Linked List

Introduction to Data Structures

Kyuseok Shim
SoEECS, SNU.

4.1 Singly Linked Lists and Chains

- Sequential representation
  - successive nodes of the data object are stored a fixed distance apart
  - order of elements is the same as in ordered list
  - adequate for functions such as accessing an arbitrary node in a table
  - operations such as insertion and deletion of arbitrary elements from ordered lists become expensive

4.1 Singly Linked Lists and Chains

- Linked representation
  - successive items of a list may be placed anywhere in memory
  - order of elements need not be the same as order in list
  - each data item is associated with a pointer (link) to the next item

4.1 Singly Linked Lists and Chains

- List of 3-letter words : (BAT, CAT, EAT, …, VAT, WAT)

	data	*link*
1	HAT	15
2		
3	CAT	4
4	EAT	9
5		
6		
7	WAT	0
8	BAT	3
9	FAT	1
10		
11	VAT	7
⋮	⋮	⋮

data[8] = BAT
first = 8
link[8] = 3
data[3] = CAT

Figure 4.1 : Nonsequential list representation

4.1 Singly Linked Lists and Chains

- List of 3-letter words : (BAT, CAT, EAT, ..., VAT, WAT)

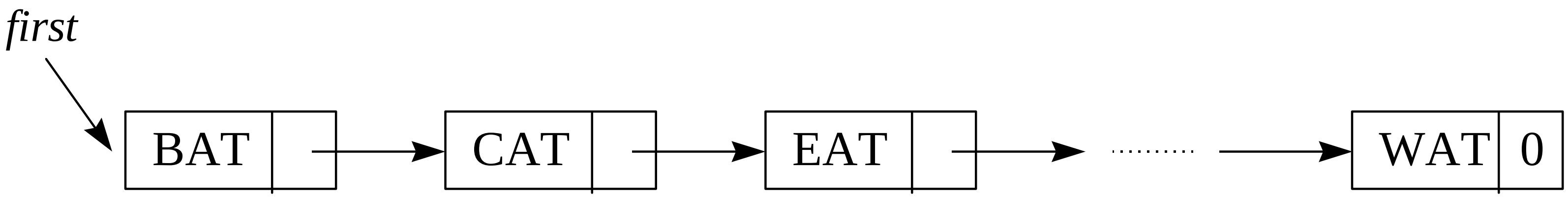


Figure 4.2 : Usual way to draw a linked list

4.1 Singly Linked Lists and Chains

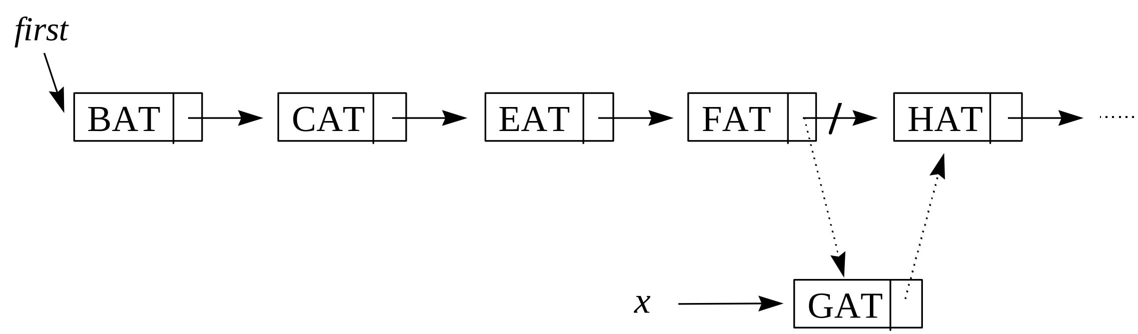
- To insert GAT between FAT and HAT
  - (1) get a node N that is currently unused ; let its address be x
  - (2) set the data field of N to GAT
  - (3) set the link field of N to point to the node after FAT, which contains HAT
  - (4) set the link field of the node containing FAT to x

4.1 Singly Linked Lists and Chains

- To insert GAT between FAT and HAT

	data	*link*
1	HAT	15
2		
3	CAT	4
4	EAT	9
5	GAT	1
6		
7	WAT	0
8	BAT	3
9	FAT	5
10		
11	VAT	7

(a) Insert GAT into data[5]



(b) Insert node GAT into list

Figure 4.3 : Inserting a node

4.1 Singly Linked Lists and Chains

- To delete GAT
  - find the element that immediately precedes GAT, which is FAT
  - set the link field of FAT to the position of HAT

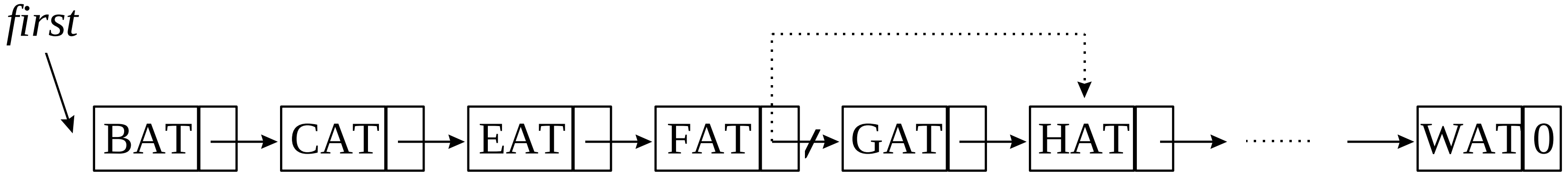


Figure 4.4 : Delete GAT from list

Note : must know the previous element

4.2 Representing Lists in C++

- Defining a Node in C++
  - Class definition for 3-letter node

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

4.2 Representing Lists in C++

- Defining a Node in C++
  - A more complicated list structure

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

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

4.2 Representing Lists in C++

- Defining a Node in C++
  - A more complicated list structure

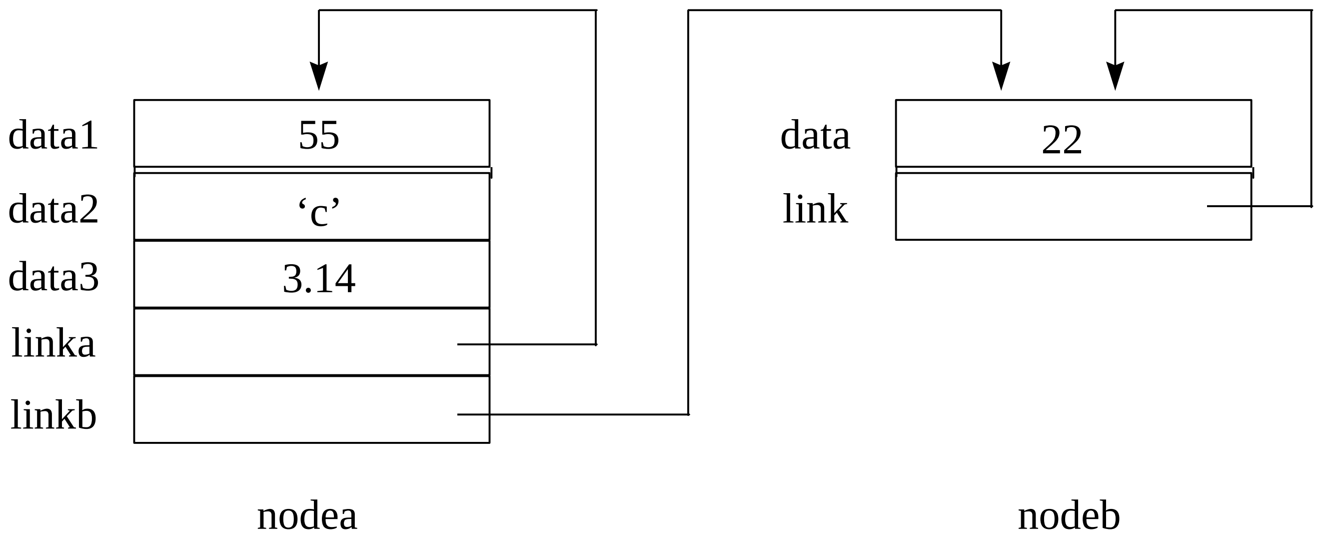


Figure 4.5 : Illustration of the code structures NodeA and NodeB

4.2.2 Designing a Chain Class in C++

- Design approach
  - use a class ThreeLetterChain corresponding to the entire list data structure
  - ThreeLetterChain supports member functions for list manipulation operations
  - use a composite of two classes, ThreeLetterNode and ThreeLetterChain
  - ThreeLetterChain HAS-A ThreeLetterNode

4.2.2 Designing a Chain Class in C++

- Definition
  - a data object of type A HAS-A data object of type B if A conceptually contains B

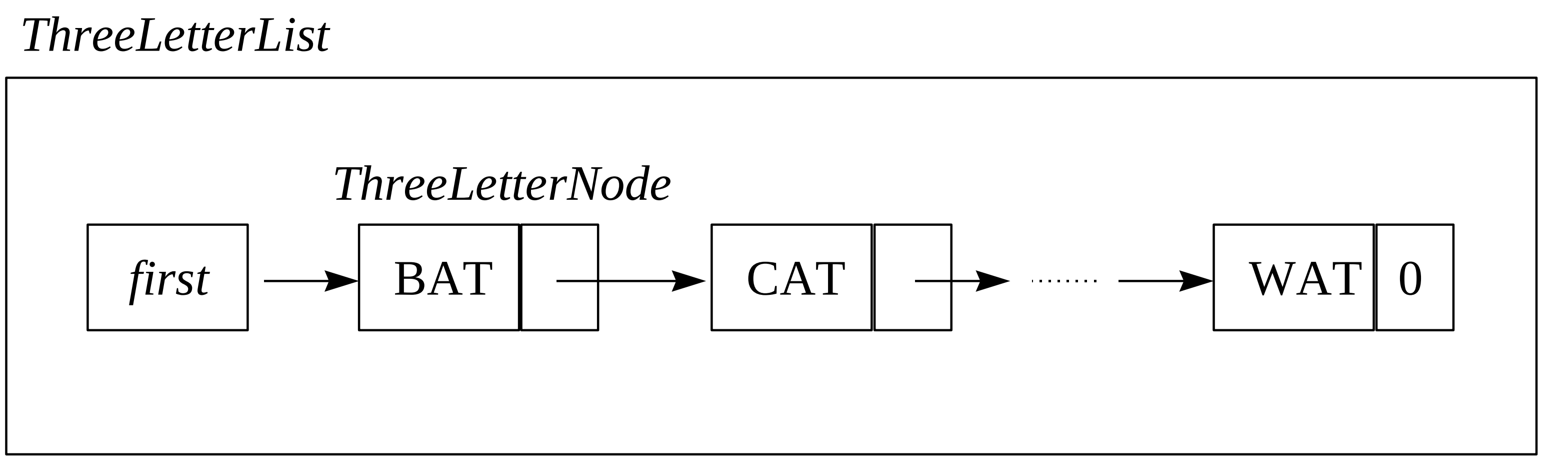


Figure 4.7 : Conceptual relationship between ThreeLetterList and ThreeLetterNode

4.2.2 Designing a Chain Class in C++

- ThreeLetterChain
  - contains only the pointer first
  - declare to be a friend of ThreeLetterNode
  - only member functions of ThreeLetterChain and ThreeLetterNode can access the private members of ThreeLetterNode
  - only list manipulation operations have access to data members of ThreeLetterNode

