國立清華大學 電機工程學系

EE2410 Data Structure

Chapter 4

Linked List (Part II)

Outline

- **Equivalence Class**
- **Sparse Matrices**
- **Doubly Linked Lists**
- **Generalized Lists**
- **Virtual Functions and Dynamic Binding**

Equivalence Relation

- **A relation is a set of pair (*x*, *y*)**
  - **where *x* and *y* are elements in a set, say S**
- **Three properties of an equivalence relation**
  - **Reflexive: $x \equiv x$**
  - **Symmetric: If $x \equiv y$, then $y \equiv x$**
  - **Transitive: If $x \equiv y$ and $y \equiv z$ then $x \equiv z$**
- **Definition**
  - **A relation over a set S, is said to be an equivalence relation over S iff it is symmetric, reflexive, and transitive over S.**
  - **E.g., "equal to" (=) relationship is an equivalence relation**

Finding Equivalence Classes

- **Input**

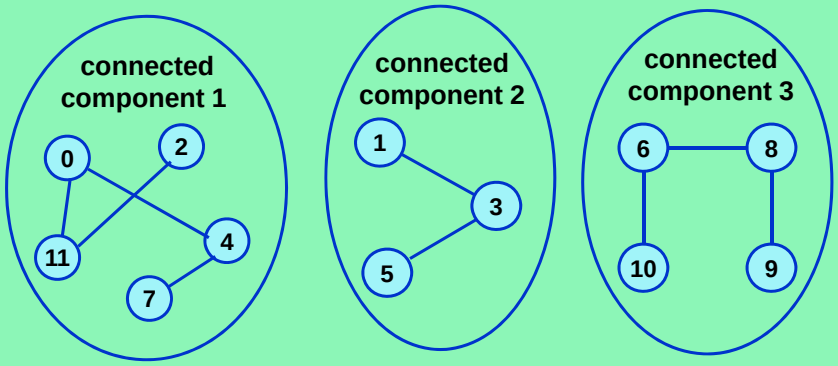
  0≡4, 3≡1, 6≡10, 8≡9, 7≡4, 6≡8, 3≡5, 2≡11, 11≡0
- **Output**
  - **Three equivalence classes by applying the three properties:**

    { 0,2,4,7,11}; { 1,3,5 }; { 6,8,9,10 }
- **Basic Algorithm**
  - **(phase 1): equivalence pairs (i, j) are read in and stored**
  - **(phase 2): find each equivalence class by transitive rule**

Relation Graph

- **Input**

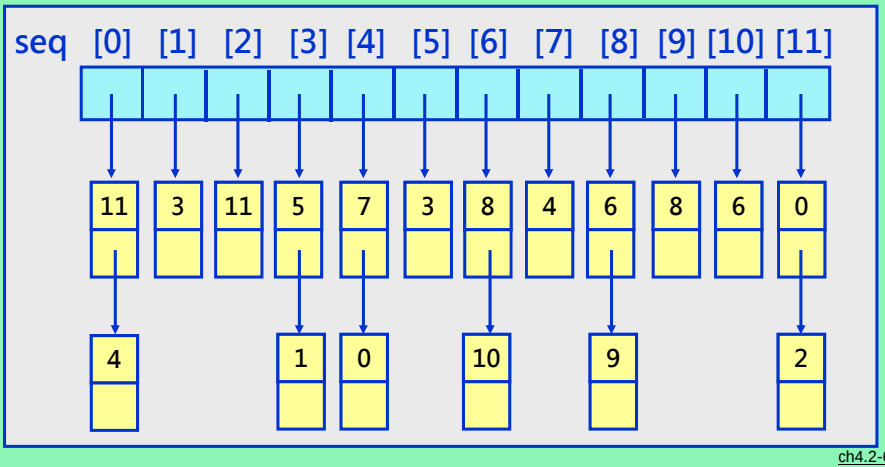
0≡4, 3≡1, 6≡10, 8≡9, 7≡4, 6≡8, 3≡5, 2≡11, 11≡0



A Graph consists of vertices and edges

Data Structure For Storing the Equivalence Pairs

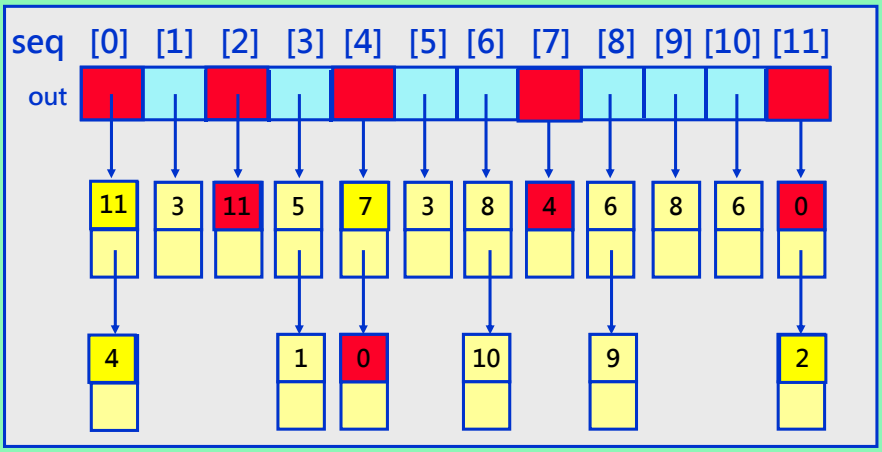
Equivalence pairs:
0≡4, 3≡1, 6≡10, 8≡9, 7≡4, 6≡8, 3≡5, 2≡11, 11≡0



