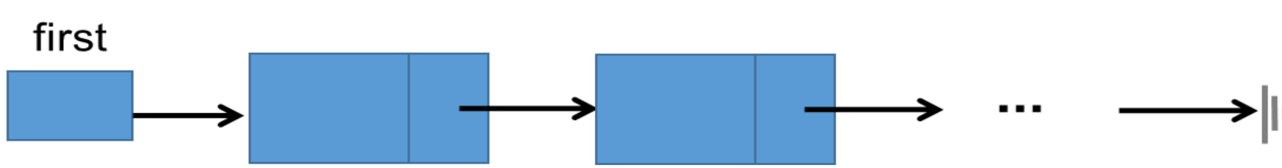
Linked Lists

Introduction

- Array
  - Successive items locate a fixed distance
- Disadvantage of storing an ordered list in array
  - Insertion and deletion of arbitrary elements are expensive.
    - Data movements during insertion and deletion
  - Storage allocation is not flexible.
    - Waste space in storing *n* ordered lists of varying size

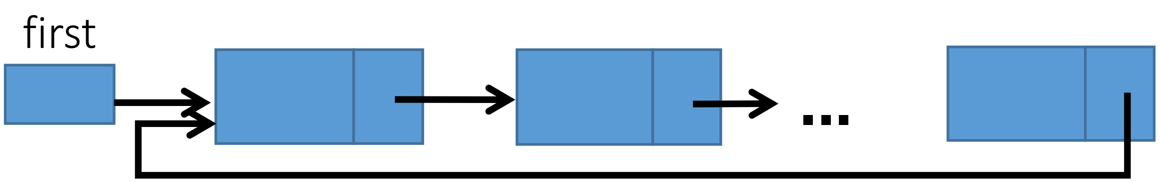
- Possible solution -- Linked list



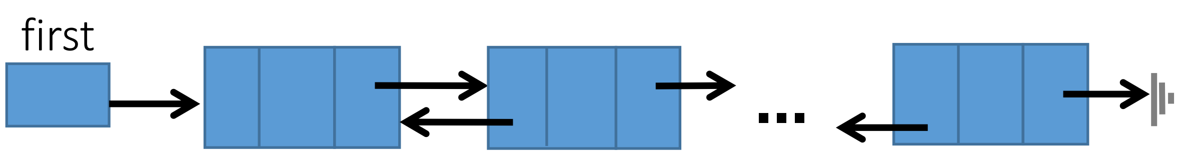
Linked list

- Possible Improvements
  - The elements in an ordered list don’t need to be stored in consecutive memory space.
    - The insertion and deletion of an element will not induce excessive data movement.
  - The element can be “dynamically” allocated.
- Types of linked list
  - Single linked list