4.2.2 Designing a Chain Class in C++

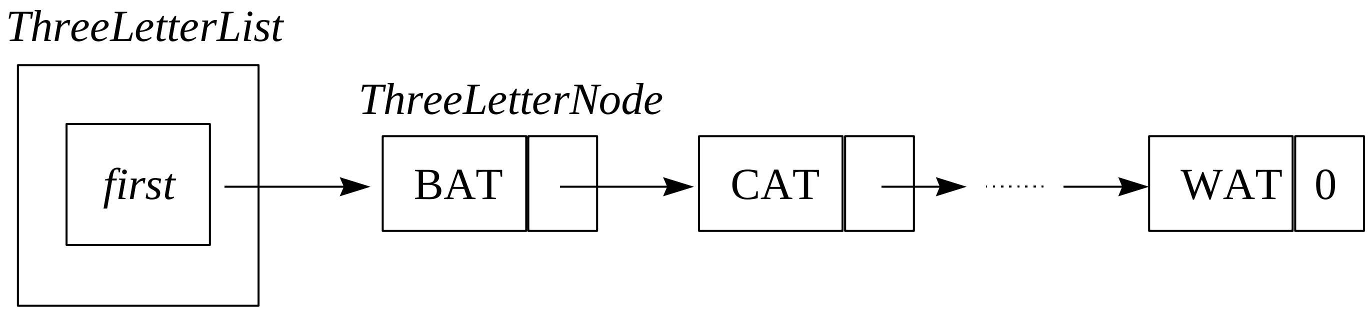


Figure 4.8 : Actual relationship between ThreeLetterChain and ThreeLetterNode

4.2.2 Designing a Chain Class in C++

```
class ThreeLetterChain;        // forward declaration
class ThreeLetterNode {
friend class ThreeLetterChain;
private:
        char data[3];
        ThreeLetterNode *link;
};
class ThreeLetterChain {
public:
        // Chain Manipulation operations
        ...
private:
        ThreeLetterNode  *first;
};
```

Program 4.1 : Composite classes

4.2.2 Designing a Chain Class in C++

- Nested Classes
  - one class is defined inside the definition of another class

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

Program 4.2 : Nested classes

4.2.2 Designing a Chain Class in C++

- Nested Classes
  - ThreeLetterNode objects cannot be accessed outside class ThreeLetterChain
  - ThreeLetterNode data members are public, so they can be accessed by member functions of ThreeLetterChain
- Using composite classes, the node class can be used by two data structures

4.2.3 Pointer Manipulation in C++

- *new* command

```
ThreeLetterNode* f;
NodeA* y;
NodeB* z;
f = new ThreeLetterNode;
y = new NodeA;
z = new NodeB;
```

- *f : the node of type ThreeLetterNode

4.2.3 Pointer Manipulation in C++

- *delete* command
  - delete f;   delete y;   delete z;
- *null* command
  - constant 0

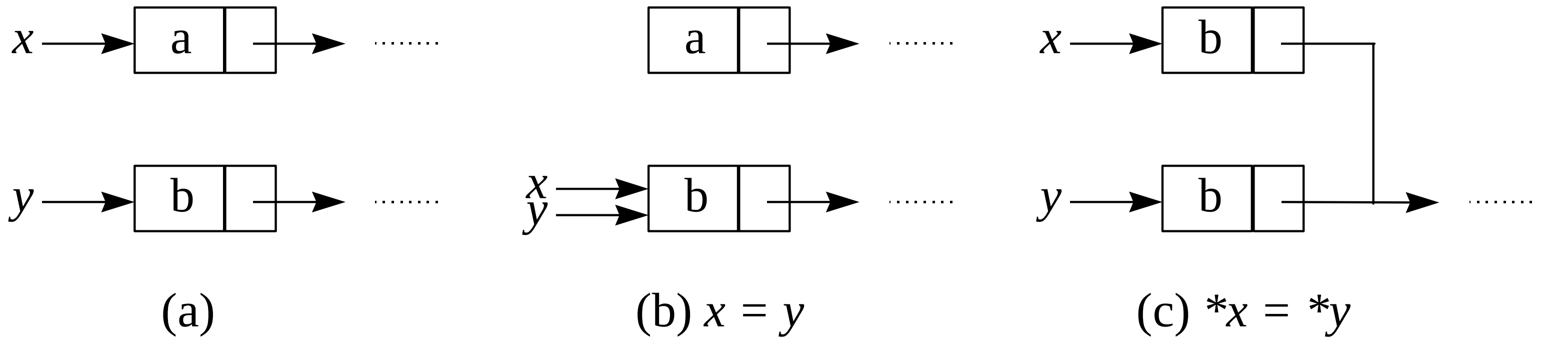


Figure 4.9 : Effect of pointer assignments

4.2.4 Chain Manipulation Operations

- Implementation uses pointer manipulation operations

```
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;
};
```

4.2.4 Chain Manipulation Operations

- Example

```
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);
}
```

Program 4.3 : Creating a two-node list

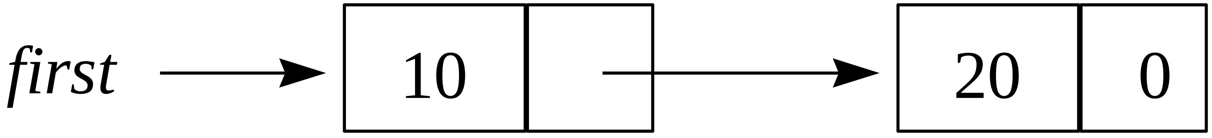


Figure 4.10 : A two-node list

4.2.4 Chain Manipulation Operations

- Example

```
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);
}
```

Program 4.4 : Inserting a node

4.2.4 Chain Manipulation Operations

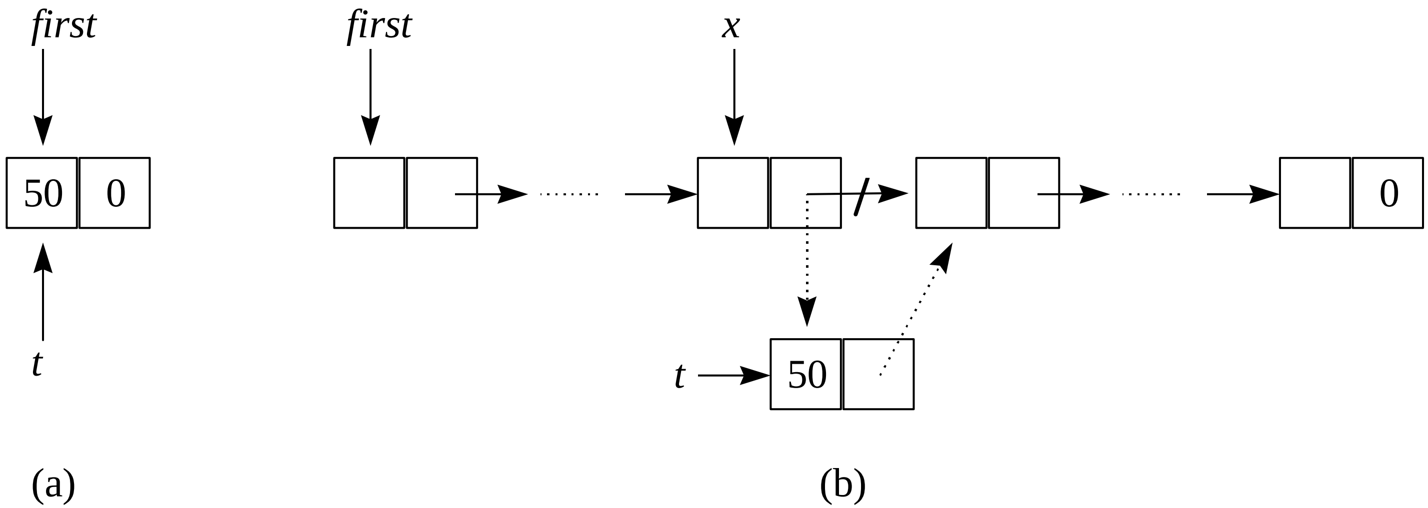


Figure 4.11 : Inserting into an empty and nonempty list

4.2.4 Chain Manipulation Operations

```
void  Chain::Delete(ChainNode *x, ChainNode *y)
{
        if(x==first) first = first->link;
        else  y->link = x->link;
        delete x;
}
-----------------------------------
Program 4.5 : Deleting a node
```

4.3 The Template Class Chain

- Implementing Chains with Templates
  - Linked-list
    - a container class
      - good for implementation with templates

4.3 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 {
public:
     Chain() {first = 0;};  // constructor initializing first to 0
     // Chain manipulation operations
             .
             .
private:
     ChainNode<T> *first;
};
----------------------------------------------
Program 4.6 : Template definition of chains
```

- an empty linked list of integers intlist   Chain<int> intlist;

4.3.2 Chain Iterators

- Iterator
  - an object that is used to traverse all the elements of a container class
- Example operations on an integer container class C
  - print all integers in C
  - obtain the max, min, mean, median, or mode of all integers in C
  - obtain the sum or product of all integers in C

4.3.2 Chain Iterators

- Pseudo-code for the operations

```
initialization step;
for each item in C
{
        currentItem = current item of C;
        do something with currentItem
}
postprocessing step;
```

- All operations have to be implemented as member functions of a particular container class

4.3.2 Chain Iterators

- Drawbacks
  - many operations do not make sense to certain object types
  - too many operations in a class
  - users have to learn how to traverse the container class
- ChainIterator<T>
  - handles details of the linked list traversal
  - retrieves elements stored in the list

4.3.2 Chain Iterators

- C++ Iterators
  - A Pointer to an element of an object
  - Go through the elements of the object one by one

```
void main()
{
        int x[3] = {0,1,2};
        // use a pointer y to iterate through the array x
        for (int* y = x; y != x + 3; y++)
                cout << *y << " ";
        cout << endl;
}
```

Program 4.8: Using an array iterator

4.3.2 Chain Iterators

- C++ Iterators (cont)
  - Output all elements in the range [start, end)

```
for ( Iterator i=start; i!=end; i++ )
        cout << *i << " ";
```

  - C++ STL defines five categories of iterators
    - input, output, forward, bidirectional, random access
  - Many algorithms are defined in STL
    - copy, sort, find, search, etc
    - These algorithms were implemented using iterators

4.3.2 Chain Iterators

- C++ Iterators (cont)

```
template <class Iterator>
void copy(Iterator start, Iterator end, Iterator to)
{// copy from [start, end) to [to, to+end-start)
      while ( start != end )
           { *to  = *start; start++; to++; }
}
```

Program 4.9: Possible code for STL copy function

- Forward Iterator for Chain
  - Implement a forward iterator class for chain

4.3.2 Chain Iterators

```
class ChainIterator {
public:
    // typedefs required by C++ for a forward iterator 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;
            }
-----------------------------------------
```

Program 4.10(1): A forward iterator for Chain

4.3.2 Chain Iterators