Finding Equivalence Class Containing Node 0

Processing order: [0] → [11] → [4] → [2] → [7]

equivalence class containing 0 : { 11, 4, 2, 7}



Overall Equivalence Class Computation

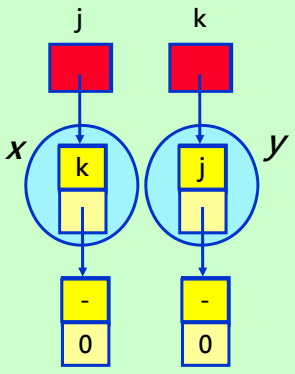
```
void equivalence()
{
   read  n;  // read in number of objects
  initialize seq to 0 and out to FASLE;
  while (more pairs) // input pairs
  {
        read the next pair (i, j);
        put j on the seq[i] list;
        put i on the seq[j] list;
   }
   /*------- print out equivalence classes -----*/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

Data Structure in C++

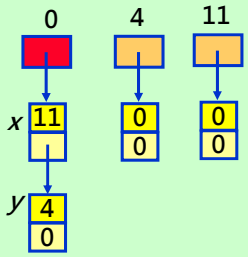
```
enum  Boolean { FALSE, TRUE };

class ListNode {
friend void equivalence ();
private:
    int          data;
    ListNode  *link;
    ListNode(int);  // private constructor
};

typedef  ListNode *ListNodePtr;
// so we can create an array of pointers using new

ListNode::ListNode(int d)  // constructor
{
    data = d;
    link = 0;
}
```



```
void  equivalence ()
// Input the equivalence pairs and output the equivalence classes
{
  ifstream  inFile("equiv.in", ios::in); // "equiv.in" is the input file
  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 (k=0; k<n; k++){
       seq[k] = 0;
       out[k] = FALSE;
  }
  // Phase 1: input equivalence classes
  inFile >> j >> k ;
  while ( inFile.good() ) {  // check end of file
        ListNode  *x = new ListNode(k);  x→link = seq[j]; seq[j] = x; // add k to seq[j];
        ListNode  *y = new ListNode(j);  y→link = seq[k]; seq[k] = y; // add j to seq[k];
        inFile >> j >> k;
   }
```



```
void  equivalence () // Phase 2: output equivalence classes
  …. (previous page)
for (k=0; k<n; k++){
        if (out[k] == FALSE) {   // needs to be output
            cout << endl << "A new class: " << k;   out[k] = TRUE;
            ListNode *x = seq[k];  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;  // skip current node x
               }
               if ( ! top )  break;
               else { x = seq[top→data];   top = top→link;  // unstack }
            } // end of while(1)
        }  // end of if (out[k] == FALSE)
}
for ( k=0; k<n; k++)
    while(seq[k]) { ListNode *delnode = seq[k];  seq[k] = delnode→link; delete delnode; }
    delete [] seq;  delete [] out;
}
```



m equivalence pairs
n nodes
→ O(m+n) algorithm

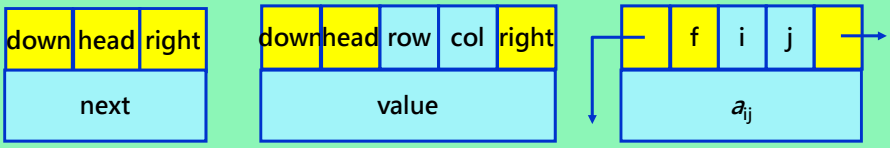
Outline

- Equivalence Class
- Sparse Matrices
- Doubly Linked Lists
- Generalized Lists
- Virtual Functions and Dynamic Binding

Sparse Matrix As A Two-Dimensional Linked List

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

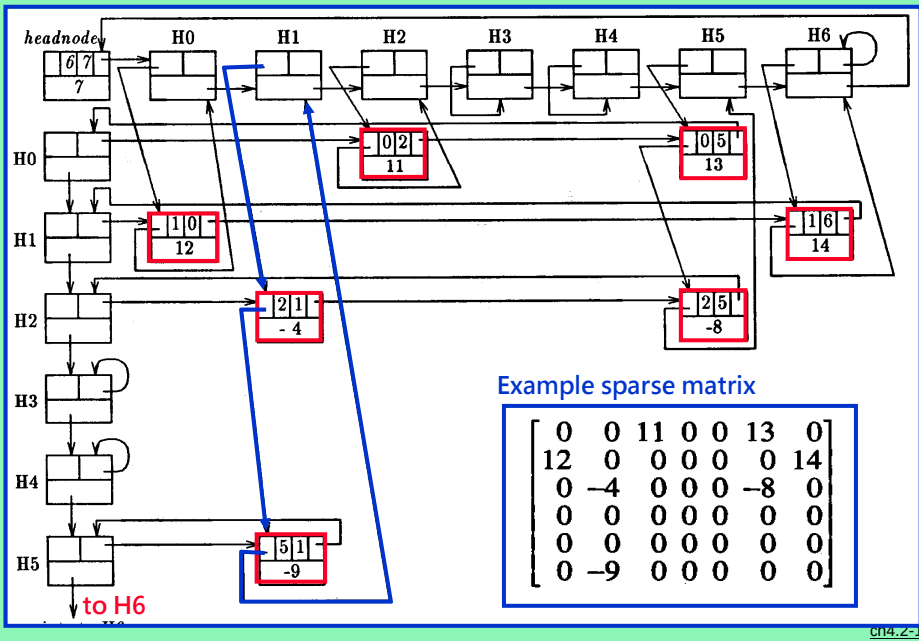
➡ 7 nonzero terms



head node　　typical node　　setup for a_{ij}

ch4.2-13

Example: Sparse Matrix



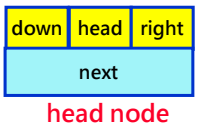
Example sparse matrix

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

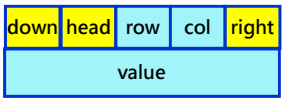
ch4.2-14

Class Definition of Sparse Matrix