  - Circular linked list

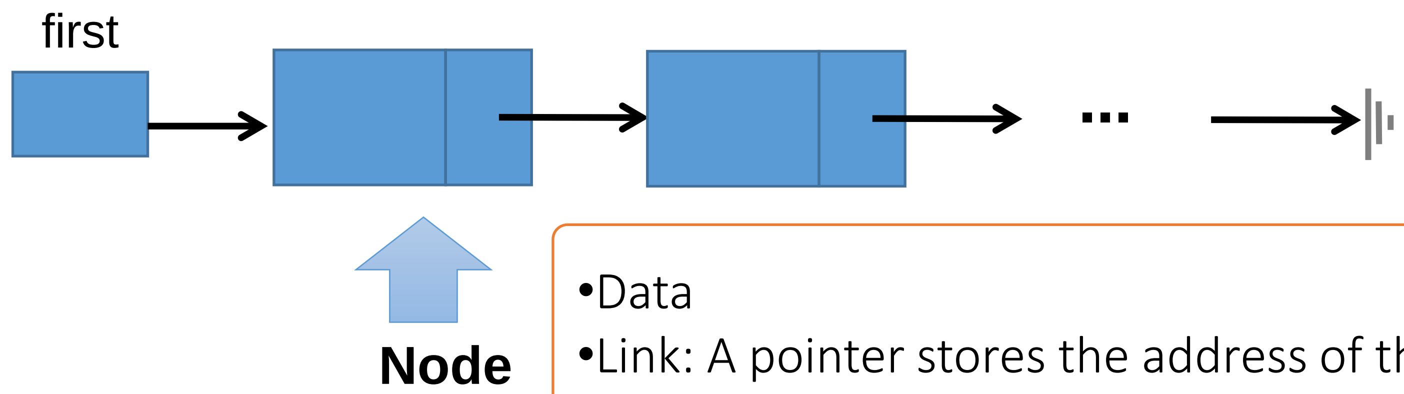


  - Doubly Linked List

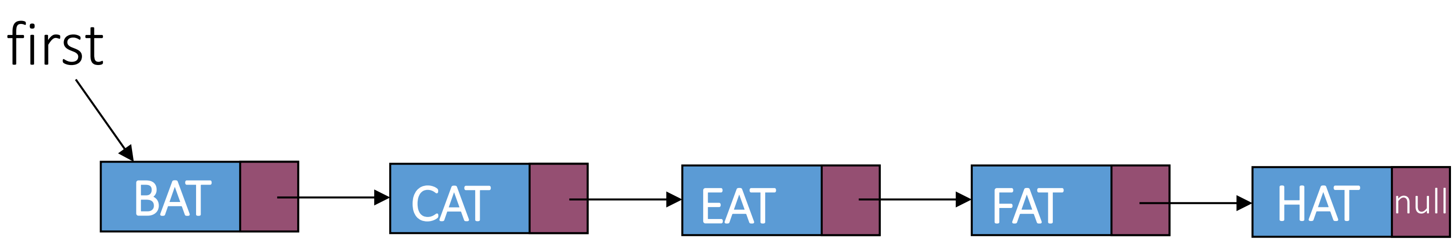


Singly Linked Lists

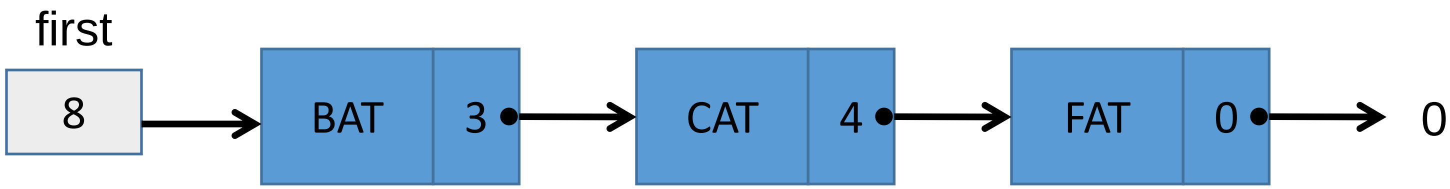
- Data structure for a linked list



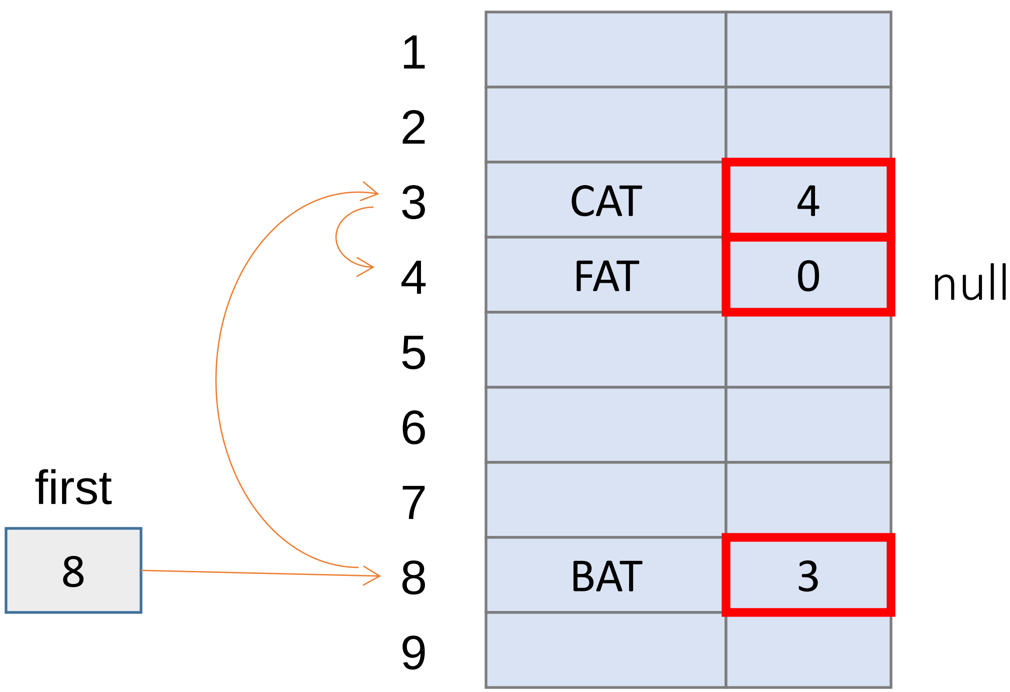
- Data
- Link: A pointer stores the address of the next node.



