Chapter 4
Linked Lists

4.1 Singly Linked lists Or Chains

The representation of simple data structure using an array and a sequential mapping has the property:

Successive nodes of the data object are stored at fixed distance apart.

This makes it easy to access an arbitrary node in O(1).

Disadvantage of sequential mapping:

insertion and deletion are expensive, e.g.:

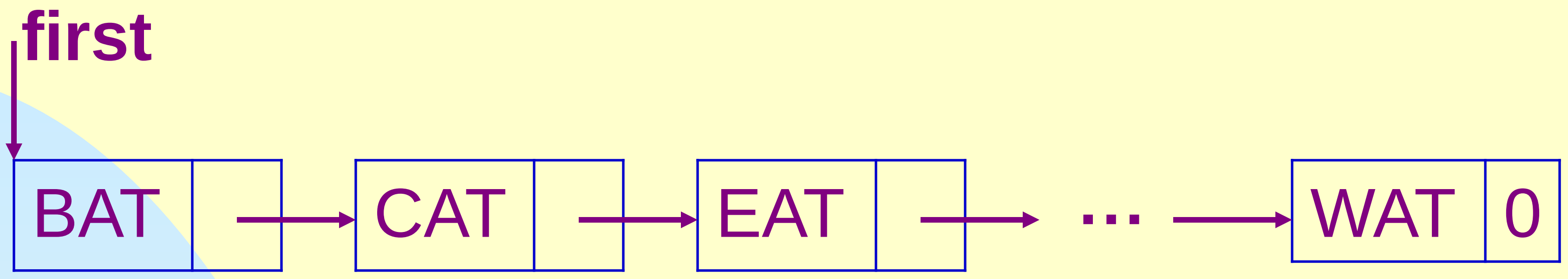
Insert “GAT” into or delete “LAT” from

(BAT, CAT, EAT, FAT, HAT, JAT, LAT, MAT, OAT, PAT, RAT, SAT, TAT,VAT,WAT)

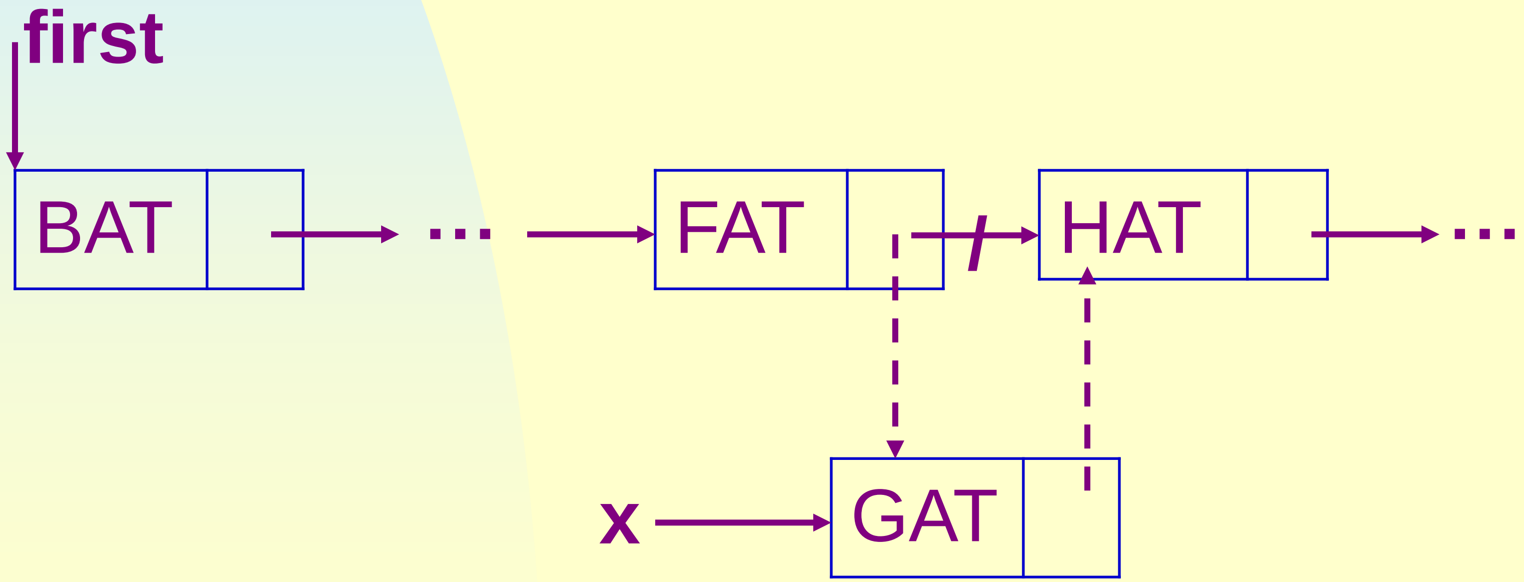
---need data movement.

Solution---linked representation:

- **items of a list may be placed anywhere in the memory.**
- **Associated with each item is a point (link) to the next item.**



In linked list, insertion (deletion) of arbitrary elements is much easier:



The above structures are called singly linked lists or chains in which each node has exactly one point field.

4.2 Representing Chains in C++

Assume a chain node is defined as:

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

```
ChainNode *f;
```

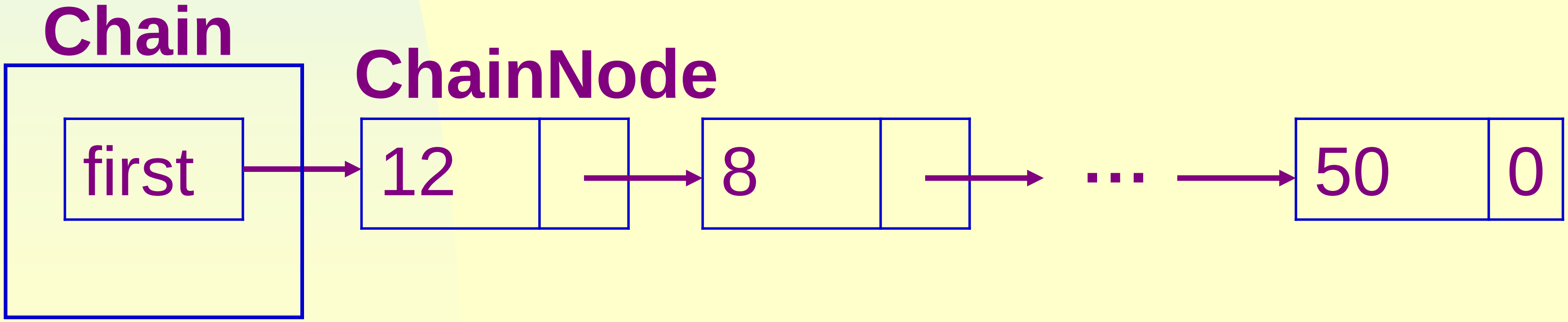
```
f→data
```

will cause a compiler error because a private data member cannot be accessed from outside of the object.

Definition: 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.

A composite of two classes: ChainNode and Chain.

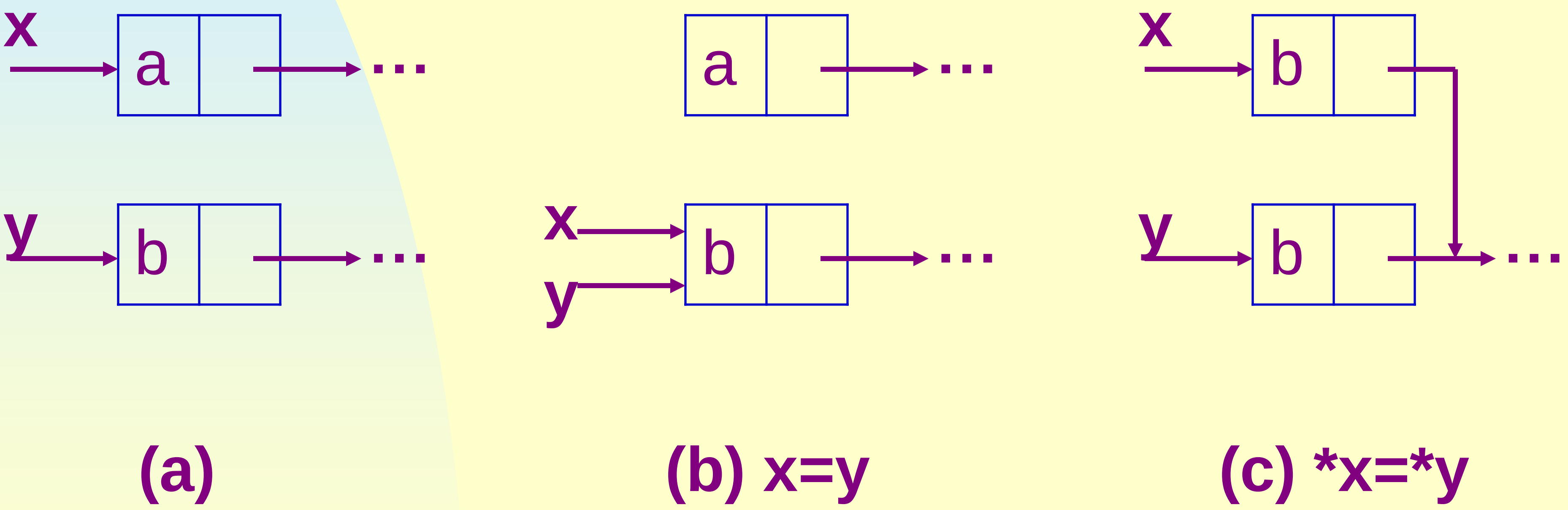
Chain HAS-A ChainNode.



```
class Chain;  // forward declaration
class ChainNode {
friend class Chain; // to make functions of Chain be able to
                      // access private data members of ChainNode
Public:
    ChainNode(int element = 0, ChainNode* next = 0)
        {data = element; link = next;}
private:
    int data;
    ChainNode *link;
};
class Chain {
public:
    // Chain manipulation operations
…
private:
    ChainNode *first;
};
```

Null pointer constant 0 is used to indicate no node.