```
    bool operator!=(const ChainIterator right) const
            { return current != right.current; }
    bool operator==(const ChainIterator right) const
            { return current == right.current; }
    private:
            ChainNode<T>* current;
};
-----------------------------------------
Program 4.10(2): A forward iterator for Chain
```

- Additionally, add following public member functions

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

4.3.3 Chain Operations

- Operations included in most reusable classes
  - constructors (including default and copy constructors)
  - a destructor
  - operator=
  - operator==
  - operators to input and output a class object
    (by overloading operator>> and operator<<)

4.3.3 Chain Operations

- Operations in a Chain class
  - insertion, deletion, other manipulations

```
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);
}
```

Program 4.11 : Inserting at the back of a list

  - last : a private data member in Chain<T>

4.3.3 Chain Operations

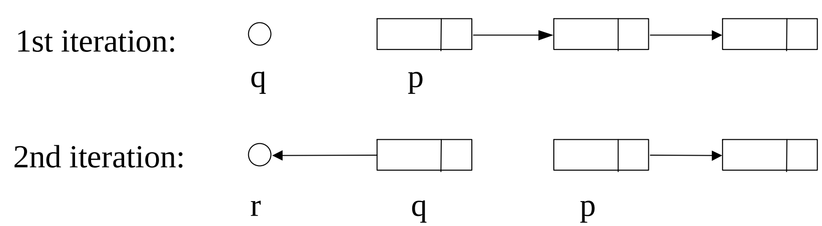
```
template <class T>
void Chain<T>::Concatenate(Chain<T>& b)
{ // b is concatenated to the end of *this
        if (first) { last->link = b.first; last = b.last; }
        else { first = b.first; last = b.last; }
        b.first = b.last = 0;
}
```

Program 4.12 : Concatenating two chains

4.3.3 Chain Operations

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

Program 4.13 : Reverse a list



4.3.4 Reusing a Class

- Scenarios should not reuse class
  - Sometimes, less efficient than directly implementing class
  - When the operations required by the application are complex and specialized

4.4 Circular Lists

- Circular list
  - link field of the last node points to the first node in the list
  - to check whether current points to the last node:
    current→link==first

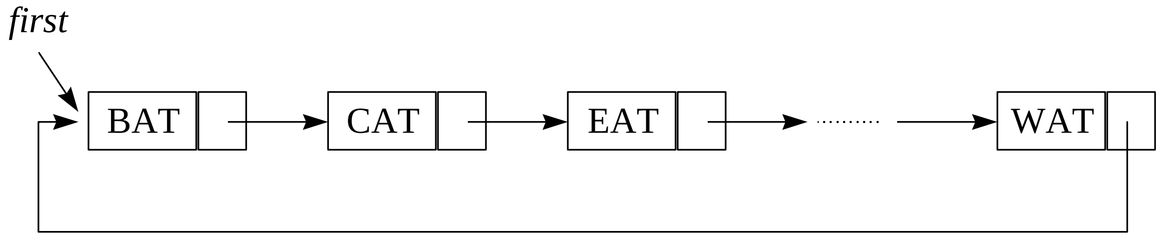


Figure 4.13 : A circular list

4.4 Circular Lists

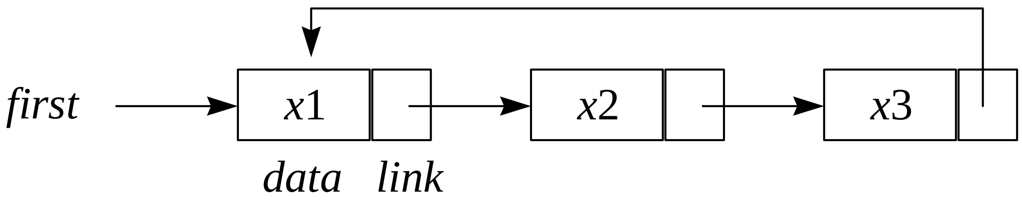


Figure 4.14 : Example of a circular list

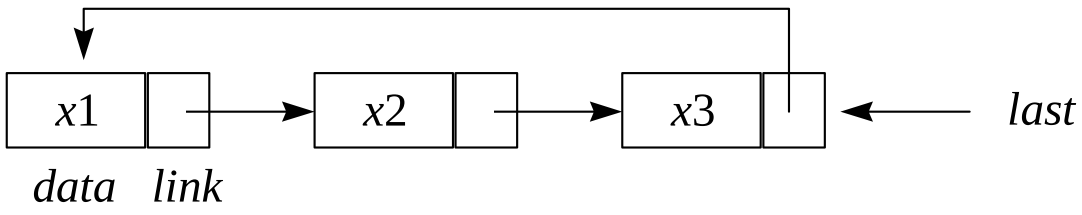


Figure 4.15 : Pointing to the last node of a circular list

4.4 Circular Lists

```
template <class T>
void CircularList::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 { // empty list
                last = newNode;
                newNode->link = newNode;
        }
}
```

Program 4.14 : Inserting at the front of circular list

- last : the private data member that points to the last node
- a dummy head node : to avoid a case in which the empty list is handled specially

4.4 Circular Lists

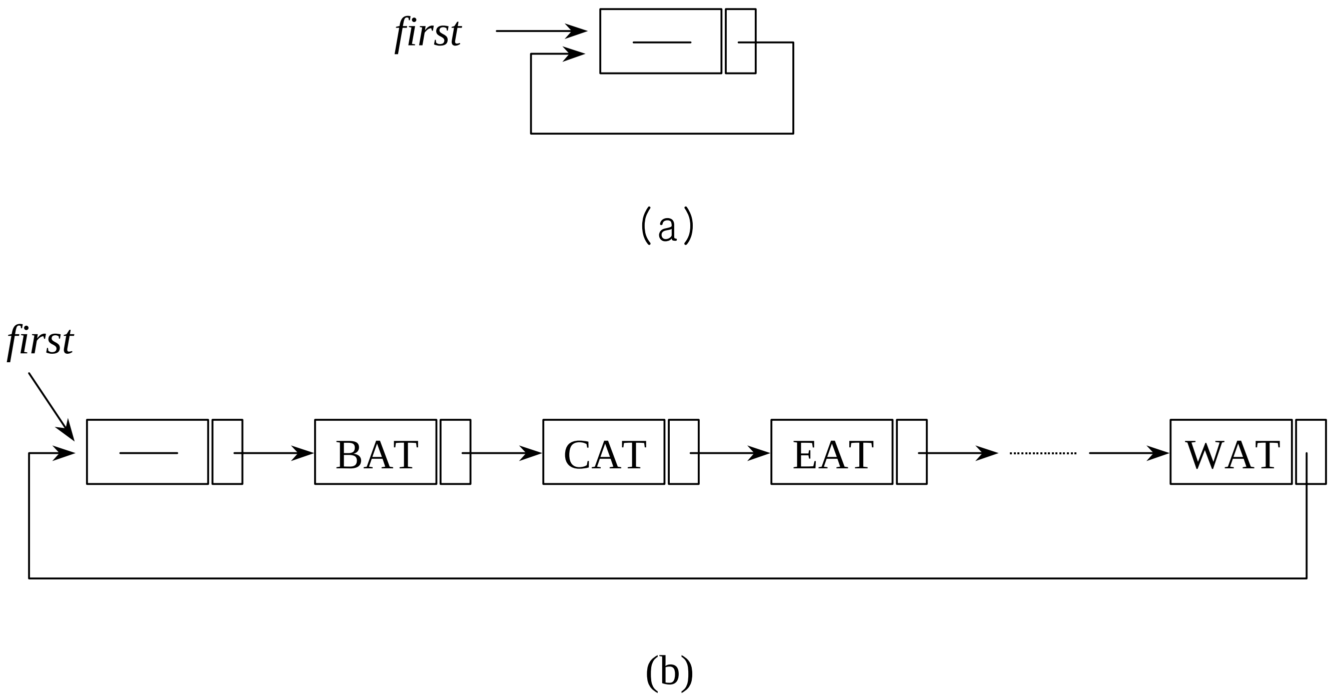
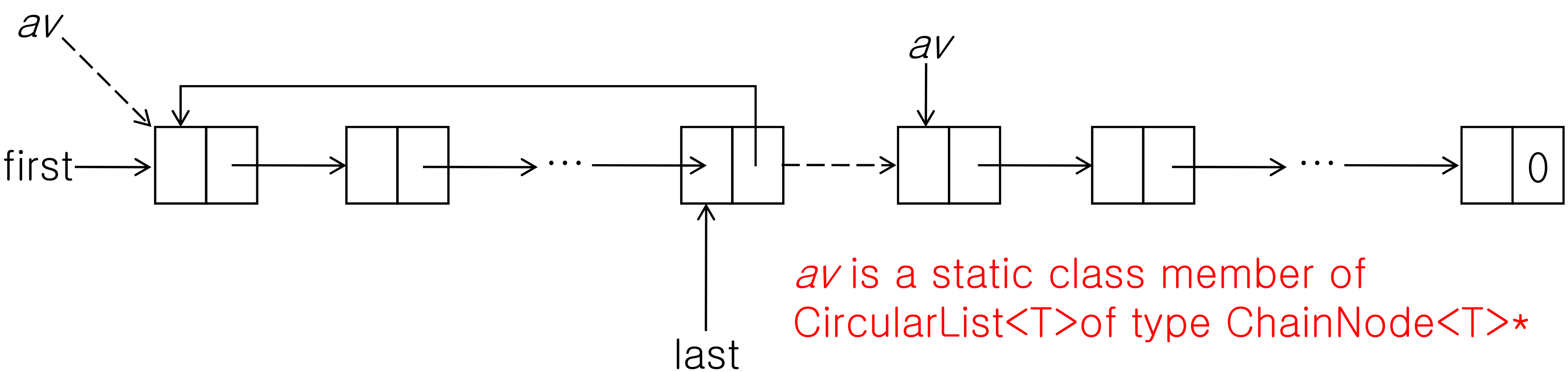


Figure 4.16 : A circular list with a head node

4.5 Available Space Lists

- Destructors for chains
  - It takes linear time to delete nodes
  - The run time may be able to be reduced
  - Do NOT erase 'target node' immediately
  - Using pointer *av* to point 'unused nodes'



av is a static class member of CircularList<T>of type ChainNode<T>*

Figure 4.17: Dashed arrows indicate changes involved in deleting a circular list

4.5 Available Space Lists

```
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;
}
-----------------------------------
```

Program 4.15: Getting 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;
}
-----------------------------------
```

Program 4.16: Returning a node

4.5 Available Space Lists

```
template <class KeyT>
void CircularList<T>::~CircularList()
{ // Delete the circular list
      if(last) {
            ChainNode<T>* first = last->link;
            last->link = av; // last node linked to av
            av = first;
            last = 0;
}
----------------------------------
Program 4.17: Deleting a circular list
```