```
enum Boolean { FALSE, TRUE };
struct Triple { int value, row, col ; };
class Matrix ; // forward declaration
class MatrixNode
{
friend class Matrix ;
friend istream& operator>>(istream&, Matrix&) ; // for reading in a matrix
private:
    MatrixNode *down , *right ;
    Boolean head ;
    union {   // anonymous union
            MatrixNode *next ;
            Triple triple ;
    };
    MatrixNode(Boolean, Triple *) ; // constructor
};
```



head node



typical node

```
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 ;
};
```

Reading In A Sparse Matrix (I)

```
1  istream& operator>>(istream& is, Matrix& matrix)
2  // Read in a matrix and set up its linked representation.
3  // An auxiliary array head is used.
4  {
5   Triple  s ; int p ;
6   is >> s.row >> s.col >> s.value ;   // matrix dimensions
7   if (s.row > s.col) p = s.row ; else  p = s.col ;
8   // set up headnode for list of head nodes.
9   matrix.headnode = new MatrixNode(FALSE, &s) ;
10  if (p == 0) { matrix.headnode →right = matrix.headnode ; return is ;}
11  // at least one row or column
12  MatrixNodePtr *head = new MatrixNodePtr [p] ;  // initialize head nodes
13  for (int i = 0 ; i < p ; i++)
14    head [i ]  = new MatrixNode(TRUE, 0) ;
15  int CurrentRow = 0 ; MatrixNode *last = head [0] ; // last node in current row
```

s holds matrix dimension
p = max {#rows, #cols}

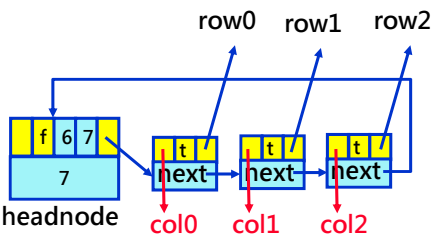
Reading In A Sparse Matrix (II)

```
16 for (i = 0 ; i < s.value ; i++)  // input triples
17 {
18   Triple t ;
19   is >> t.row >> t.col >> t.value ;
20   if (t.row > CurrentRow) {  // close current row
21     last →right = head [CurrentRow ] ;
22     CurrentRow = t.row ;
23     last = head [CurrentRow ] ;
24   } // end of if
25   last = last →right = new MatrixNode(FALSE, &t) ;  // link new node into row list
26   head [t.col]→next = head [t.col]→next →down = last ; // link into column list
27 } // end of for
28 last →right = head [CurrentRow ] ; // close last row
29 for (i = 0 ; i < s.col ; i++) head [i ]→next →down = head [i ] ; // close all column lists
30 // link the head nodes together
31 for (i = 0 ; i < p − 1 ; i++) head [i ]→next = head [i +1] ;
32 head [p −1]→next = matrix.headnode ;
33 matrix.headnode →right =  head [0] ;
34 delete [ ] head;
35 return is ;
36 }
```

Trick: head[i]→next is used initially to keep track of the last node in column i. But eventually, the head nodes are linked together through *next* (in line 30).

Erasing a Sparse Matrix

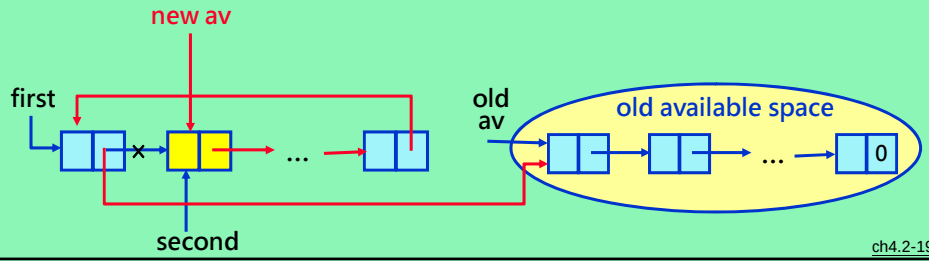
```
Matrix::~Matrix ()
// Return all nodes to the av list. This list is a chain linked via the right
// field. av is a global variable of type MatrixNode * and points to its first node.
{
  if (!headnode) return; // no nodes to dispose
  MatrixNode *x = headnode →right , *y ;
  headnode →right = av ; av = headnode ; // return headnode
  while (x != headnode) { // erase by rows
      y = x →right ;
      x →right = av ;
      av = y ;
      x = x →next ; // next row
  }
  headnode = 0 ;
}
```



Erasing A Circular List