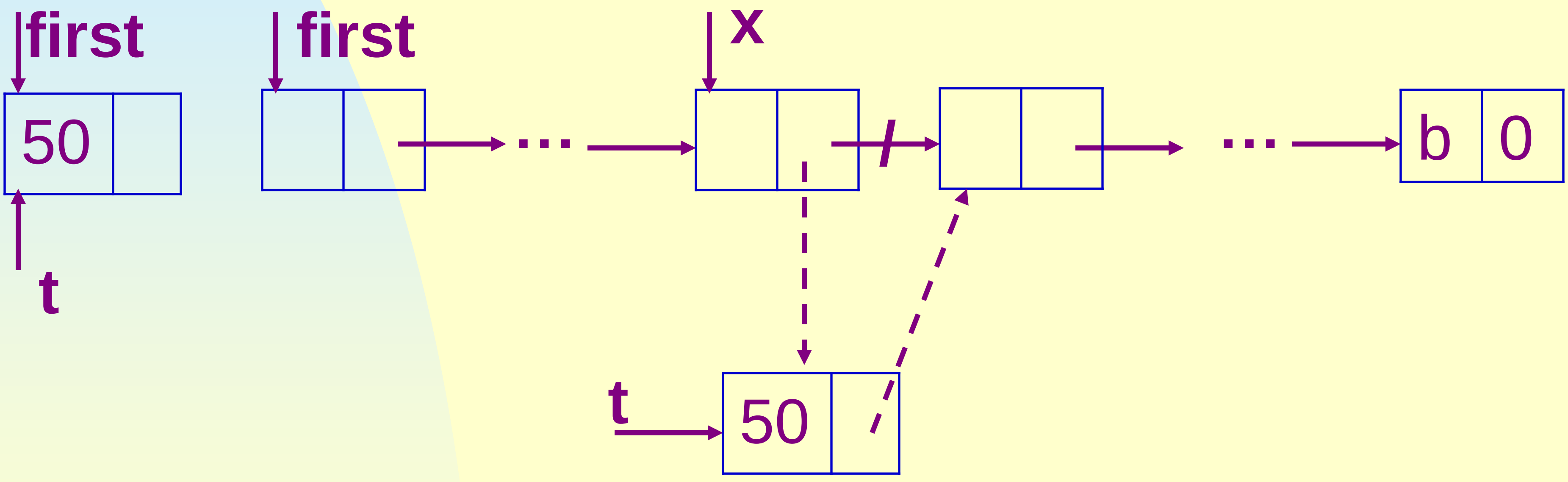
Pointer manipulation in C++:



(a) **(b) x=y** **(c) *x=*y**

Chain manipulation:

Example 4.3 insert a node with data field 50 following the node x.



(a) first=0

(b) First !=0

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

Exercises: P183-1,2

4.3 The Template Class Chain

We shall enhance the chain class of the previous section to make it more reusable.

4.3.1 Implementing Chains with Templates

```
template <class T> class Chain;  // forward declaration

template <class T>
class ChainNode {
friend class Chain<T>;
public:
    ChainNode(T element, ChainNode* next = 0)
        { data = element; link = next;}
private:
    T data;
    ChainNode *link;
};
```

```
template <class T>
class Chain {
public:
    Chain() { first=0;}; // constructor initializing first to 0
    // Chain manipulation operations
…
private:
    ChainNode<T> *first;
};
```

A empty chain of integers intchain would be defined as:

```
    Chain<int> intchain ;
```

4.3.2 Chain Iterators

A **container class** is a class that represents a data structure that contains or stores a number of data objects.

An **iterator** is an object that is used to access the elements of a container class one by one.

Why we need an iterator?

Consider the following operations that might be performed on a container class C, all of whose elements are integers:

(1) Output all integers in C.

(2) Obtain the sum, maximum, minimum, mean, median of all integers in C.

(3) Obtain the integer x from C such that f(x) is maximum.

……

These operations have to be implemented as member functions of C to access its private data members.

Consider the container class Chain<T>, there are, however, some drawbacks to this:

(1) All operations of Chain<T> should preferably be independent of the type of object to which T is initialized. However, operations that make sense for one instantiation of T may not for another instantiation.

(2) The number of operations of Chain<T> can become too large.

(3) Even if it is acceptable to add member functions, the user would have to learn how to sequence through the container class.

These suggest that container class be equipped with iterators that provide systematic access to the elements of the object.

User can employ these iterators to implement their own functions depending upon the particular application.

Typically, an iterator is implemented as a nested class of the container class.

A forward Iterator for Chain

A forward Iterator class for Chain may be implemented as in the next slides, and it is required that ChainIterator be a **public nested member class** of Chain.

```
class ChainIterator {
public:
    // typedefs required by C++ omitted

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

    // dereferencing operators
    T& operator *() const { return current→data;}
    T* operator →() const { return &current→data;}
```

```
// increment
ChainIterator& operator ++() // preincrement
    {current = current→link; return *this;}
ChainIterator& operator ++(int) // postincrement
    {
        ChainIterator old = *this;
        current = current→link;
        return old;
     }
    // equality testing
    bool operator !=(const ChainIterator right) const
        { return current != right.current; }
    bool operator == (const ChainIterator right) const
        { return current == right.current; }

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

Additionally, we add the following public member functions to Chain:

```
ChainIterator begin() {return ChainIterator(first);}
ChainIterator end() {return ChainIterator(0);}
```

We may initialize an iterator object yi to the start of a chain of integers y using the statement:

```
Chain<int>::ChainIterator yi = y.begin();
```

And we may sum the elements in y using the statement:

```
sum = accumulate(y.begin(), y.end(), 0);
// note sum does not require access to private members
```

Exercises: P194-3, 4

4.3.3 Chain Operations

Operations provided in a reusable class should be enough but not too many.

Normally, include: constructor, destructor, operator=, operator==, operator>>, operator<<, etc.

A chain class should provide functions to **insert** and **delete** elements.

Another useful function is **reverse** that does an "in-place' reversal of the elements in a chain.

To be efficient, we add a private member last to Chain<T>, which points to the last node in the chain.

InsertBack

```
template <class T>
void Chain<T>::InsertBack(const T& e)
{
   if (first) { // nonempty chain
      last→link = new ChainNode<T>(e);
      last =last→link;
    }
   else first = last= new ChainNode<T>(e);
}
```

The complexity: O(1).

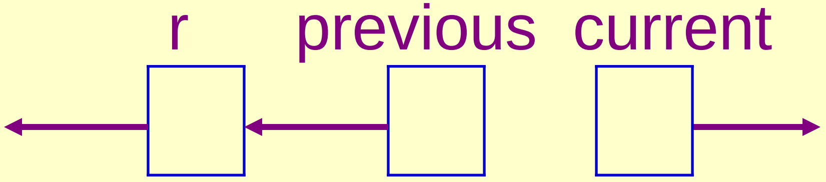
Concatenate

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

The complexity: O(1).

Reverse

```
template <class T>
void Chain<T>::Reverse()
{ // make ($a_1$,.., $a_n$) becomes ($a_n$,.., $a_1$).
   ChainNode<T> *current = first, *previous = 0;
   while (current) {
       ChainNode<T> *r = previous; // r trails previous
       previous = current;
       current = current→link;
       previous→link = r;   //
   }
   first = previous;
}
```

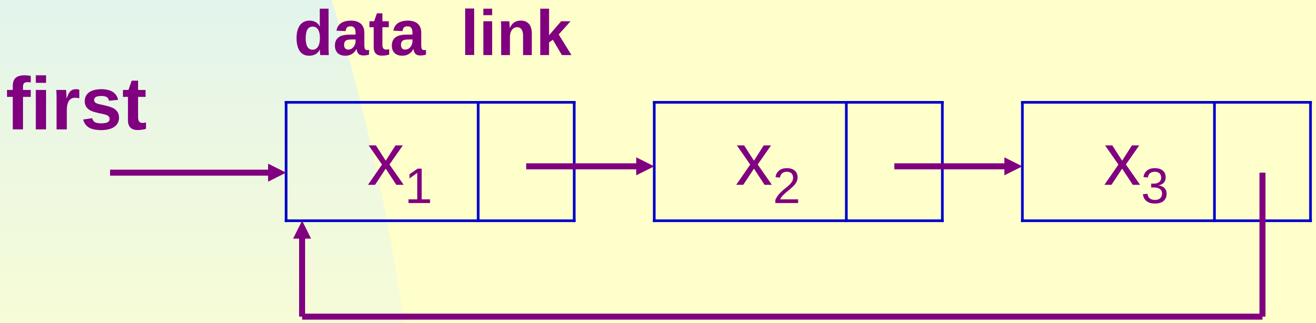


For a chain with m $\geq$ 1 nodes, the computing time of Reverse is O(m).

Exercises: P184-6

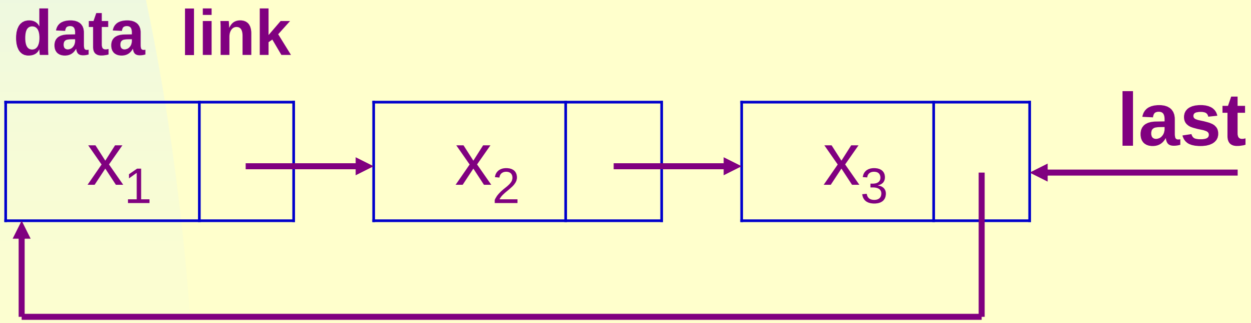
4.4 Circular Lists

A circular list can be obtained by making the last link field point to the first node of a chain.



Consider inserting a new node at the front of the above list. We need to change the link field of the node containing x_3.

It is more convenient if the access pointer points to the **last** rather than the first.



Now we can insert at the front in O(1):

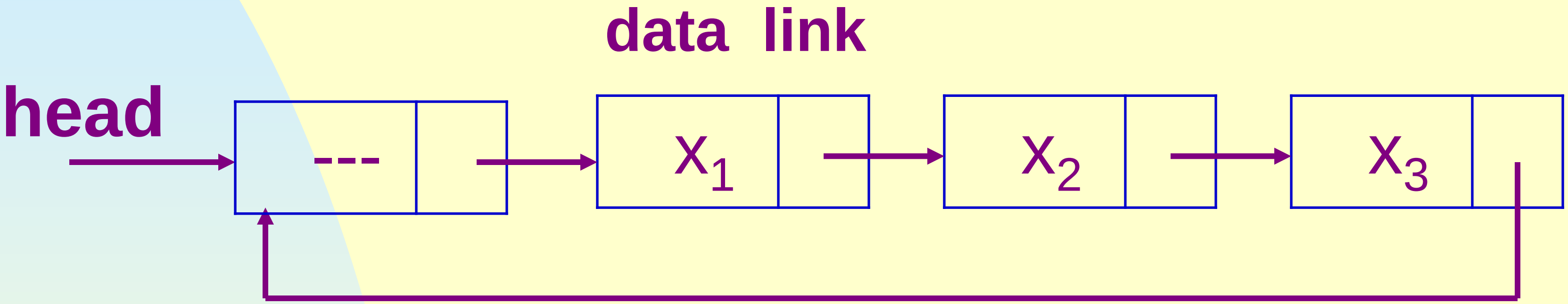
```
template <class T>
void CircularList<T>::InsertFront(const T& e)
{ // insert the element e at the “front” of the circular list *this,
 // where last points to the last node in the list.
   ChainNode<T>* newNode = new ChainNode<T>(e);
   if (last) { // nonempty list
       newNode→link = last→link;
       last→link = newNode;
    }
   else { last = newNode; newNode→link = newNode;}
}
```

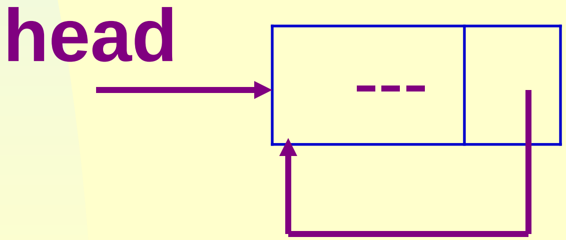
To insert at the back, we only need to add the statement