4.6 Linked Stacks and Queues

- How to implement them using lists

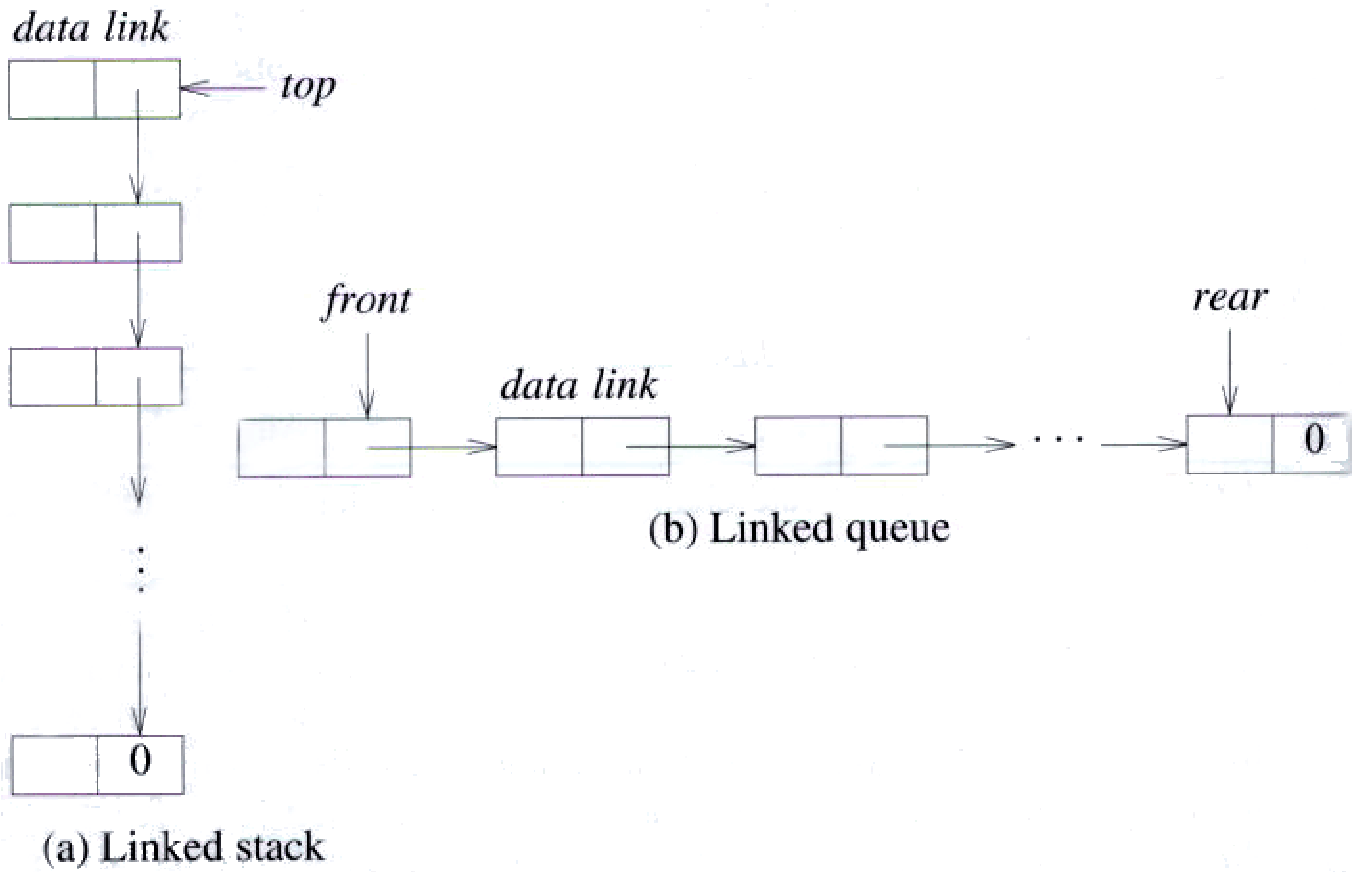


Figure 4.18: Linked stack and queue

4.6 Linked Stacks and Queues

```
template <class T>
void LinkedStack<T>::Push(const T& e) {
        top = new ChainNode<T>(e, top);
}
-----------------------------------
Program 4.19: Adding to a linked stack
```

```
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;
}
----------------------------------------------
Program 4.20: Deleting from a linked stack
```

4.6 Linked Stacks and Queues

```
template <class T>
void LinkedQueue<T>::Push(const T& e) {
        if(IsEmpty()) front = rear = new ChainNode(e,0);
        else rear = rear->link = new ChainNode(e,0);
}
--------------------------------------------
Program 4.21: Adding to a linked queue
```

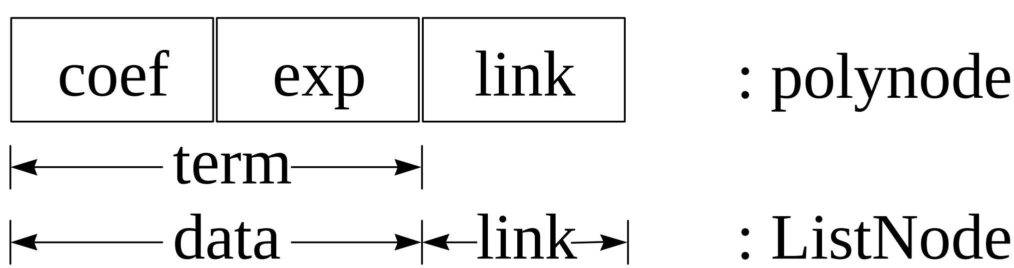
```
template <class T>
void LinkedQueue<T>::Pop()
{ // Delete first element in queue.
        if (IsEmpty()) throw "Queue is empty. Cannot delete.";
        ChainNode<T>* delNode = front;
        front = front->link;
        delete delNode;
}
------------------------------------------------------------
Program 4.22: Deleting from a linked queue
```

4.7 Polynomials

- 4.7.1 Polynomial Representation
  - Polynomial class
    - a polynomial is represented by a list
    - Polynomial IS-IMPLEMENTED-BY List
  - Definition
    - a data object of Type A IS-IMPLEMENTED-BY a data object of Type B if the Type B object is central to the implementation of the Type A object.

4.7 Polynomials

- 4.7.1 Polynomial Representation
  - Implementation
    - linked list object poly is data member of Polynomial
    - list template Type instatiation
      struct term



4.7 Polynomials

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

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

Program 4.23 : Polynomial class definition

4.7 Polynomials

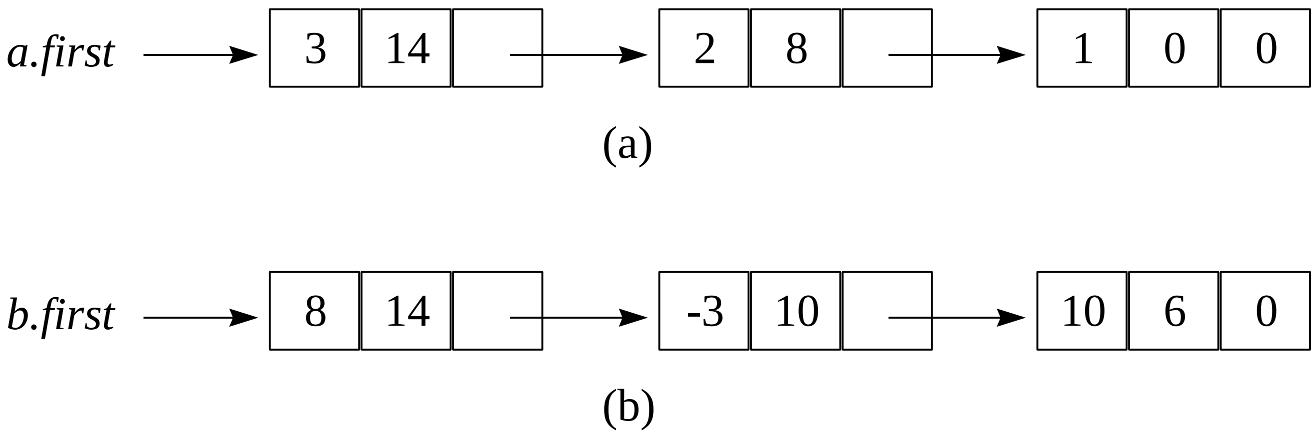


Figure 4.19 : Polynomial representation of $3x^{14}+2x^{8}+1$ and $8x^{14}-3x^{10}+10x^{6}$

4.7.2 Adding Polynomials

- Add polynomials a and b
  - use list iterators Aiter and Biter to examine their terms
  - two variables p and q are used to move along the terms of a and b
  - if p→exp == q→exp
    - coefficients are added
    - a new term is created for the result polynomial c if the sum is not zero
  - if p→exp < q→exp
    - copy of the term in b is attached to c
    - q is advanced
  - if p→exp > q→exp
    - similar action is taken on a