```
template <class Type>
void CircularList<Type>::~CircularList()
// Erase the entire circular list pointed by first
{
   if ( first ) {
      ListNode*  second = first → link; // second node
      first → link = av;    // first node linked to av
      av = second;   // second node of list becomes front of av list
      first = 0;
   }
}
```

A Circular List can be erased in a fixed amount of time
→ Independent of the number of nodes in the list



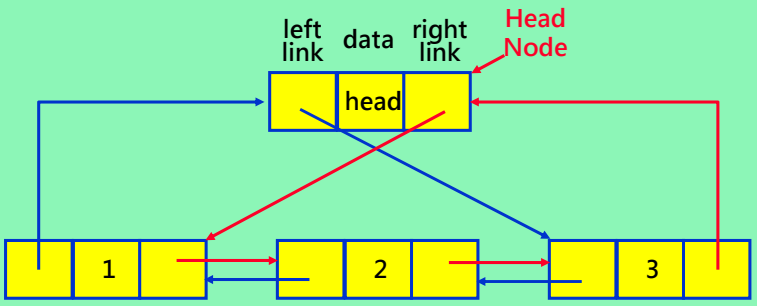
ch4.2-19

Outline

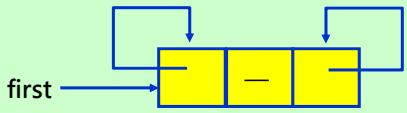
- Equivalence Class
- Sparse Matrices
- Doubly Linked Lists
- Generalized Lists
- Virtual Functions and Dynamic Binding

ch4.2-20

Doubly Linked List



Empty doubly linked circular list with head node



ch4.2-21

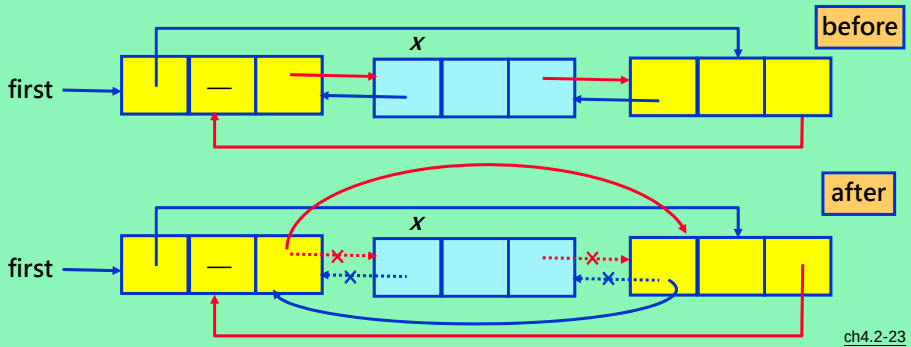
Class of a Doubly Linked List

```
class  DbList;
class  DbListNode {
friend  class DbList;
private:
    int   data;
    DbListNode  *llink, *rlink;
};

class  DbList {
public:
    // List manipulation operations
private:
    DbListNode  *first;  // points to head node
};
```

ch4.2-22

Deletion From a Doubly Linked List

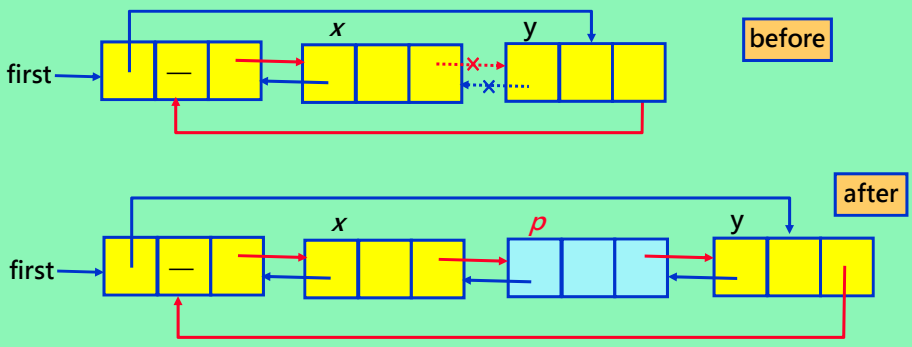
```
void   DbList::Delete ( DbListNode *x)
{
  if (x == first)  cerr << "Deletion of head node not permitted" << endl;
  else {
        x→llink→rlink = x→rlink;
        x→rlink→llink = x→llink;
        delete x;
   }
}
```



ch4.2-23

Insertion To a Doubly Linked List