```
    last = newNode;
```

to the if clause of InsertFront, the complexity is still O(1).

To avoid handling empty list as a special case, introduce a dummy **head** node:.





empty circular list

4.5 Available Space lists

- the time of destructors for chains and circular lists is **linear** in the length of the chain or list.
- it may be reduced to O(1) if we maintain our own chain of free nodes.
- the available space list is pointed by **av**.
- **av** be a static class member of **CircularList<T>** of type **ChainNode<T> ***, initially, **av** = 0.
- only when the **av** list is empty do we need use **new.**

We shall now use CircularList<T>::GetNode instead of using new:

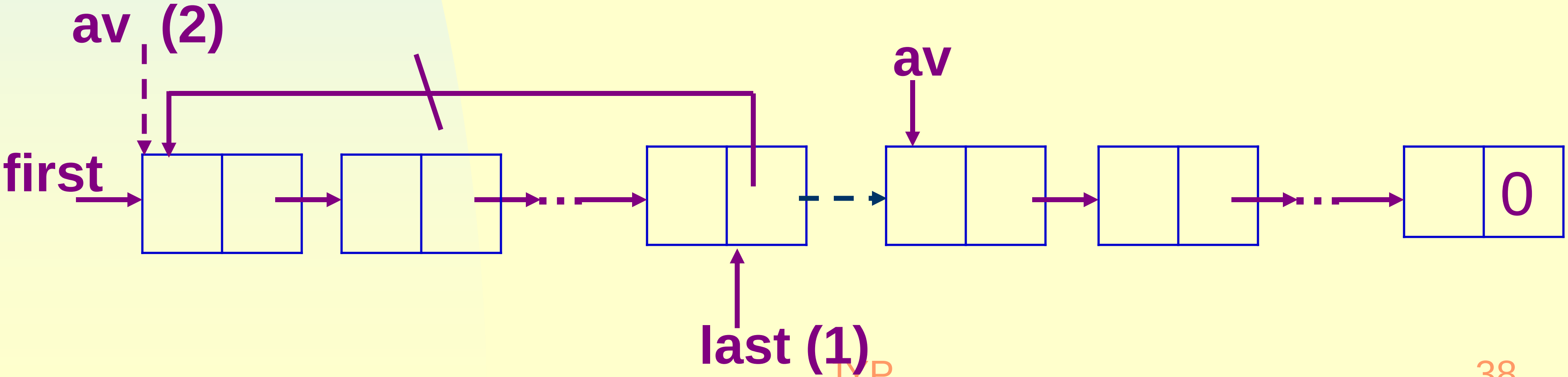
```
template <class T>
ChainNode<T>* CircularList<T>::GetNode( )
{ //provide a node for use
    ChainNode<T> * x;
    if (av) { x = av; av = av→link;}
    else x = new ChainNode<T>;
    return x;
}
```

And we use CircularList<T>::RetNode instead of using delete:

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

A circular list may be deleted in O(1):

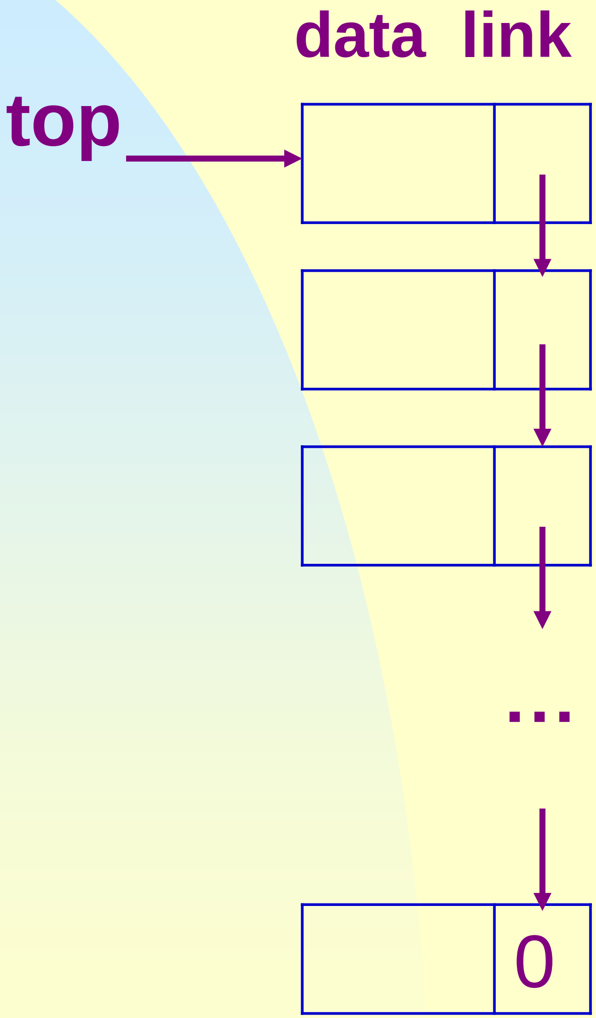
```
template <class T>
void CircularList<T>::~CircularList()
{ // delete the circular list.
   if (last) {  ChainNode <T> * first = last→link;
         last→link = av;   // (1)
         av = first; // (2)
         last = 0;
    }
}
```



A chain may be deleted in O(1) if we know its first and last nodes:

```
template <class T>
Chain<T>::~Chain()
{ // delete the chain
    if (first) {
        last→link = av;
        av = first;
        first = 0;
    }
}
```

4.6 Linked Stacks and Queues



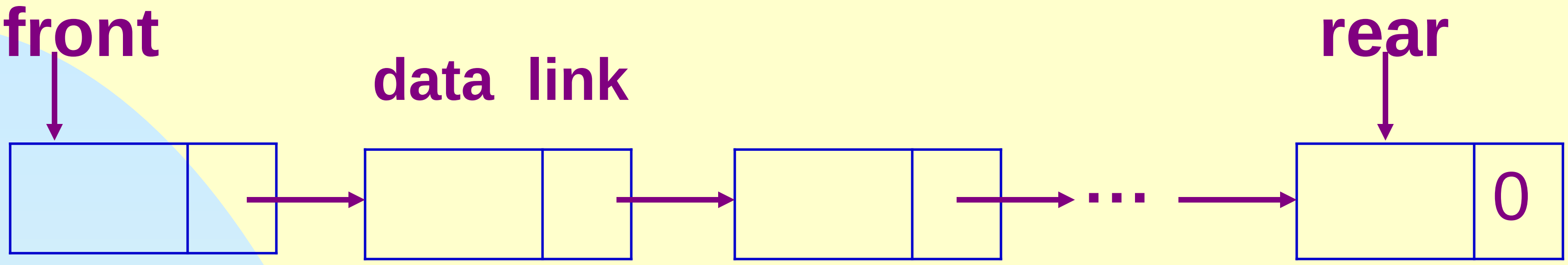
linked stack

Assume the LinkedStack class has been declared as friend of ChainNode<T>.

```
template <class T>
class LinkedStack {
public:
    LinkedStack() { top=0;}; // constructor initializing top to 0
    // LinkedStack manipulation operations
…
private:
    ChainNode<T> *top;
};
```

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

template <class T>
void LinkedStack<T>::Pop()
{ // delete top node from the stack.
    if (IsEmpty()) throw “Stack is empty. Cannot delete.”;
    ChainNode<T> * delNode = top;
    top = top→link;
    delete delNode;
}
```

The functions IsEmpty and Top are easy to implement, and are omitted.



linked queue

The functions of LinkedQueue are similar to those of LinkedStack, and are left as exercises.

Exercises: P201-2

4.7 Polynomials

4.7.1 Polynomial Representation

Since a polynomial is to be represented by a list, we say Polynomial is IS-IMPLEMENTED-IN-TERMS-OF List.

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 Type A object. ---Usually by declaring the Type B object as a data member of the Type A object.