4.7.2 Adding Polynomials

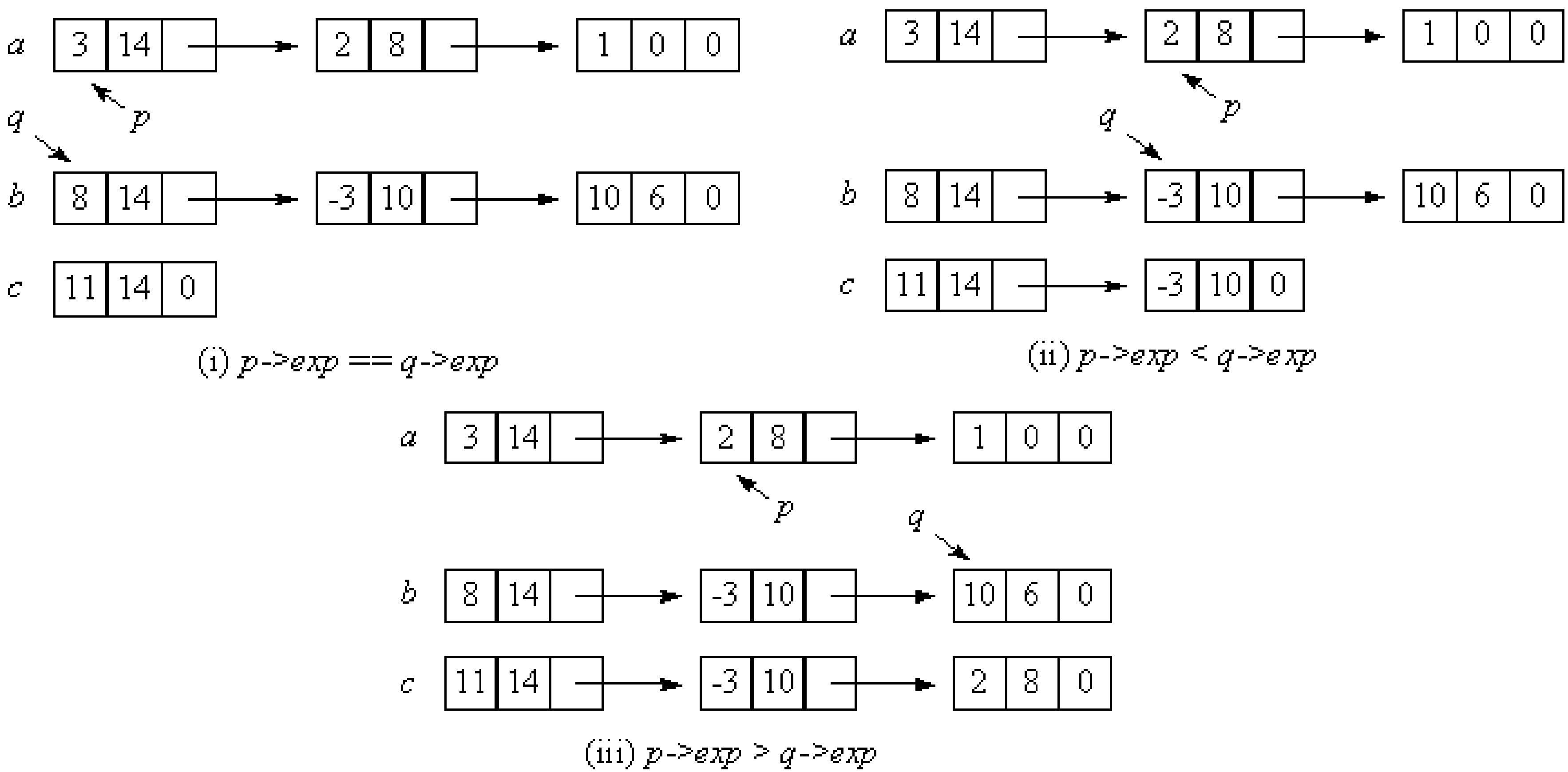


Figure 4.20 : Generating the first three terms of c=a+b

4.7.2 Adding Polynomials

```
Polynomial operator+(const Polynomial& b) const
{ // Polynomials *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 && bi) { // current node is not null
                if ( ai->exp == bi->exp ) {
                    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 ) {
                     c.poly.InsertBack(temp.Set(bi->coef, bi->exp));
                     bi++; // next term of a
              }
              else {
                     c.poly.InsertBack(temp.Set(ai->coef, ai->exp));
                     ai++; // next term of a
              }
         }
```

4.7.2 Adding Polynomials

```
        while (ai) { // copy rest of 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;
}
```

Program 4.24 : Adding two polynomials

4.7.2 Adding Polynomials

- Analysis of operator+( )
  - operations contributing to the computing time :
    - coefficient additions
    - exponent comparisons
    - additions/deletions to available space
    - creation for new nodes for c
  - assume a and b have m and n terms
  - $0 \le$ coefficients additions $\le \min\{m, n\}$
  - exponent comparisons $< m+n$
  - the computing time is $O(m+n)$
  - algorithm operator+( ) is optimal within a constant factor

4.7.3 Circular List Representation of Polynomials

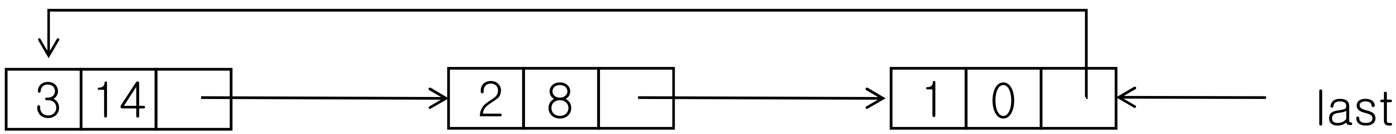
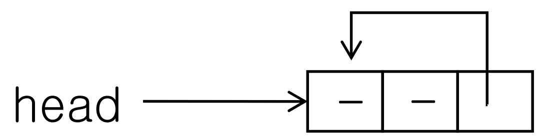
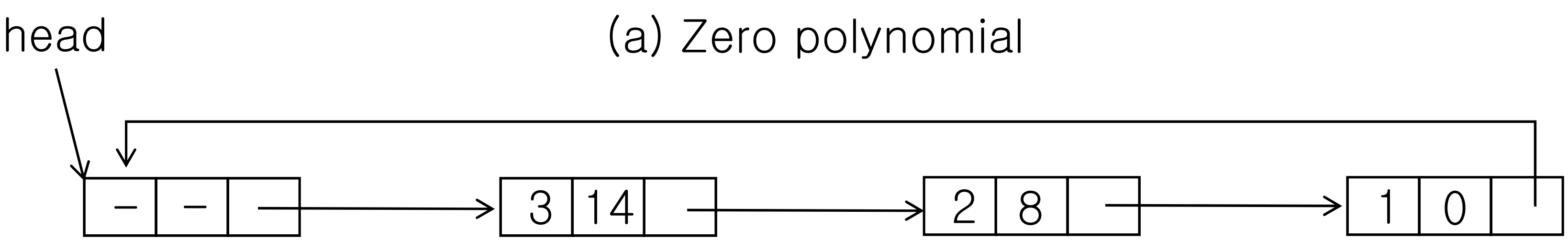


Figure 4.21: Circular list representation of $3x^{14}+2x^{8}+1$

- This structure causes some problems during addition and other polynomial operations. So suggest other structure



(a) Zero polynomial



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

Figure 4.22: Example polynomials

4.7.3 Circular List Representation of Polynomials

```
Polynomial Polynomial::operator+(const Polynomial& b) const
{ // Polynomials *this(a) and b are added and the sum returned.
      Term temp;
      CircularListWithHeader<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++; // advance 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
            }
      }
}
```

Program 4.25: Adding two polynomials represented as circular lists with header nodes

4.8 Equivalence Class

- Definition
  - A relation ≡ over a set S, is said to be an equivalence relation over S if it is symmetric, reflexive, and transitive over S
  - Example
    - 0≡4, 3≡1, 6≡10, 8≡9, 7≡4, 6≡8, 3≡5, 2≡11,11 ≡0
      -> {0, 2, 4, 7, 11}; {1, 3, 5}; {6, 8, 9, 10};

4.8 Equivalence Class

- Algorithm
  1. Read the equivalence pairs (i, j) and sort
  2. Begin at 0 and find all pairs (0, j)
     - j s are in the same set as 0
  3. Find other pairs (j, k)
     - k s are in the same set as 0
  4. Continue until the entire equivalence class containing 0 has been found
  5. Move other set and Repeat 2-4 steps

4.8 Equivalence Class

```
Void Equivalence()
{
        initialize;
        while more pairs
        {
            input the next pair(i,j);
            process this pair;
        }
        initialize for output
        for (each object not yet output)
            output the equivalence class that contains this object
```

Program 4.26: First version of equivalence algorithm

- Let m and n represent the number of input pairs and objects
- Need 2D Boolean array, *pairs[n][n]*
- The element *pairs[i][j]*=**true** is an input pair
- Wasteful of space and $O(n^2)$ time complexity

4.8 Equivalence Class

```
Void Equivalence() {
        read n;      // read in number of objects
        initialize first[0:n-1] to 0 and out[0:n-1] to false;
        while more pairs {   // input pairs
            read the next pair (i,j);
            put j on the chain first[i];
            put i on the chain first[j];
        }
        for (i=0; i<n; i++)
            if(!out[i]){
                    out[i]=true;
                    output the equivalence class that contains this object i;
            }
}
```

Program 4.27: A more detailed version of equivalence algorithm

- Consider chain representation
- 1D-array and chain structure

4.8 Equivalence Class

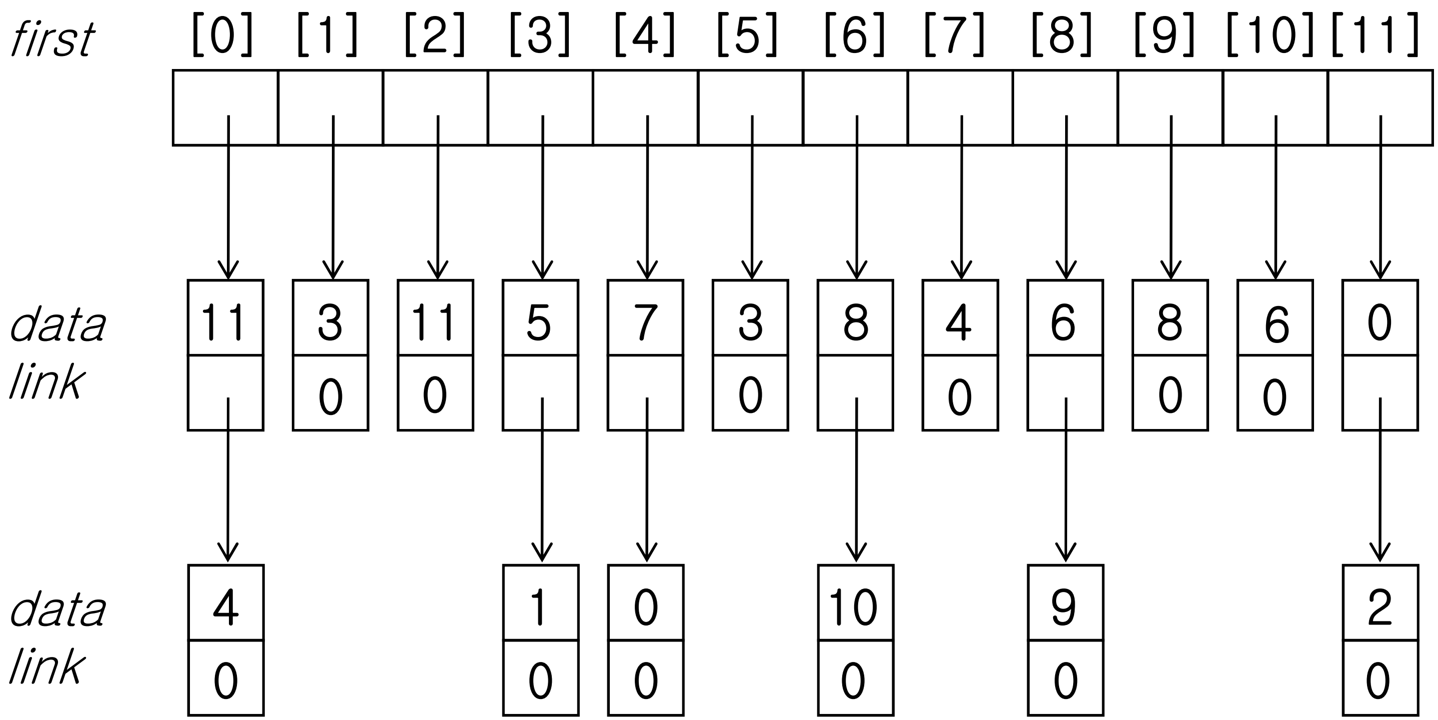


Figure 4.23: Lists after pairs have been input

4.8 Equivalence Class

```
Class ENode {
friend void Equivalence();
public:
      ENode(int d = 0, 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( "eqiv.in", ios::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];
```

4.8 Equivalence Class

```
bool *out = new bool[n];
// use STL function fill to initialize
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]) { // needs to be output
       cout << endl << "A new class : " << i;
       out[i] = true;
       ENode *x = first[i]; ENode *top = 0; // initialize stack
```

4.8 Equivalence Class

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

```
    for ( i=0; i<n; i++ )
        while (first[i]) {
            ENode *delnode = first[i];
            first[i] = delnode->link;
            delete delnode;
        }
    delete [] first; delete [] out;
}
```