```
void   DbList::Insert ( DbListNode *p, DbListNode *x)
// Insert node p to the right of node x
{
   p→llink = x;  p→rlink = x→rlink;
   x→rlink→llink = p;  x→rlink = p;
}
```



ch4.2-24

Outline

- Equivalence Class
- Sparse Matrices
- Doubly Linked Lists
- Generalized Lists
- Virtual Functions and Dynamic Binding

Generalized Lists

- **Definition**
  - A generalized list, A, is a finite sequence of $n \geqq 0$ elements, $(\alpha_0, \ldots, \alpha_{n-1})$ where α_i is either an atom or a list.
- **Head**
  - α_0 is called the head of A
- **Tail**
  - $(\alpha_1, \ldots, \alpha_{n-1})$ is called the tail of A
- **This is a recursive definition**
  - E.g., C=(*a*, C)=(*a*, (*a*, (*a*, …))), A=(*a*, (*b*, *c*)), B=(A, A, ())
  - A compact way of describing a large and varied structure

Polynomial With Multiple Variables

- **Example**

  $P(x, y, z) = x^{10}y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz$

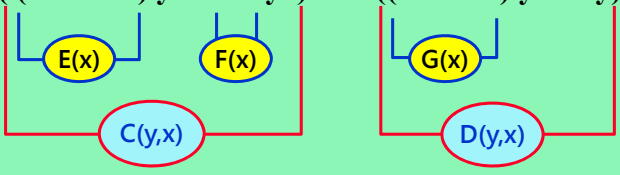
- **Sequential Representation**
  - Use a structure with four fields to represent a single array element
    - Coef, Exp_x, Exp_y, Exp_z

Coef	Exp_x	Exp_y
Exp_z	link	

A polynomial term

Factored Form of Polynomial

- **Variable order**
  - { z, y, x }
  - z is the main variable, y is the second, x is the third
- **Factored Expression**
  - $((x^{10} + 2x^8)\, y^3 + 3x^8\, y^2)\, z^2 + ((x^4 + 6x^3)\, y^4 + 2y)\, z$



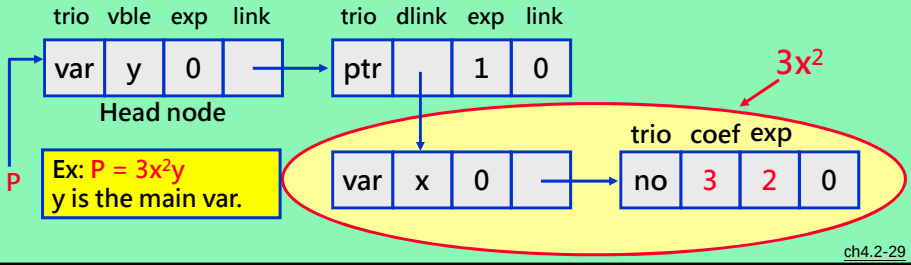
$P(x, y, z) = x^{10}y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz$

Variable Independent PolyNode

