode *x)
//insert node p to the right of node x
{
   p->llink = x ; //(1)
   p->rlink = x->rlink ; //(2)
   x->rlink->llink = p ; //(3)
   x->rlink = p ; //(4)
}
```



Deletion

```
void DblList::Delete(DblListNode *x) {
if(x == first) cerr <<"Deletion of head  node not permitted" <<endl ;
   else {
       x->llink->rlink = x->rlink ; //(1)
       x->rlink->llink = x->llink ;//(2)
       delete x ;
   }
}
```



Node X