Program 4.28: C++ function to find equivalence calsses

4.8 Equivalence Class

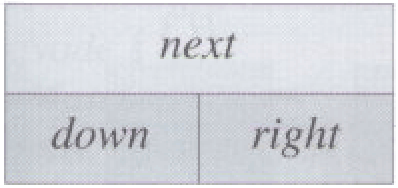
- Analysis of Equivalence
  - Initialize first and out takes O(n) time
  - Phase 1 takes a constant time O(m)
  - Phase 2 each node is put onto the linked stack at most once
  - Totally, O(m+n) time
  - Space required is also O(m+n)

4.9 Sparse Matrices

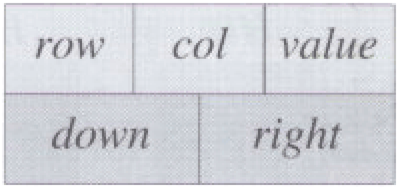
- “Matrices are sparse” means…
  - Many of the entries are zero
  - Nonzero terms are scattered
- Waste of memory
  - Zero terms are usually no meaning
- Many useless operations on zero term
  - Takes long time to operate
- Keep nonzero term using Linked list

4.9 Sparse Matrices

- Each nonzero terms has three members
  - Row, Column, Value
  - down : link to the next nonzero term in the same column
  - right : link to the next nonzero term in the same row
- Nodes type is *MatrixNode*
  - Header node, Element node
  - They can be distinguished by *head* field



(a) header node



(b) element node

4.9 Sparse Matrices

```
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
};
```

Program 4.29: Class definitions for sparse matrices

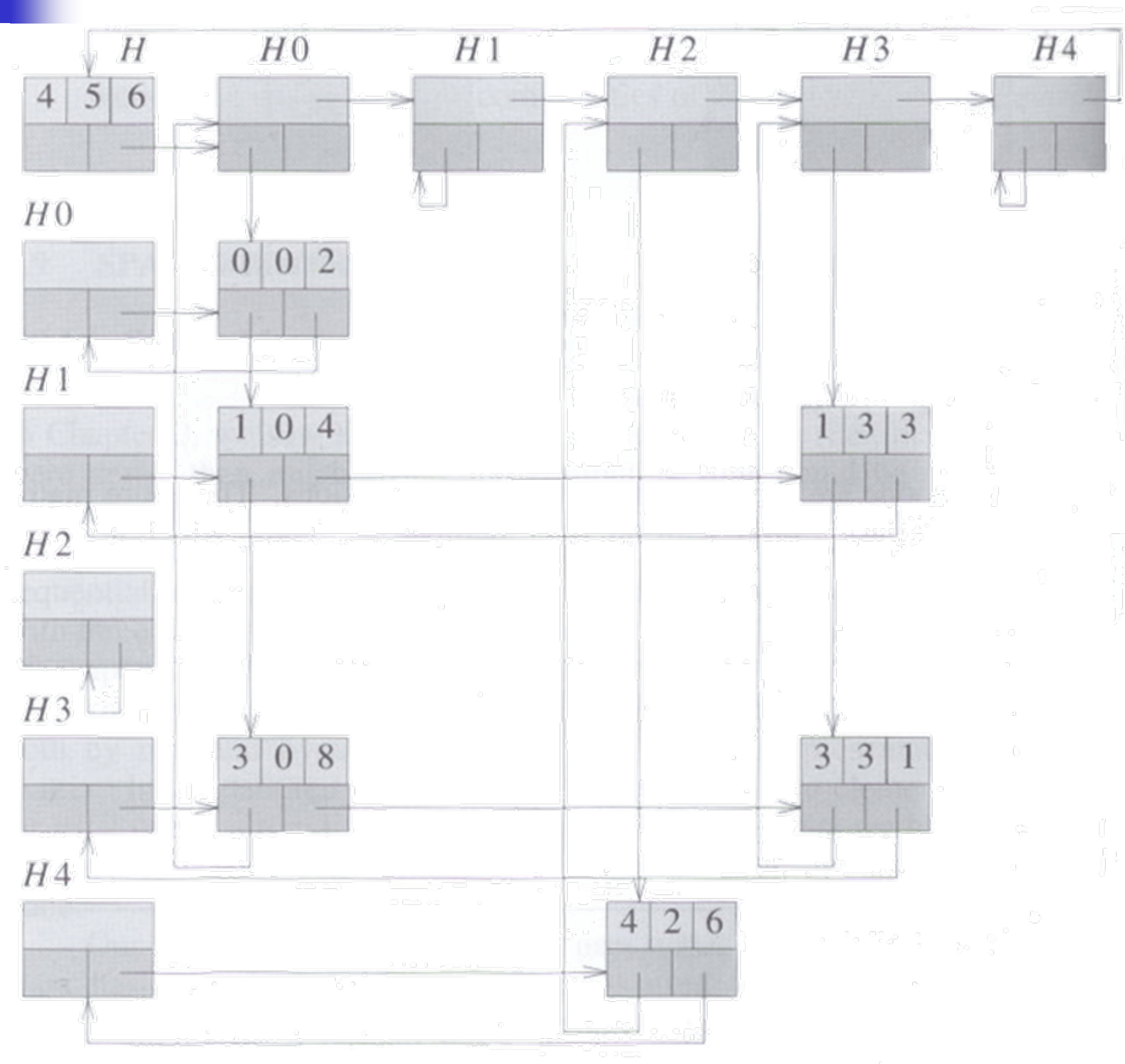
4.9 Sparse Matrices

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

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

Program 4.29: Class definitions for sparse matrices

4.9 Sparse Matrices



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

4.9.2 Sparse Matrix Input

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

- Input information
  - row, col, value
    - ex) <u>5, 4, 6</u>; 0, 0, 2; 1, 0, 4; 1, 3, 3; ....

5X4 matrix and 6 elements

elements(i,j,data)...

- Function operator>>
  - Create header nodes
  - Input the matrix elements
  - Link header nodes together

4.9.2 Sparse Matrix Input

```
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 header node for list of header nodes
     matrix.headnode = new MatrixNode(false, &s);
     if ( p == 0) { matrix.headnode->right = matrix.headnode; return is; }
     // at least one row or column
     MatrixNode *head = new MatrixNode[p];
     for ( int i = 0; i < p; i++ )
         head[i] = new MatrixNode(TRUE, 0);
     int currentRow = 0;
     MatrixNode *last = head[0]; // last node in current row
```

Program 4.30(1): Reading in a sparse matrix

4.9.2 Sparse Matrix Input

```
for ( int 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); // link new node into row list
     head[t.col]->next = head[t.col]->next->down = last; // link into column list
  } // end of for
last->right = head[currentRow]; // close last row
for ( int i=0; i<s.col; i++ ) head[i]->next->down = head[i];
// link the header nodes together
for ( int i=0; i<p-1; i++ ) head[i]->next = head[i+1];
head[p-1]->next = matrix.headnode;
matrix.headnode->right = head[0];
delete [] head;
return is;
}
```

Program 4.30(2): Reading in a sparse matrix

4.9.2 Sparse Matrix Input

- Analysis of operator>>
  - Assume **new** works in O(1) time
  - All header node may be set up in O(max{n,m}) time
  - Each nonzero term is set up in O(1) time
  - for loop of lines 15–26 takes O(r) time
  - The rest of the algorithm takes O(max{n,m})
  - The total time is O(max{n,m} + r)=O(n+m+r)
  - This time is better than 2D array(O(nm))
  - But, slightly worse than sequential sparse method of Section 2.3

4.9.3 Deleting a Sparse Matrix

- Instead of **delete**, using *av* points
  - First to last through *right*

```
Matrix::~Matrix()
{ // Return all nodes to the av list. This list is a chain linked via the right
  // field. av is a static variable that points to the first node of the av list.
        if (!headnode) return;
        MatrixNode *x = headnode->right;
        headnode->right = av; av = headnode; // return headnode
        while ( x!=headnode ) { // erase by rows
              MatrixNode *y = x->right;
              x->right = av;
              av = y;
              x = x->next; // next row
        }
        headnode = 0;
}
```

Program 4.31: Deleting a sparse matrix

4.9.3 Deleting a Sparse Matrix

- Analysis of ~Matrix()
  - Each row list is circularly linked through the field right
  - Do not need erase one by one
  - Totally O(n+m) time

4.10 Doubly Linked Lists

- Problems with singly linked lists
  - can move only in the direction of the links
  - deletion requires knowing the preceding node
- Node
  - at least three fields : data, llink(left link), rlink(right link)

4.10 Doubly Linked Lists

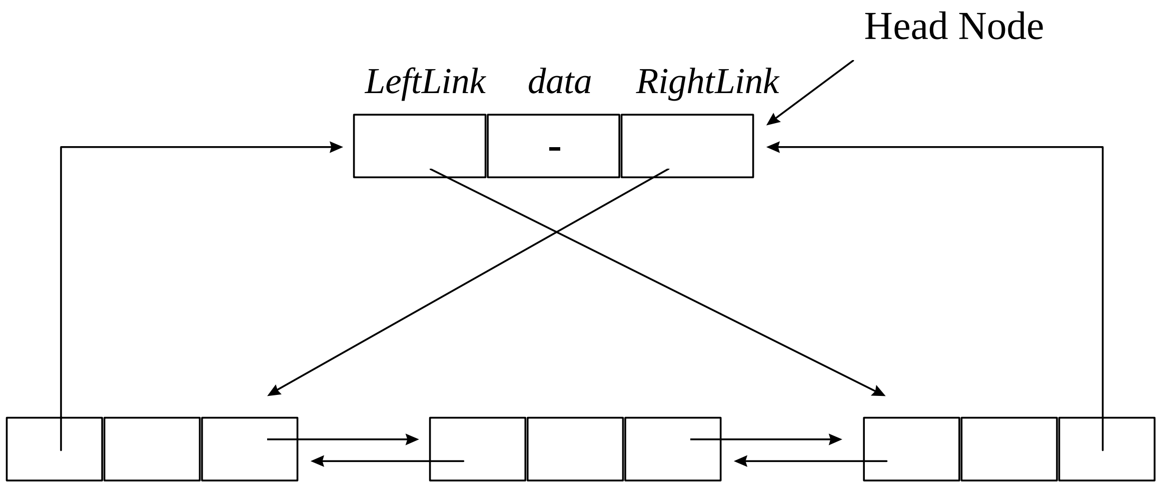


Figure 4.27 : Doubly linked circular list with head node

4.10 Doubly Linked Lists

- Head node
  - convenient for algorithms
  - data field usually contains no information
- Essential virtue
  - one can go back and forth with equal ease
  - $p == p \rightarrow llink \rightarrow rlink == p \rightarrow rlink \rightarrow llink$

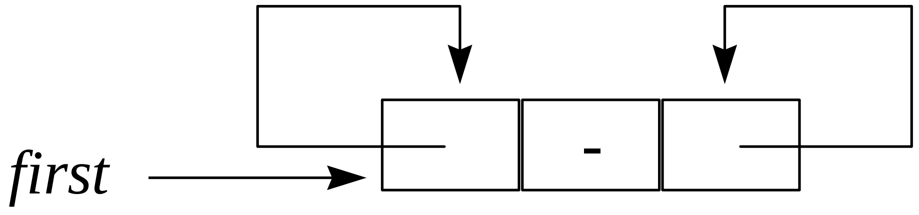


Figure 4.28 : Empty doubly linked circular list with head node

4.10 Doubly Linked Lists

```
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
};
```

Program 4.32 : Class definition of a doubly linked list

4.10 Doubly Linked Lists

```
void DblList::Delete(DblListNode *x)
{
  if ( x == first ) throw "Deletion of headnode not permitted“;
  else {
    x->left->right = x->right;
    x->right->left = x->left;
    delete x;
  }
}
```