```
enum  Triple { var, ptr, no };
class PolyNode
{
   PolyNode  *link;
   int  exp;
   Triple  trio;
   union {
      char          vble;
      PolyNode  *dlink;
      int             coef;
   };
};
```

trio	exp	link
vble	dlink	coef

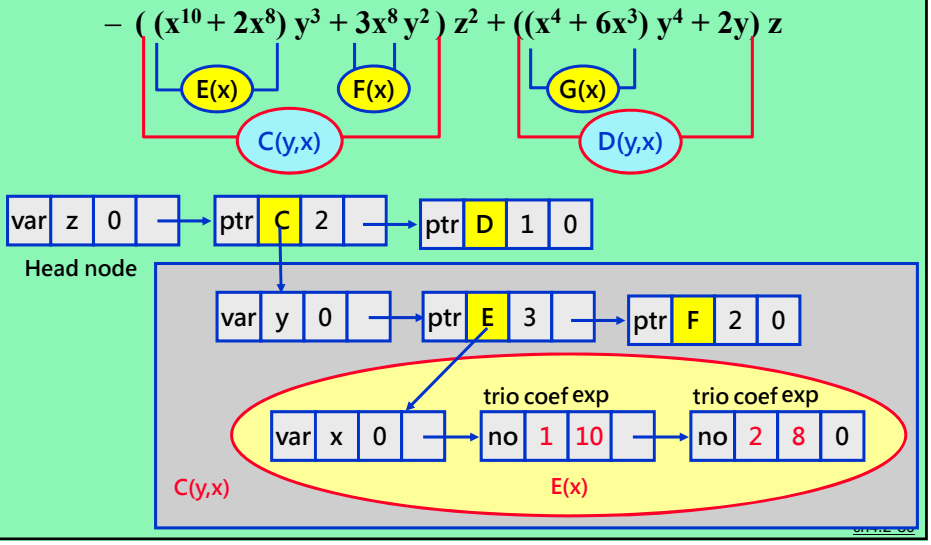
Case 1: trio==var → head node
Case 2: trio==ptr →
coef is a sub-list pointed by dlink
Case 3: trio==no
coef is an integer



Ex: Polynomial in General List

- **Factored Expression**

$$-\left((x^{10}+2x^8)\,y^3+3x^8\,y^2\right)z^2+\left((x^4+6x^3)\,y^4+2y\right)z$$



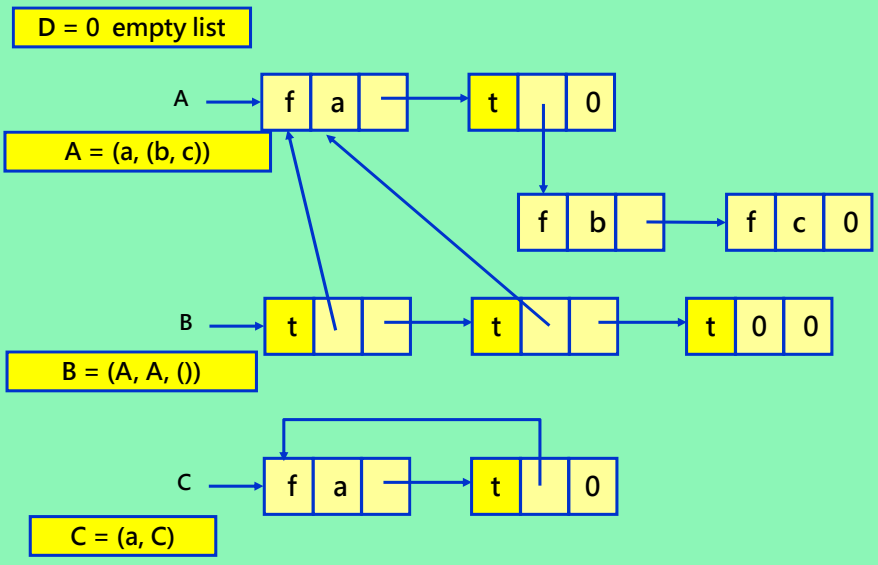
General List Class

```
enum Boolean { FALSE, TRUE};
class GenList; // forward declaration
class GenListNode {
friend class GenList;
private:
    GenListNode  *link;
     Boolean tag;   // for indication of an atom or a list
     union {
         char  data;
         GenListNode  *dlink;
     };
};
class GenList {
public:
   // List manipulation operations
private:
        GenListNode  *first;
};
```

tag = FALSE/TRUE	data / dlink	link

ch4.2-31

Example: General Lists



ch4.2-32

Recursive Algorithms

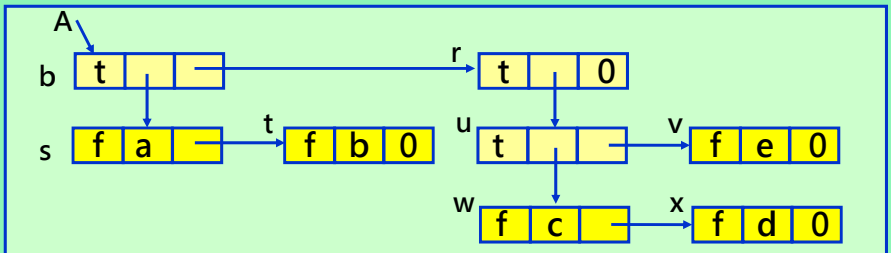
- **For recursively defined data object**
  - **It is often easy to describe algorithms that work on these objects recursively**
- **Two components in a recursive operation**
  - **(1) workhorse: the recursive function itself**
    - **Often declared as a private function**
  - **(2) driver: the function that invokes the recursive function at the top level**
    - **Declared as a public function**

Copying A General List

```
// Driver
void  GenList::Copy(const GenList& l)
{
    first = Copy(l.first);
}