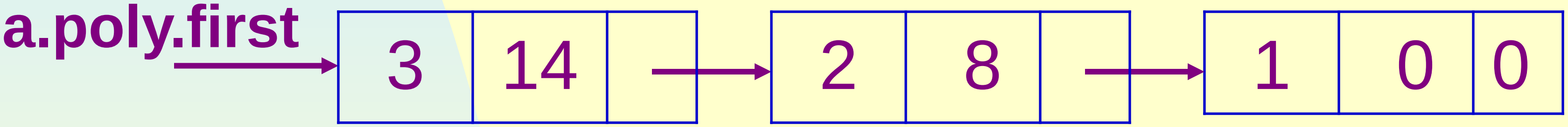
$$A(x) = a_m x^{em} + a_{m-1} x^{em-1} +, \ldots, + a_1 x^{e1}$$

$$\text{Where } a_i \neq 0, \quad e_m > e_{m-1} >, \ldots, e_1 \geq 0$$

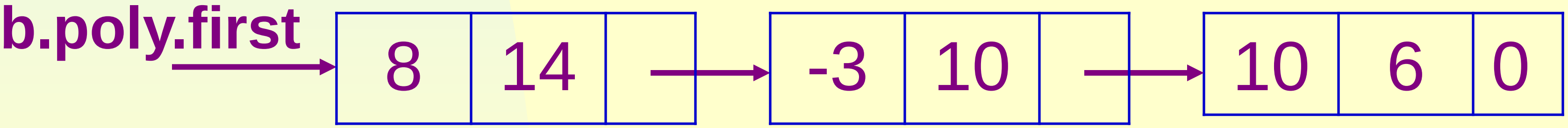
- Make the chain **poly** a data member of **Polynomial.**
- Each **ChainNode** will represent a term. The template T is instantiated to struct **Term**:

```
struct Term
{ // all members of Term are public by default
    int coef;
    int exp;
    Term Set(int c, int e) { coef=c; exp=e; return *this;};
};
```

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



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



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

4.7.2 Adding Polynomials

To add two polynomials a and b, use the chain iterators ai and bi to move along the terms of a and b.

```
Polynomia Polynomial::operaor+ (const Polynomial& b)
{ // *this (a) and b are added and the sum returned
  Term temp;
  Chain<Term>::ChainIterator ai = poly.begin(),
                             bi = b.poly.begin();
  Polynomial c;
```

```
while (ai != poly.end() && bi != b.poly.end()) { //not null
    if (ai→exp == bi→exp) {
        int sum = ai→coef + bi→coef;
        if (sum) c.poly.InsertBack(temp.Set(sum, bi→exp));
        ai++; bi++; // to next term
    }
    else if (ai→exp < bi→exp) {
            c.poly.InsertBack(temp.Set(bi→coef, bi→exp));
            bi++;  // next term of b
    }
    else {
            c.poly.InsertBack(temp.Set(ai→coef, ai→exp));
            ai++;  // next term of a
     }
  }
```

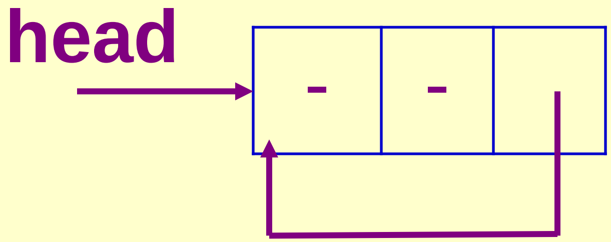
```
    while (ai != poly.end()) { // copy rest of a
        c.poly.InsertBack(temp.Set(ai→coef, ai→exp));
        ai++;
   }
    while (bi != b.poly.end()) { // copy rest of b
        c.poly.InsertBack(temp.Set(bi→coef, bi→exp));
       bi++;
     }
    return c;
}
```

Analysis:

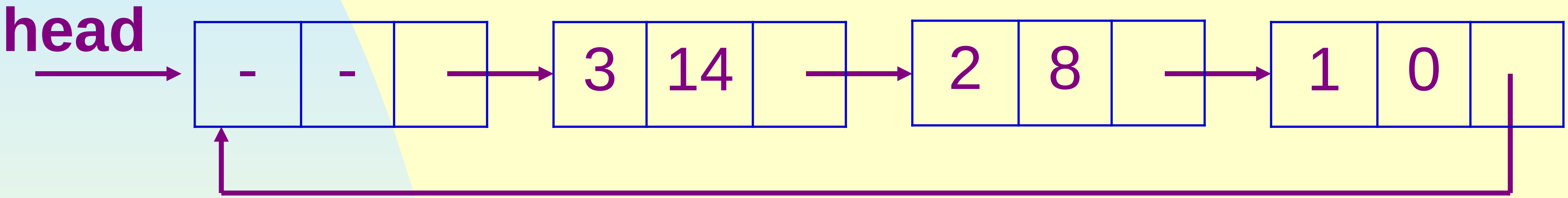
Assume a has m terms, b has n terms. The computing time is O(m+n).

4.7.3 Circular List Representation of Polynomials

Polynomials represented by circular lists with head node are as in the next slide:



(a) Zero polynomial



(b) $3x^{14} + 2x^{8} + 1$

Adding circularly represented polynomials

- The exp of the head node is set to –1 to push the rest of a or b to the result.

- Assume the begin() function for class CircularListWithHead return an iterator with its current points to the node head→link.

```
Polynomial Polynomial::operaor+(const Polynomial& b) const
{ // *this (a) and b are added and the sum returned
    Term temp;
    CircularListWithHead<Term>::Iterator ai = poly.begin(),
                                         bi = b.poly.begin();
    Polynomial c; //assume constructor sets head→exp = -1
    while (1) {
        if (ai→exp == bi→exp) {
           if (ai→exp == -1) return c;
           int sum = ai→coef + bi→coef;
           if (sum) c.poly.InsertBack(temp.Set(sum, ai→exp));
           ai++; bi++; // to next term
        }
        else if (ai→exp < bi→exp) {
            c.poly.InsertBack(temp.Set(bi→coef, bi→exp));
            bi++;  // next term of b
        }
        else {
            c.poly.InsertBack(temp.Set(ai→coef, ai→exp));
            ai++;  // next term of a
        }
    }
}
```

Experiment: P209-5

4.8 Equivalence Classes

Definition: **A relation ≡ over a set S is an equivalence relation over S iff it is symmetric, reflexive, and transitive over S.**

≡ partitions the set S into equivalence classes. For example, let S={0,1,..,10,11} **and**

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

S is partitioned into 3 equivalence classes:
{0,2,4,7,11}; {1,3,5}; {6,8,9,10}

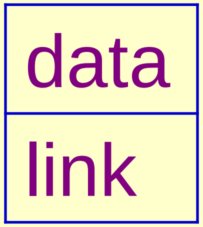
To determine equivalence classes, begin at 0 and find all pairs of (0, j). 0 and j are in the same class. By transitivity, all (j, k) imply k is in the same class. Continue this way until the entire class containing 0 has been found, marked. Then go on for another class…

Major difficulty: find the implied k. Use a strategy similar in the maze problem.

Representation:
m---the number of related pairs.
n---the number of objects.

We could use a bool array pairs[n][n], such that pairs[i][j]=TRUE iff i ≡j .

- **To save space use a linked list to represent each row.**

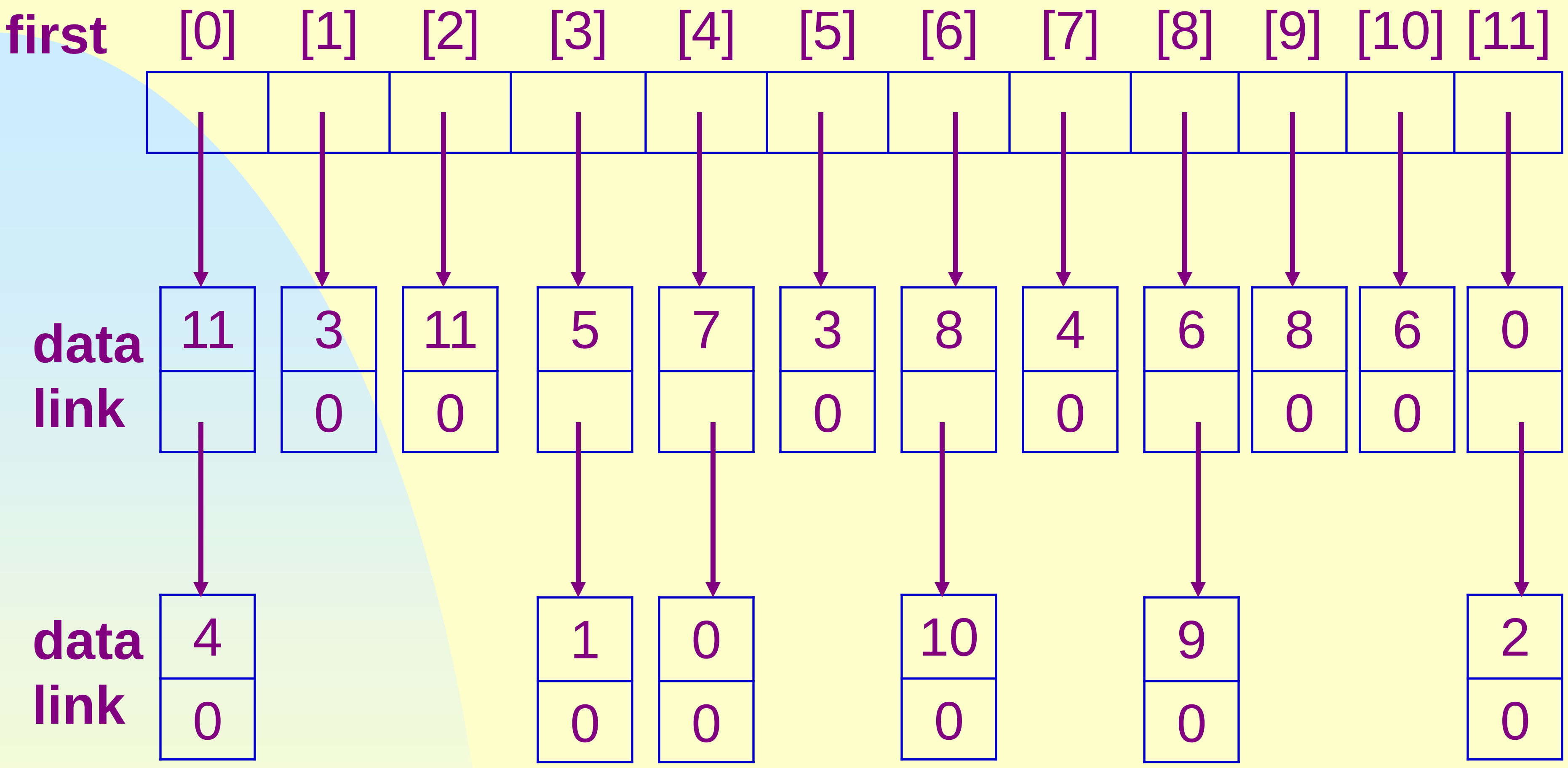


data

link

- **To access row i at random, first[n].**
- **A bool array out[n] to mark objects visited.**

Given the above data, the lists will be as in the next slide.



first	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
data	11	3	11	5	7	3	8	4	6	8	6	0
link		0	0			0		0		0	0	
data	4			1	0		10		9			2
link	0			0	0		0		0			0

- For each (i, j), 2 nodes used.
- To determine an equivalence class containing i, 0≤i<n, and out[i]==false, each element in the list first[i] is printed.
- To process the remaining lists which , by transitivity, belong in the same class as i, a stack of their nodes is created, using inverse list skill.