Program 4.33 : Deletion from a doubly linked circular list

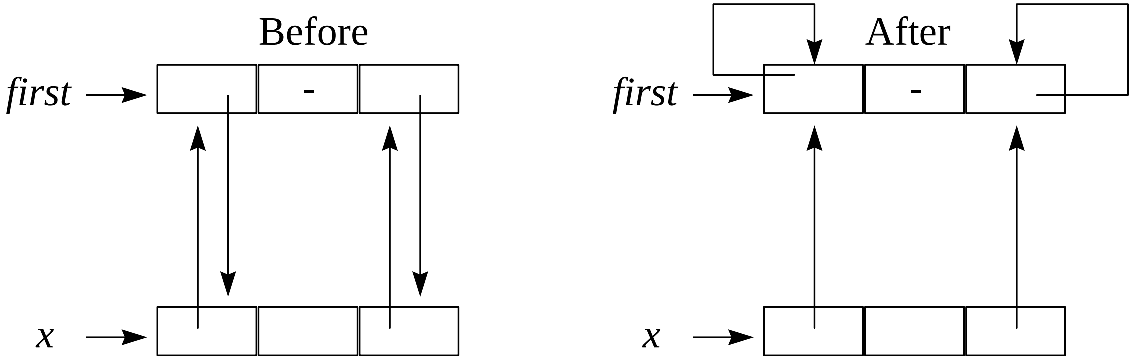
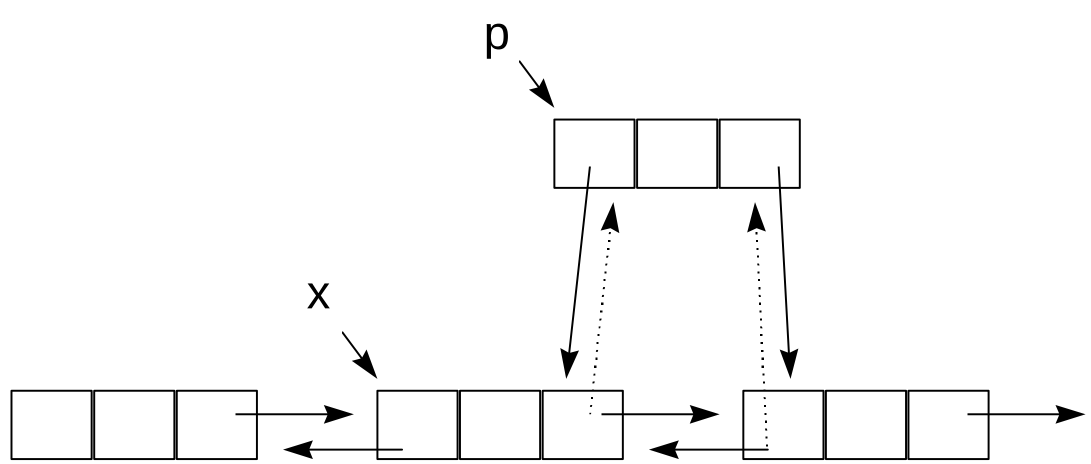


Figure 4.31 : Deletion from a doubly linked circular list

4.10 Doubly Linked Lists

```
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;
}
```

Program 4.34 : Insertion into a doubly linked circular list



4.11 Generalized Lists

- Definition
  - A generalized list is a finite sequence of $n \geq 0$ elements, a_0, …, a_{n-1}, where a_i is either an atom or list
  - Example
    - (1) A = () : the null, or empty
    - (2) B = (a,(b,c)) : a list of length two; first element atom a, and second element list (b,c)
    - (3) C = (B, B, ()) : a list of length three; first two elements are the list B, and the third element is the null list
    - (4) D = (a, D) : a recursive list of length two;
D = (a, (a, (a, …)

4.11 Generalized Lists

- Polynomial representations

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

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

  - Second form can be represented by generalized Lists

- Definition of *PolyNode*

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

4.11 Generalized Lists

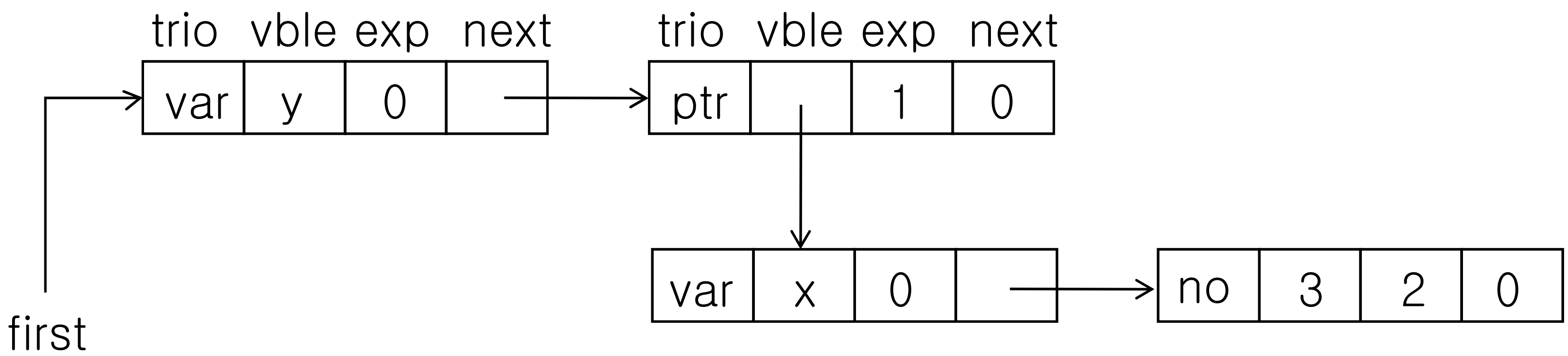


Figure 4.30 : Representation on $3x^2y$

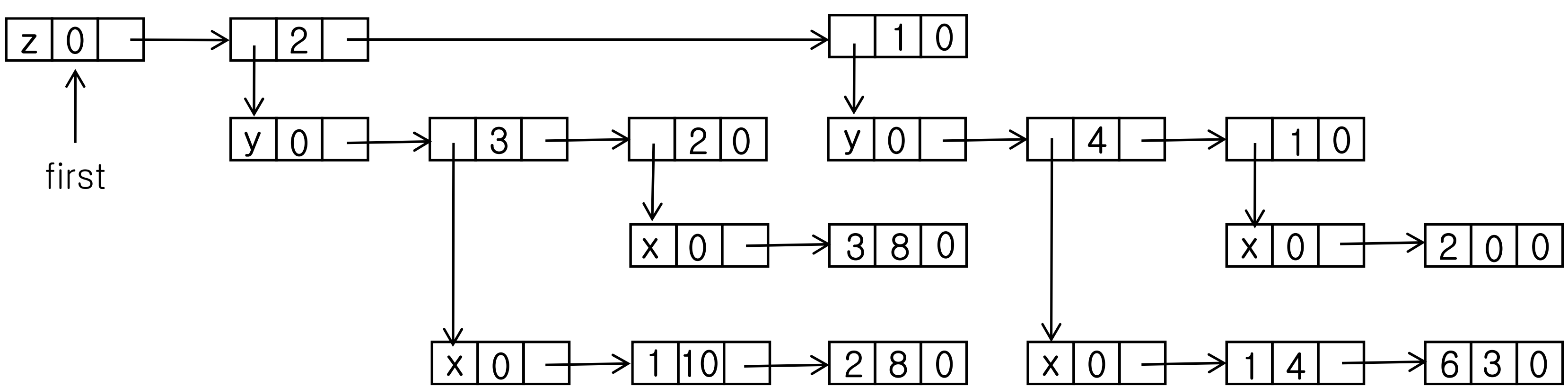


Figure 4.31 : $((x^{10}+2x^{8})y^{3}+3x^{8}y^{2})z^{2}+((x^{4}+6x^{3})y^{4}+2y)z$ (trio field is omitted)

4.11 Generalized Lists

- This data structure may be defined in C++

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

template<class T>
class GenListNode {
friend class GenList<T>;
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;
};
```

4.11 Generalized Lists

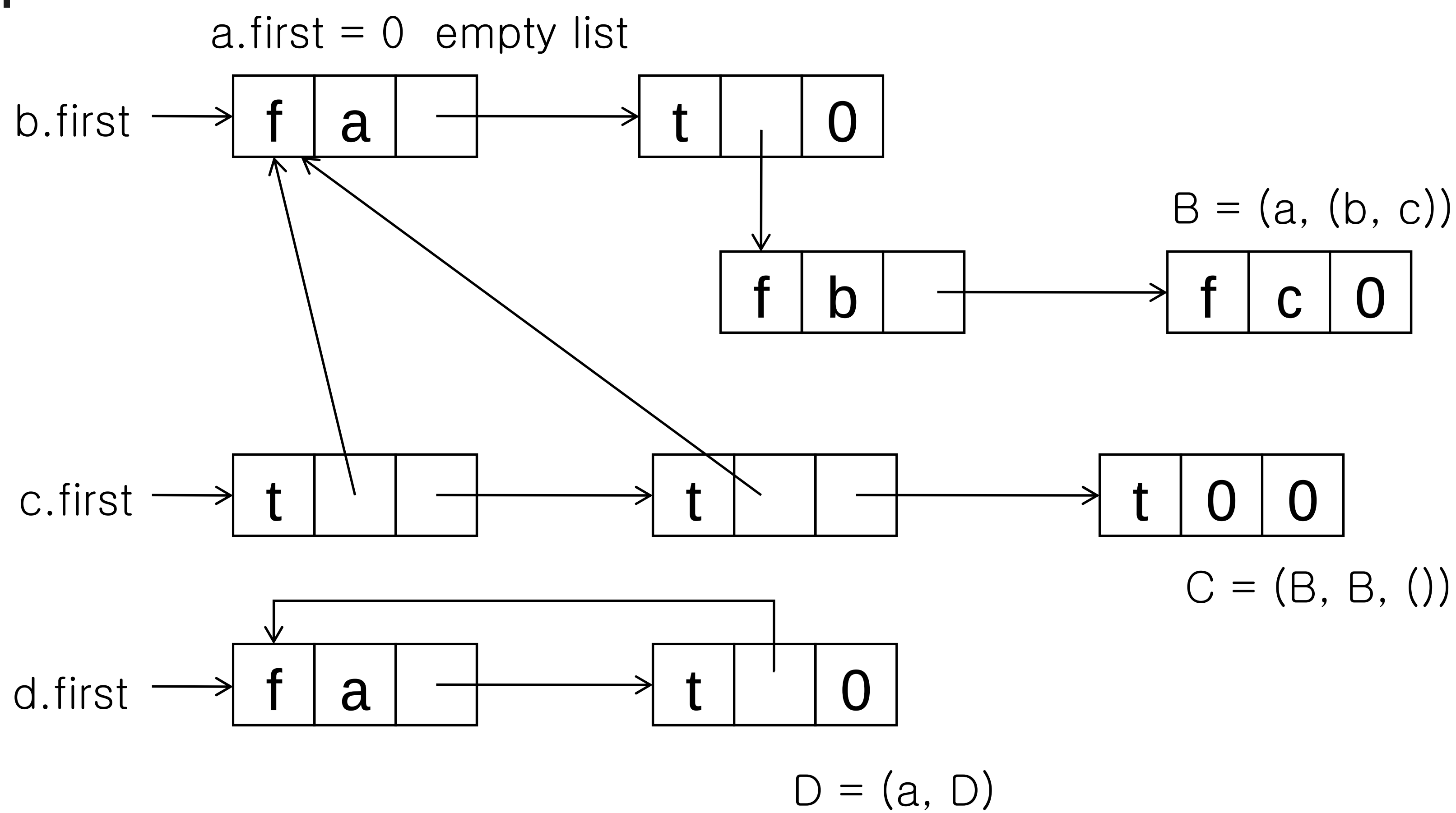


Figure 4.32 : Representation of list (1) to (4)

4.11.2 Recursive Algorithms for Lists

- Recursive algorithms typically consist of two components
  - Recursive function itself (workhorse)
  - The function invoke recursive function at the top level (driver)
  - The driver is declared as a public member while the workhorse is declared as a private member function

4.11.2 Recursive Algorithms for Lists