// Workhorse
GenListNode *GenList::Copy(GenListNode *p)
// Copy the recursive list with no shared sublists pointed at by p
{
   GenListNode  *q = 0;
   if(p) {
      q = new GenListNode;  // q is the copied node
      q→tag = p→tag;
      if ( ! p→tag )  q→data = p→data;     // p is an atom node
      else   q→dlink = Copy ( p→dlink );  // p is a list pointer
      q→link = Copy( p→ link);
   }
   return q;
}
```

Proof: by induction
Complexity: O(m), or 3m steps
Recursion depth: m

Example: General List Copy



level of recursion	value of *p*	continuing level	*p*	continuing level	*p*
1	b	2	r	3	u
2	s	3	u	4	v
3	t	4	w	5	0
4	0	5	x	4	v
3	t	6	0	3	u
2	s	5	x	2	r
1	b	4	w	3	0
				2	r
				1	b

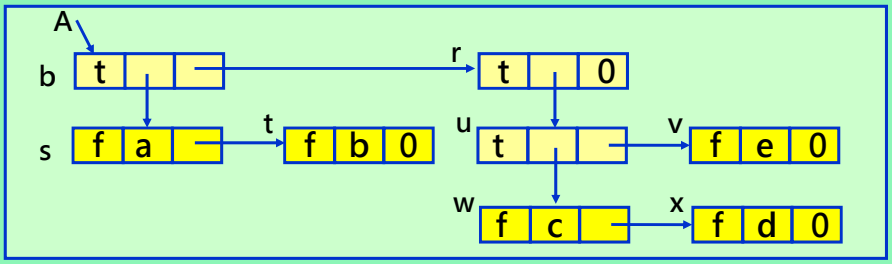
5

List Equality

```
// Driver – assumed to be a friend of GenList
int  operator==(const GenList& l, const GenList& m)
// l and m are non-recursive lists
// The function returns 1 if the two lists are identical and 0, otherwise
{
    return ( equal (l.first, m.first));
}
// Workhorse – assumed to be a friend of GenListNode
Int  equal ( GenListNode *s, GenListNode *t)
{
    int   x;
    if ( !s && !t ) return 1;     // both lists are null
    if ( s && t && (s→tag == t→tag) )
    {
        if ( ! (s→tag) )  // atom node
            if ( s→data == t→data )  x=1;  else x=0;
        else   x=equal (s→dlink, t→dlink);  // recursive call when list node
        if(x)   return ( equal ( s→link, t→link) ); else return 0;
    }
    return 0;  // only one list is null
}
```

2-36

Example: Depth of General List



A has two sub-lists: b→dlink and r→dlink
Depth of list pointed by b→dlink: 1
Depth of list pointed by r→dlink: 2
→ Depth(A) = max(Depth(b), Depth(r)) + 1 = 3

List Depth Computation

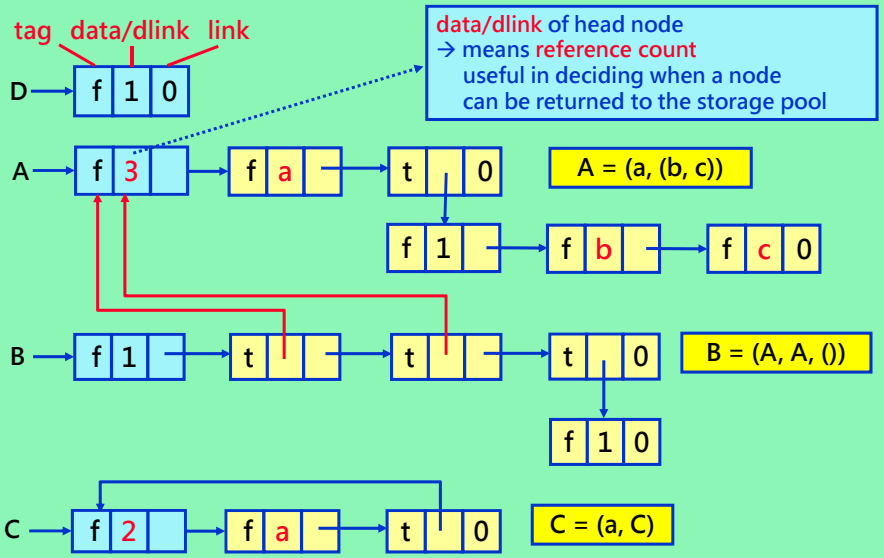
```
// Driver
int  GenList::depth()
// Compute the depth of a non-recursive list
{
    return (depth (first) );
}
// Workhorse
int  GenList::depth(GenListNode *s)
{
   if (!s) return 0;
   GenListNode *p = s; int  m=0;
   while (p) {
       if (p→tag) {   // sublist node
            int  n = depth (p→dlink);
            if (m < n)  m = n;
       }
       p = p→link;  // move forward
   }
   return  m+1;
}
```

depth(s) =

1 if s is an atom

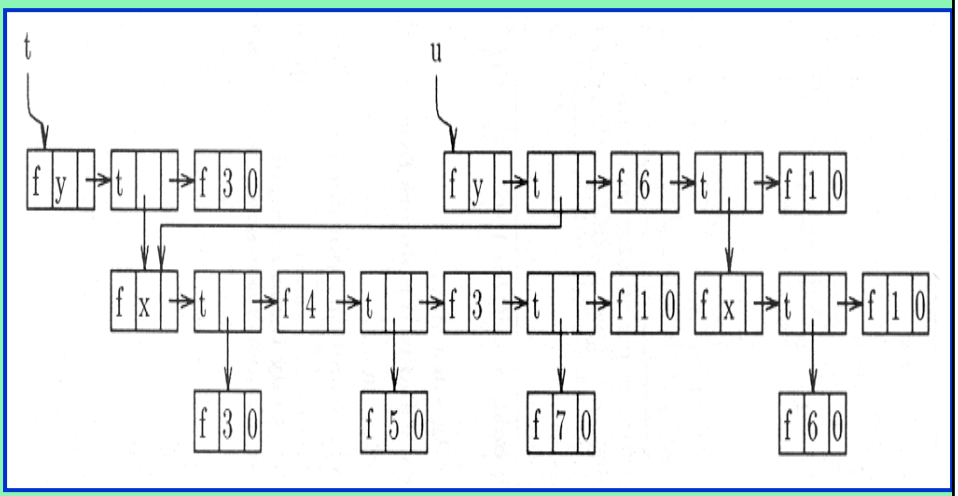
$1+\max\{ depth(x_1), \ldots, depth(x_n) \}$ if s is the list $(x_1, \ldots, x_n)$, $n \geq 1$

General Lists With Sharing



ch4.2-39

Polynomials With Sharing



$t = (3x^4 + 5x^3 + 7x)\,y^3$

$u = (3x^4 + 5x^3 + 7x)\,y^6 + (6x)y$

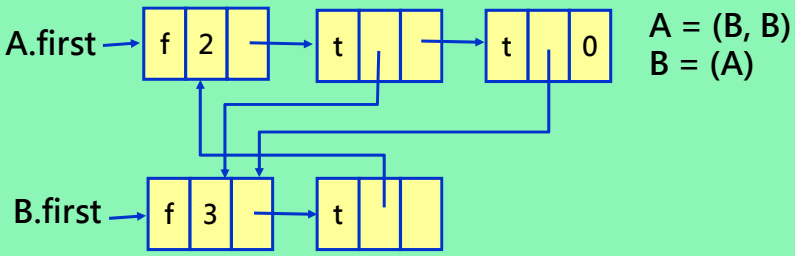
ch4.2-40

Erasing A List Recursively