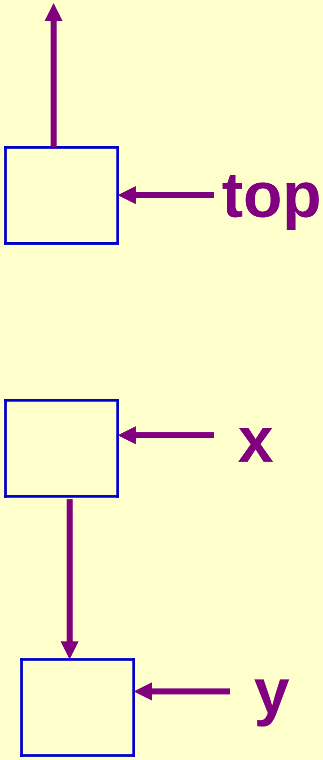
Now the code for the algorithm.

```
class ENode {
friend void equivalence();
Public:
    Enode(int d, Enode* l = 0) //constructor
        {data=d; link=l;}
private:
    int  data;
    ENode *link;
};
```

```
void Equivalence()
{
 // Input the equivalence pairs and output the equivalence
 // classes
  ifstream inFile(“equiv.in”,iso::in); // “equiv.in” is the input file
  if (!inFile) throw ”Cannot open input file.”;
  int i, j, n;
  inFile>>n; // read number of objects
```

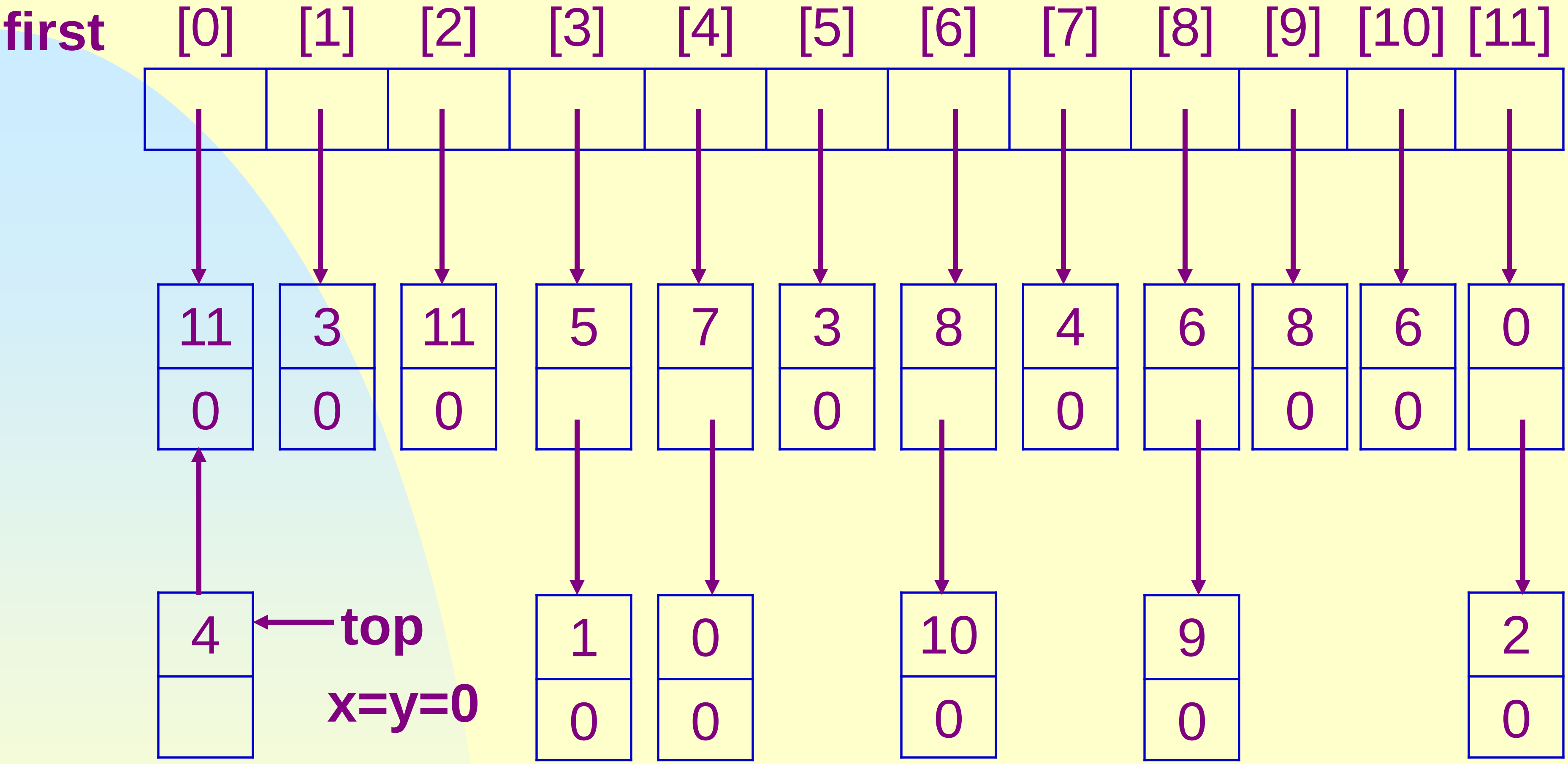
```
// initialize first and out
Enode ** first = new Enode*[n];
bool *out = new bool[n];
fill(first, first + n, 0);
fill(out, out + n, false);

// Phase 1:input equivalence pairs
inFile>>i>>j;
while (inFile.good()) { // check end of file
    first[i] = new ENode(j, first[i]);
    first[j] = new Enode(i, first[j]);
    inFile >> i >> j;
}
```

```
// Phase 2: output equivalence classes
for (i=0; i<n; i++)
 if (!out[i]) { // need to be output
   cout<<endl<<“A new class: “<<i;
   out[i]=true;
   ENode *x=first[i]; ENode *top=0; //initialize stack
   while (1) { // find rest of class
while (x)  {  // process the list
    j=x->data;
   if (!out[j]) {
      cout<<",”<<j;
      out[j]=true;
      ENode *y=x->link;
      x->link=top;
      top=x;
      x=y;
   }
   else {ENode *y=x->link; delete x; x=y;}
 }   // end of while (x)
```



```
            if (!top) break;
            x=first[top->data];
            ENode *y=top; top=top->link; delete y; //unstack
        }  // end of while (1)
     }  // end of if (!out[i])
     delete [ ] first; delete [ ] out;
} // end of equivalence
```

Running process for the previous data:



A new class: 0, 11, 4 at this moment

Follow the algorithm using the above data, we get:

A new class: 0, 11, 4, 7 (skip over 0, 4, 0), 2 (skip over 11)

A new class: 1, 3, 5 (skip over 1, 3)

A new class: 6, 8, 10 (skip over 6, 6), 9 (skip over 8)

Analysis of Equivalence:

- initialization of first and out ---O(n)

- Phase 1---O(m+n)

- in Phase 2, each node is put onto the stack once, there are only 2m nodes. The for loop is executed n times. ---O(m+n)

Overall time: O(m+n)

Space required is also O(m+n)

Exercises: P215-1

4.9 Sparse Matrices

For various number of nonzero terms in sparse matrices, sequential representation is inefficient duo to the data movement involved. To overcome it, linked representation is a better choice.

- **each row , and each column also, is represented as a circularly linked list with head node.**

- **each node has a field head for distinguishing between head node and typical node:**

<table>
<tr><td>down</td><td>head</td><td>right</td></tr>
<tr><td colspan="3">next</td></tr>
</table>

(a) head node, head=true

<table>
<tr><td>down</td><td>head</td><td>row</td><td>col</td><td>right</td></tr>
<tr><td colspan="5">value</td></tr>
</table>

(b) typical node, head=false

- each head node is in 3 lists: row (linked by **right**), column (linked by **down**) and head (linked by **next**). The head node for row i is also for column i.

- each typical node is in 2 lists: row (linked by **right**) and column (linked by **down**).

- the head node for the head nodes list is a typical node with its **row** and **col** storing dimensions, **right** linked into the head nodes list, **value** storing the number of nonzero terms.

For an n × m matrix with r nonzero terms, the total nodes needed:

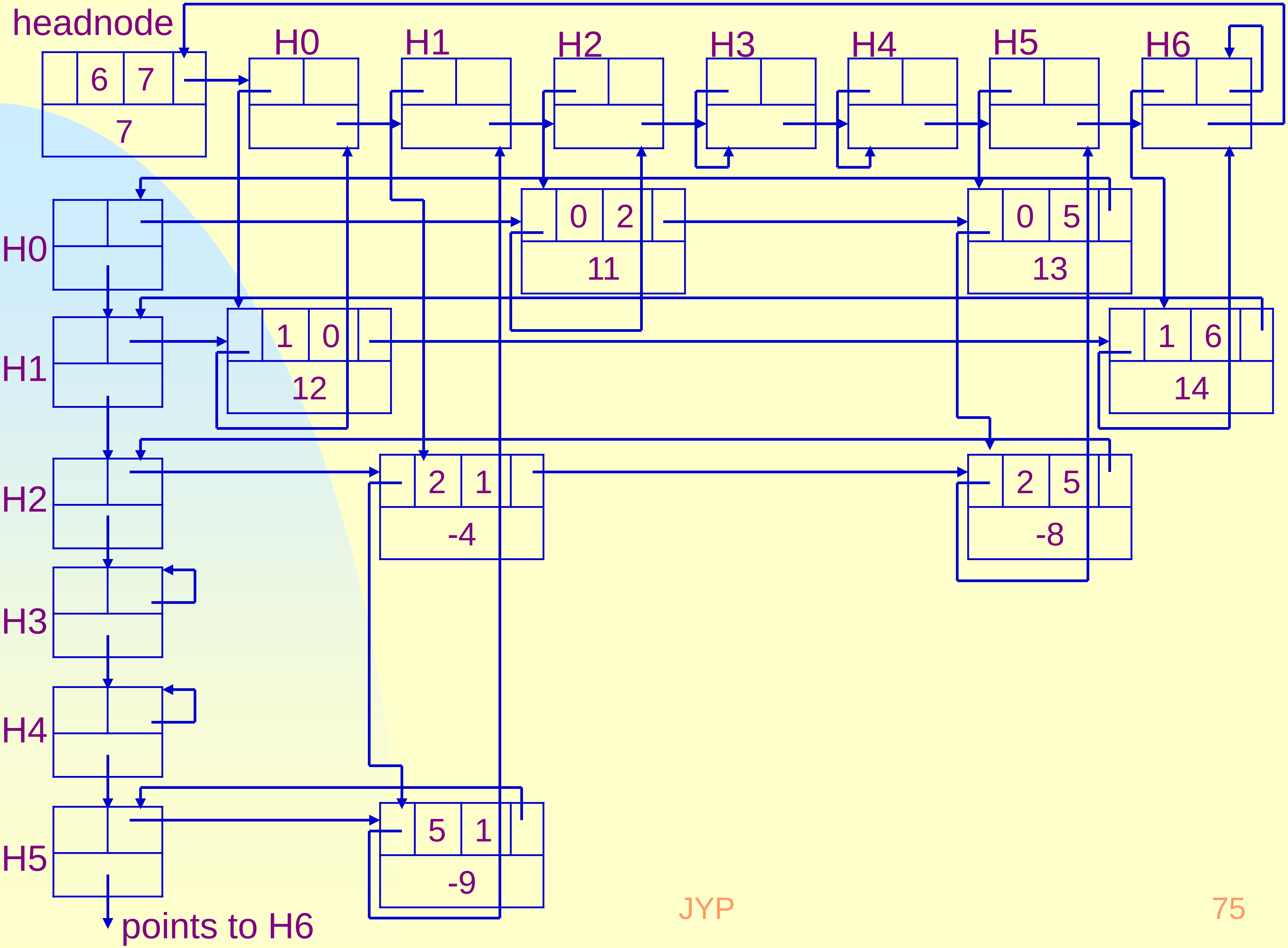
max{m,n} + r +1 (for the head of head)

```
struct Triple { int row, col,value;};
class Matrix;
class MatrixNode {
friend class Matrix;
friend istream& operator>>( istream&, Matrix&);
private:
    MatrixNode *down, *right;
    bool head;
    union {MatrixNode *next; Triple triple;  };
    MatrixNode ( bool, Triple* );   //constructor
};
```

```
MatrixNode::MatrixNode (bool b, Triple * t ) {
    head = b;
    if ( b ) {right=next=this;} // row/column head node,
                                // down will be set in program
    else triple=*t;
}

class Matrix {
friend istream& operator>>(istream&,Matrix& );
public:
    ~Matrix ( );
private:
    MatrixNode *headnode;   //points to the head of head nodes
};
```

$$\begin{vmatrix} 0 & 0 & 11 & 0 & 0 & 13 & 0 \\ 12 & 0 & 0 & 0 & 0 & 0 & 14 \\ 0 & -4 & 0 & 0 & 0 & -8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -9 & 0 & 0 & 0 & 0 & 0 \end{vmatrix}$$

The above 6×7 matrix is represented as in the next slide.


headnode
6
7
7
H0
H1
H2
H3
H4
H5
H6
H0
H1
H2
H3
H4
H5
0
2
11
0
5
13
1
0
12
1
6
14
2
1
-4
2
5
-8
5
1
-9
points to H6

4.8.2 Sparse Matrix Input

Input:

n, m, r; i, j, a_{ij}; …; and (i, j, a_{ij}) ordered by (i, j).

Auxiliary array head of size max {n,m}, head[i] points to the head for column i and also row i---permitting efficient random access.

The next of head node i initially keep track of the last node in column i.

Method:

- set up all head nodes;
- set up each row list, simultaneously build the column lists;
- finally close the last row list, close all column lists, set up a head node for the head nodes list and link all head nodes together.

```
istream& operator>>(istream& is, Matrix& matrix )
// read in a matrix and set up its linked representation
{
    Triple s;
    is >> s.row >> s.col >> s.value; //matrix dimensions
    int p=max(s.row, s.col);

    // set up head node for the head nodes list
    matrix.headnode=new MatrixNode ( false, &s );
    if ( p==0 ) {
        matrix.headnode→right = matrix.headnode;
        return is;
    }   // there is always the head of head nodes list

   // at least one row or column
   MatrixNode **head = new MatrixNode* [p];
```

```
for (int i = 0; i < p; i++) // initialize head nodes
     head[i] = new MatrixNode(true, 0 );
int CurrentRow=0;
MatrixNode *last=head[0]; // last node in current row

for (i = 0; i < s.value; i++ ) { //input triples
    Triple t;
    is >> t.row >> t.col >> t.value;
    if ( t.row > CurrentRow ) { // close current row
         last→right = head[CurrentRow];
         CurrentRow = t.row;
         last = head[CurrentRow];
     }
     last=last→right=new MatrixNode (false, &t); // into row
     head[t.col]→next=head[t.col]→next→down= last;//column
 }
```

```
    last→right=head[CurrentRow]; // close last row
    //close all column lists
    for (i=0; i<s.col; i++) head[i]→next→down=head[i];
    // link the head nodes together
    for (i=0; i< p-1; i++) head[i]→next =head[i+1];
    head[p-1]→next = matrix.headnode; // p≠0
    matrix.headnode→right = head[0];
    delete [ ] head;
    return is;
}
```

Computing time:

- **all head nodes---O(max(n,m))**
- **all triples---O(r)**

Total time: O(n+m+r)

4.8.3 Erasing a Sparse Matrix

Assume av points to the front of the available space list linked through the field right.

Erase a matrix by rows, each row list is circularly linked by the field right, can be done in O(1). Altogether in O(max {n, m})=O(n+m)

```
Matrix::~Matrix ( )
// return all nodes to the av list linked through right,
// av is a static variable of type MatrixNode*
{
   if (! headnode) return; // no nodes to dispose
   MatrixNode *x = headnode→right, *y;
   headnode→right=av; av=headnode;//return the head of head
   while ( x!=headnode ) {
        y = x→right;  x→right = av;  av = y;
        x = x→next;
   }
   headnode = 0;
}
```

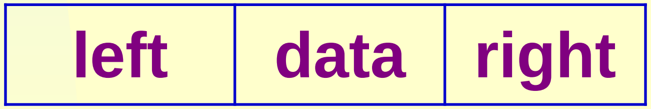
Exercises: **P222-1,3**

4.10 Doubly Linked Lists

Difficulties with singly linked list:

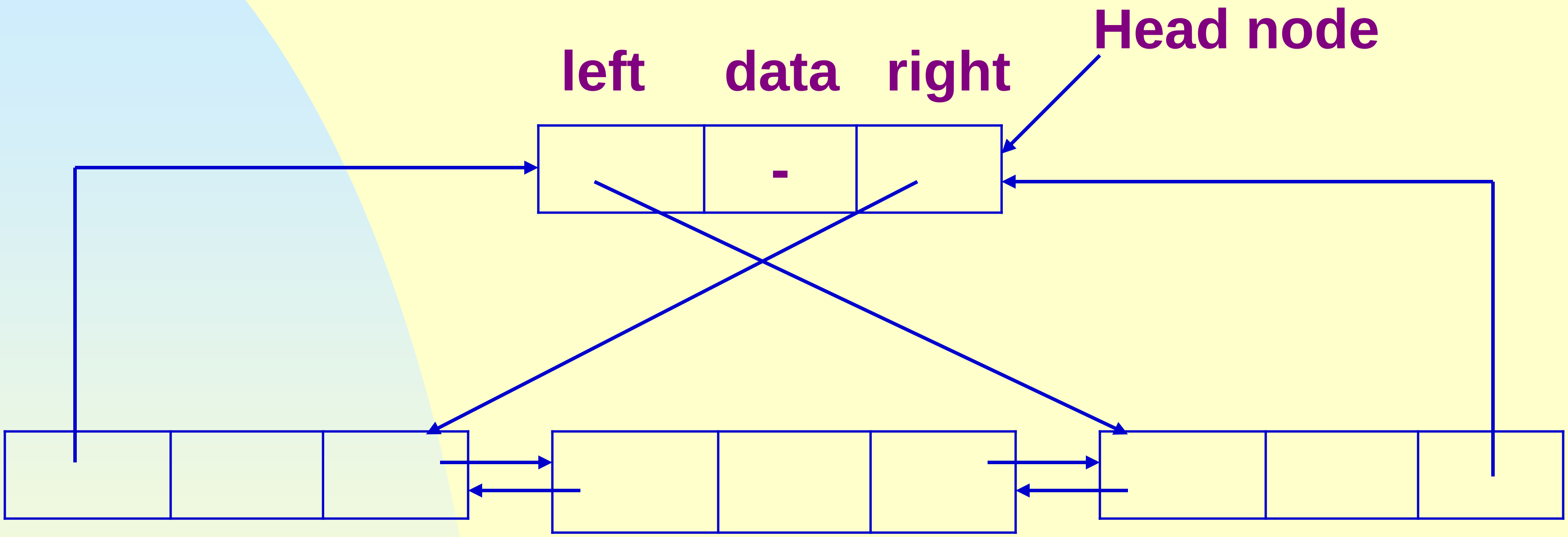
- **can easily move only in one direction**
- **not easy to delete an arbitrary node---requires knowing the preceding node**

A node in doubly linked list has at least 3 fields: data, left and right, this makes moving in both directions easy.



left	data	right

A doubly linked list may be circular. The following is a doubly linked circular list with head node:



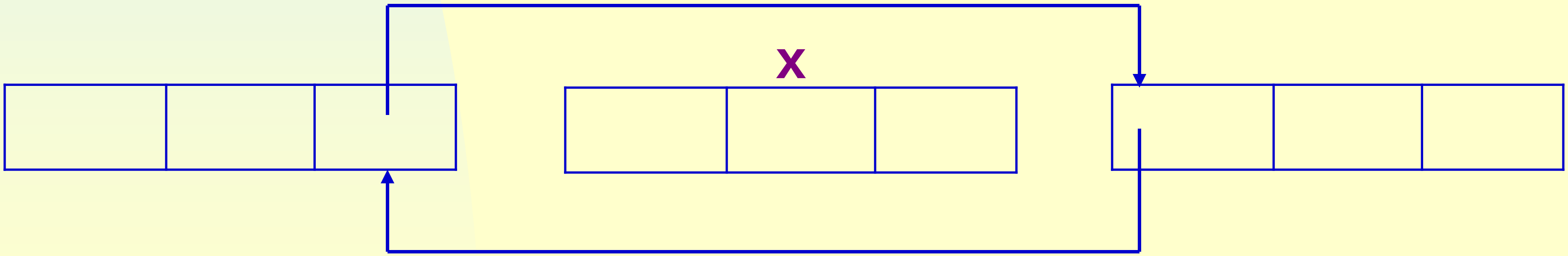
Suppose p points to any node, then

p == p→left→right == p→right→left

```
class DblList;

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

class DblList {
public:
    // List manipulation operations
    …
private:
    DblListNode *first; // points to head node
};
```