```
// Driver
Void GenList<T>::Copy(const GenList<T>&l) const
{ // Make a copy of l
        first = Copy(l.first);
}

// workhouse
GenListNode<T>* GenList<T>::Copy(GenListNode<T>* p)
{ // Copy the nonrecursive list with no shared sublists pointed at by p.
        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;
}
```

Program 4.35: Copying a list

4.11.2 Recursive Algorithms for Lists

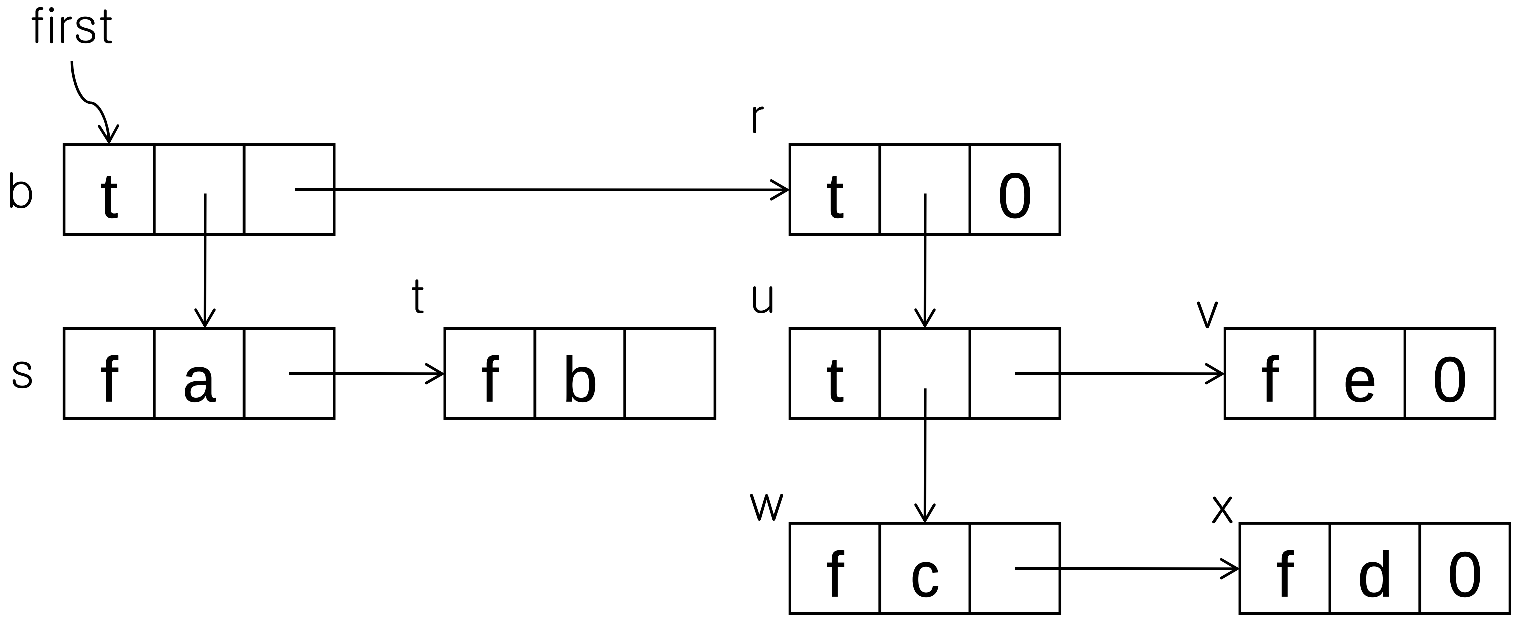


Figure 4.33 : Linked representation for A

4.11.2 Recursive Algorithms for Lists

level of recursion	vallue 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

Figure 4.34: Values of parameters in excution of GenList<T>::Copy(A)

4.11.3 List Equality

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

Program 4.36: Determining if two lists are identical

4.11.2.3 list Depth

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

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

Program 4.37: Computing the depth of a list

4.11.3 Reference Counts, Shared and Recursive Lists

- Lists are allowed to be shared by other lists
  - Saving storage used
  - But, necessary some changes in structure
    - Header nodes, changing pointers

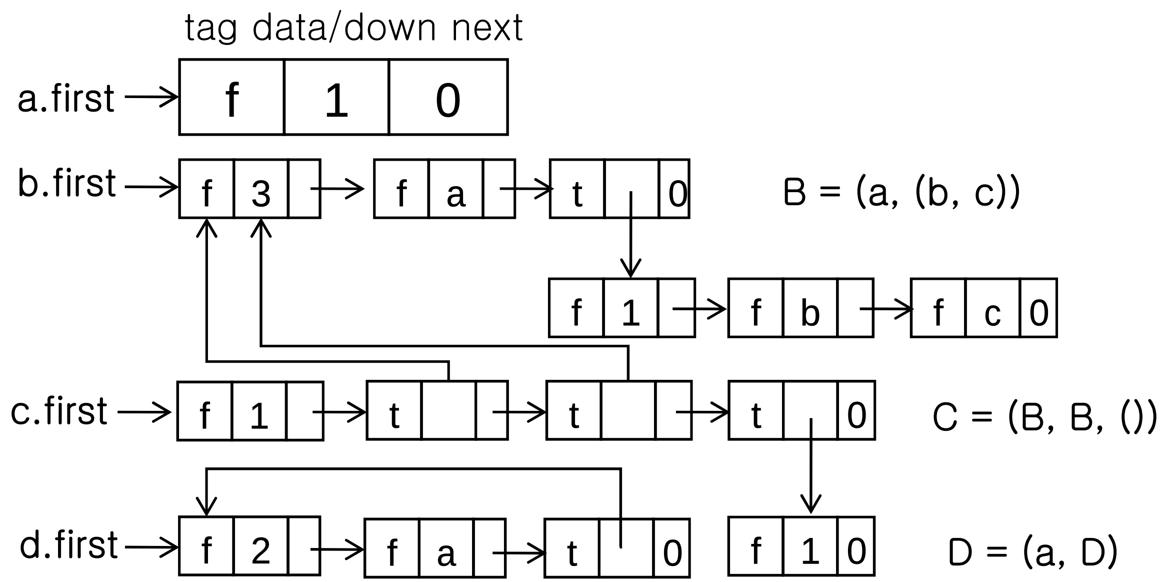


Figure 4.35 : Structure with header nodes for lists (1) to (4)

4.11.3 Reference Counts, Shared and Recursive Lists

- Need a mechanism to help determine whether or not the list nodes may be physically returned to the available-space list
  - So, the reference count is maintained in this field

  (1) a.first->ref=1 accessible only via a.first

  (2) b.first->ref=3 pointed to by b.first and two pointers from c

  (3) c.first->ref=1 accessible only via c.first

  (4) d.first->ref=2 accessible via d.first and one pointer from itself

4.11.3 Reference Counts, Shared and Recursive Lists

- Change the definition of GenListNode<T>

```
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;
    };
};
```

4.11.3 Reference Counts, Shared and Recursive Lists

```
// Driver
template<class T>
GenList<T>::~GenList()
{ // Each header node has a reference count
        if (first)
        {
                Delete(first);
                first = 0;
         }
}
// Workhorse
void GenList<T>::Delete(GenListNode<T> *x)
{
        x->ref--; // decrement reference count of header 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; // attach top-level nodes to av list
                av = x;
        }
}
```

Program 4.38: Deleting a list recursively

4.11.3 Reference Counts, Shared and Recursive Lists

- Sequence of c.~GenList<T>()

  (1) the reference count of c becomes zero

  (2) b.first->ref becomes 1 when the second top-level node of c is processed

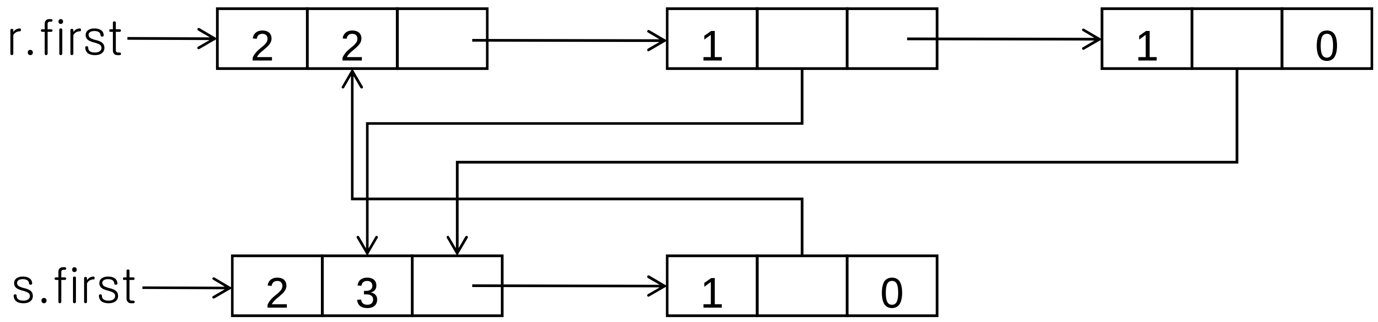
  (3) b.first->ref becomes 0 when the third top-level node of c is processed now, the five nodes of list B(a,(b,c)) are returned to the available-space list

  (4) the top-level nodes of c are linked into the available-space list

4.11.3 Reference Counts, Shared and Recursive Lists

- The reference count does not zero
  - d.~GenList<T>() results in d.first->ref becoming one but, no longer accessible
- The same is true in the case of indirect recursion (Figure 4.36)
  - After call r.~GenList<T>() : r.first->ref=1
  - After call s.~GenList<T>() : s.first->ref=2
  - But, they are no longer accessible
- Unfortunately, there is no simple way to supplement the list structure

4.11.3 Reference Counts, Shared and Recursive Lists



r = A(B,B) and s = B(A)

Figure 4.36: Indirect recursion of lists r=A and s=A