```
// Driver
int  GenList::~GenList()
// Each head node has a reference count. We assume first ≠0
{
    Delete ( first );     first = 0;
}
// Workhorse
int  GenList::Delete ( GenListNode *x)
{
   x → ref--;  // decrement the reference count of head node
   if ( ! x → ref )
   {
      GenListNode  *y = x;  // y traverses top-level of x
       while ( y → link )  {
            y = y → link;
            if ( y → tag == 1 )    Delete ( y → dlink ); }
            y → link = av;   // attach top-level nodes to av list
           av = y;
       }
   }
}
```


Indirect Recursion Case

- **For recursive list such as C = (a, C)**
  - **The reference count will never be 1**
  - **So, they cannot be recycled**
- **Indirect recursive lists cannot be recycled either**



Outline

- Equivalence Class
- Sparse Matrices
- Doubly Linked Lists
- Generalized Lists
- Virtual Functions and Dynamic Binding

Dynamic Binding

- **Public Inheritance**
  - Rectangle IS-A Polygon
  - Rectangle has all attributes of Polygon
  - Pointer to a derived class is implicitly converted to a pointer to its bass class
- **For example**
  - Rectangle r; // instance of derived class
  - Polygon *s = &r; // assign rectangle to polygon
- **Member function types**
  - Virtual functions
  - Non-virtual functions
  - Pure virtual functions
    - The responsibility of the implementation is passed on to the derived class

Example: Inheritance

```
class Polygon
{
public:
    int  GetId();  // non-virtual member function
    virtual  Boolean  Concave();
    virtual  int  Perimeter() = 0;  // pure virtual function
protected:
    int   id;
};

class Rectangle : public Polygon  // Rectangle publicly inherits from Polygon
{
public:
    Boolean  Concave();  // redefined in Rectangle
    int  Perimeter();       // defined in Rectangle
    // GetId() and id are inherited from Polygon
    // They, respectively, become public and protected members of Rectangle
private:
    // additional data members required to specialize Rectangle
    int  x1, y1, h, w;
};
```


Example: Inheritance (con't)

```
// GetId() must never be redefined in a derived class
int  Polygon::GetId(){ return id; }

// Default implementation of Concave() in Polygon. A polygon is concave
// if it is possible to construct a line joining two points in the polygon
// that does not entirely lie within the polygon
Boolean  Polygon::Concave() {  return TRUE; }

// Rectangle must define Perimeter() because it is a pure virtual function
int  Rectangle::Perimeter() {  return 2*(h+w); }

// The default implementation of Concave() does not apply to rectangles
// So, it has to be redefined
Boolean Rectangle::Concave(){ return FALSE; }
```

The End of Linked Lists

Next Topic:

Trees