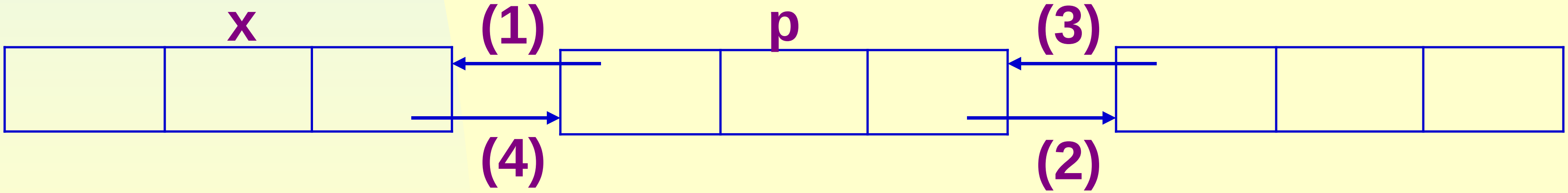
Delete

```
void DblList::Delete(DblListNode *x )
{
    if(x == first) throw "Deletion of head node not permitted";
    else  {
        x→left→right = x→right;
        x→right→left = x→left;
        delete x;
    }
}
```



Insert

```
void DblList::Insert(DblListNode *p, DblListNode *x )
{ // insert node p to the right of node x

      p→left = x;              // (1)
      p→right = x→right;  // (2)
      x→right→left = p;     // (3)
      x→right = p;            // (4)
}
```



Exercises: P225-2

4.11 Generalized Lists

4.11.1 Representation of Generalized Lists

Consider a linear list of $n \geq 0$ elements.

$$A = (\alpha_0, \dots , \alpha_{n-1})$$

where α_i $(0 \leq i \leq n-1)$ can only be an atom, the only structure property is α_i precedes α_{i+1} $(0 \leq i < n-1)$.

Definition: A generalized list, A, is a finite sequence of $n \geq 0$ elements $\alpha_0, \dots , \alpha_{n-1}$, where α_i be either an atom or a list. The elements α_i that are not atoms are said to be the sublists of A.

Note the definition is recursive.

The list A is written as A = (α_0, … , α_{n-1})

A --- name

n --- length

α_0 --- head

(α_1, … , α_{n-1}) --- tail

By convention

list name --- capital letters

atoms --- lowercase letters

Examples:

(1)A = ( )

(2)B = (a, (b, c))

(3)C = (B, B, ( ))

(4)D = (a, D) = (a, (a, (a, …)

2 consequences:

(1)lists may be shared as in (3)

(2)lists may be recursive as in (4)

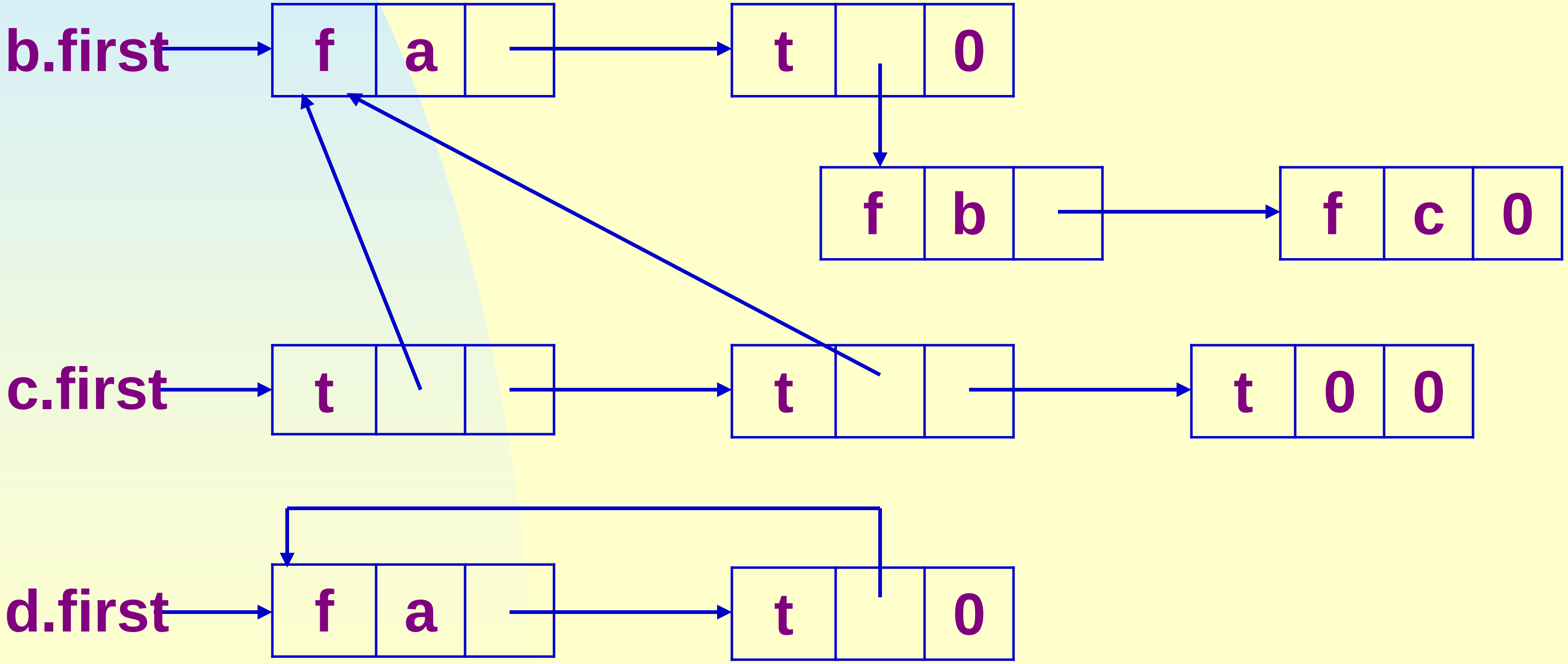
In general, every generalized list can be represented using the node structure:

tag=false/true	data/down	next

```
template <class T> class GenList;
template <class T>
class GenListNode {
friend class GenList;
private:
    GenListNode <T> *next;
    bool tag;
    union { T data;  GenListNode<T> *down;};
};

template <class T>
class GenList {
public:
    // List manipulation operations;
private:
    GenListNode<T> *first;
};
```

For any list A, the next points to the tail of A, the data/down can hold an atom (when tag==false) or a pointer to the list of head(A).

a.first=0 empty list



4.11.2 Recursive Algorithms for Lists

A recursive algorithm consists of two components:

- **workhorse---the recursive function itself.**
- **driver---the function that invokes the recursive function at the top level.**

4.11.2.1 Copying a List

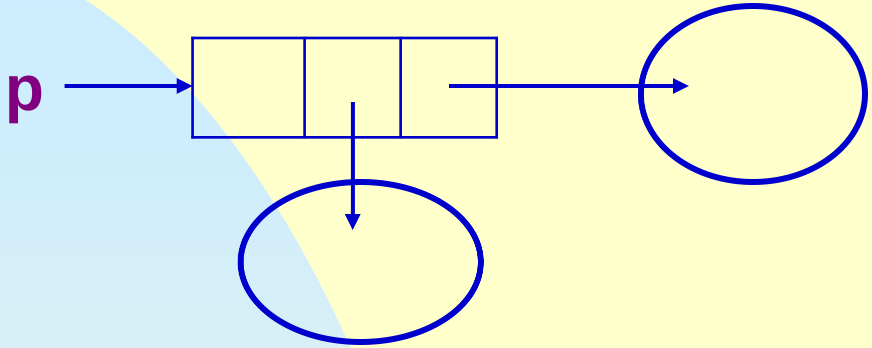
Produces an exact copy of a nonrecursive list l in which no sublists are shared.

Copy

```
template <class T>
void  GenList<T>::Copy(const GenList<T>& l) // Driver
{  first = Copy(l.first);  }
```

```
template <class T>
GenListNode<T> *GenList<T>::Copy(GenListNode<T>* p)
// Workhorse
// copy the non-recursive list with no shared sublists
{
   GenListNode<T> *q=0;
   if (p) {
         q =new GenListNode<T>;
         q→tag = p→tag;
         if   (p→tag) q→down = Copy(p→down);
         else q→data = p→data;
         q→next = Copy(p→next);
    }
    return q;
}
```

The correctness can be verified by induction.

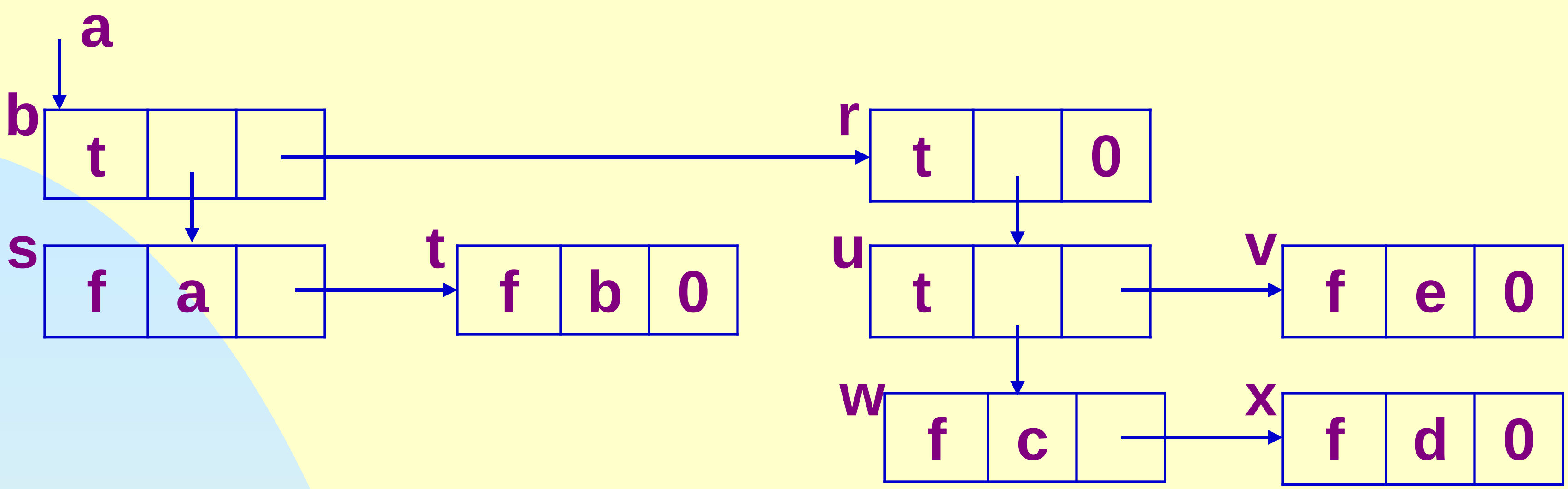


Let us run Copy for A = ((a, b), ((c, d), e)) as in the next slide.

Computing time:

- nodes with tag==false, visited twice.
- nodes with tag==true visited three times.

If the list has m nodes, the time is O(m).



Level	p	level	p	level	p	level	p
1	b	1	b	5	x	3	u
2	s	2	r	4	w	2	r
3	t	3	u	3	u	3	0
4	0	4	w	4	v	2	r
3	t	5	x	5	0	1	b
2	s	6	0	4	v		

4.10.2.2 List Equality

Two lists are identical if they have the same structure and the same data in corresponding data members.

Equal

```
// Driver
template <class T>
bool GenList<T>::operator==(const GenList<T>& l) const
{ // *this and l are non-recursive lists
  // returns true if the two lists are identical.
  return Equal(first, l.first);
}
```

```
// Workhorse– assume to be a friend of GenListNode
template <class T>
bool Equal(GenListNode<T>* s, GenListNode<T>* t)
{
  if ((!s) && (!t)) return true;
  if ( s && t && (s→tag == t→tag) )
     if (s→tag)
        return Equal(s→down, t→down)
                  && Equal(s→next, t→next);
     else return (s→data == t→data)
                  && Equal(s→next, t→next);
  return false;
}
```

Computing time:
No more than linear when no sublists are shared.
Note if a sublist is shared, it may be visited many times.

Note the algorithm terminates as soon as it discover two lists are not identical.

4.10.2.3 List Depth

$$\text{Depth(s)}=\begin{cases} 0 & \text{if s is an atom or empty list} \\ 1+\max\{\text{depth}(x_0),\ldots,\ \text{depth}(x_{n-1})\} & \text{if s is the list } (x_0,\ \ldots\ ,\ x_{n-1})\ n\geq 0 \end{cases}$$

Depth

```
// Driver
template <class T>
int GenList::Depth()
{ // compute the depth of a non-recursive list
   return Depth(first);
}
```

```
// Workhorse
template <class T>
int GenList::Depth(GenListNode<T>* s)
{
    if (!s) return 0;
    GenListNode<T>* current=s; int m=0;
    while (current) {
        if (current→tag) m=max(m,Depth(current→down));
        current=current→next;
     }
    return m+1;
}
```

4.10.3 Reference Counts,Shared and Recursive Lists

Sharing of sublists:

- **saving storage.**
- **consistency.**

To facilitate specification, extend the definition of a list to allow for naming of sublist. E.g.:
A=(a, (b,c)), Z=(b,c) ➔ A=(a, Z(b, c)) and to be consistent, A is written as A(a, Z(b, c)).

Problems with lists sharing:

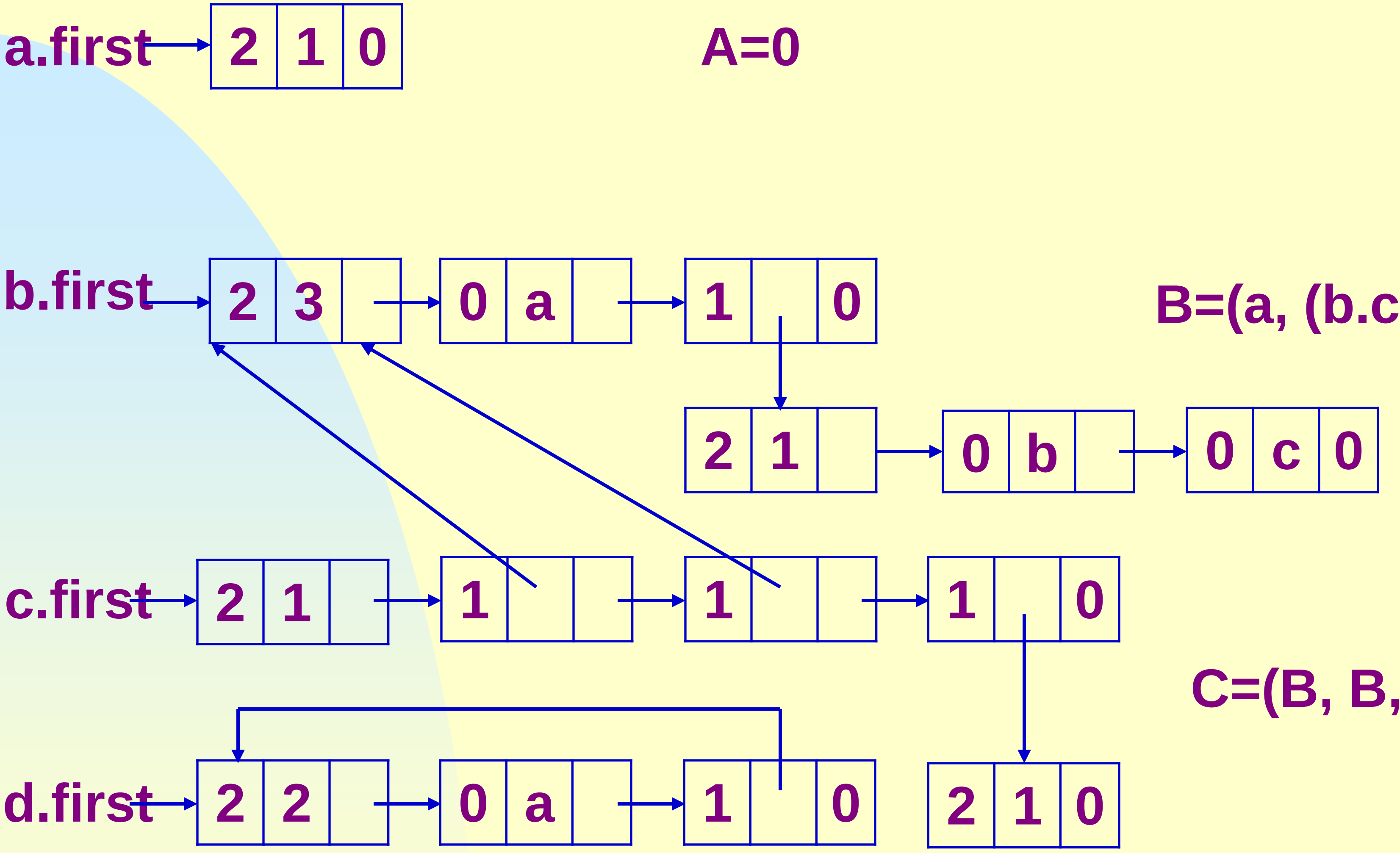
(1) Refer to the Fig. on the slide 93, if the first node of B is deleted, the pointers from C should be updated, but normally we do not know all the pointers from which a particular list is being referenced. --- solution: head node.

(2) How to determine whether or not the lists nodes may be actually freed? --- solution: reference counts, using the head node. The reference count of a list is the number of pointers to it.

We need some change in the class GenListNode:

```
template <class T>
class GenListNode<T> {
friend class GenList<T>;
private:
    GenListNode<T> *next;
    int tag;  // 0 for data, 1 for down, 2 for ref
    union {
        T data;
        GenListNode<T> *down;
        int ref;
    };
};
```

With head node, lists A= ( ), B = (a, (b, c)), C = (B, B, ( )) and D = (a, D) will be represented as in the next slide.



A=0

B=(a, (b.c))

C=(B, B, ( ))

D=(a, D)

Now the ideas of erasing list:
Decrement the reference count by 1, if it becomes 0, then examining the top level nodes, any sublists encountered are erased and finally, the top level nodes are linked into the av list.

~GenList

```
// Driver
template <class T>
GenList::~GenList()
{ // Each head node has a reference count.
    if (first) { Delete(first); first = 0;}
}
```

```
// Workhorse
template <class T>
void GenList<T>::Delete(GenListNode<T>* x)
{
   x→ref--; // decrement reference count of the head node
   if ( !x→ref )
   {
         GenListNode<T> *y=x; // y traverses top level of x.
         while (y→next)  {
             y=y→next;  if (y→tag==1) Delete(y→down);
         }
         y→next = av;
         av = x;
    }
}
```

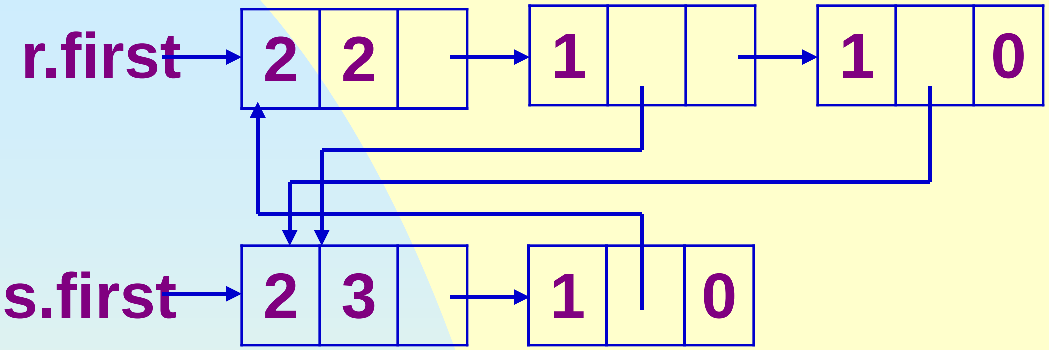
Refer to slide 106, a call to b.~GenList( ) makes:

- ref(b)→2. After that

A call to c.~GenList( ) results in:

- ref(c)→0, ref(b)→1→0.
- all the six nodes of B(a, (b, c)) returned to av list.
- the empty list node of C(B, B, ()) returned to av.
- the top level nodes of c returned to av.

For recursive lists, reference count never becomes 0, even if it is no longer accessible.



r=A(B,B)

S=B(A)

After calls to r.~GenList( ) and s.~GenList( ), ref(r) =1, ref(s)=2, but the structure is no longer being used. --- there is no simple way to solve it.

Exercises: P240-6