Example



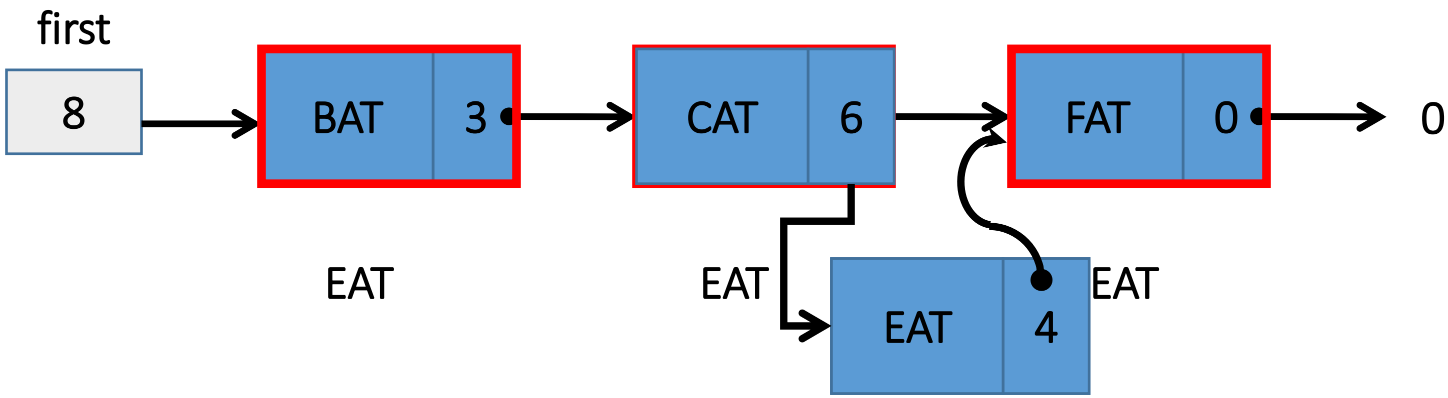
first
8
BAT
3
CAT
4
FAT
0
0

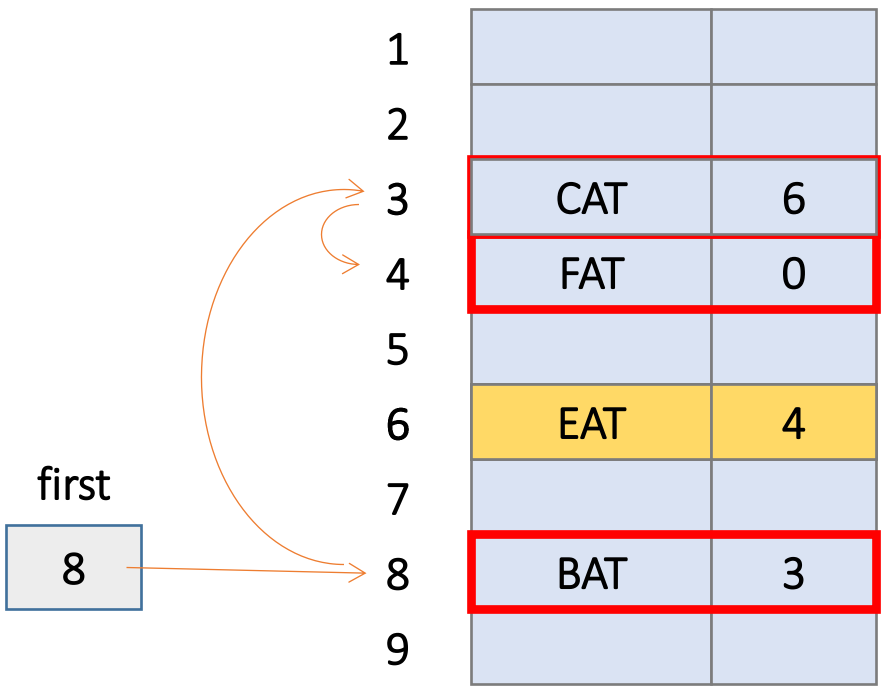


1
2
3
CAT
4
4
FAT
0
null
5
6
7
first
8
8
BAT
3
9

Insertion

- Insert EAT into an ordered linked list





Find the position where EAT is to be inserted.

1) Get a new node *a*

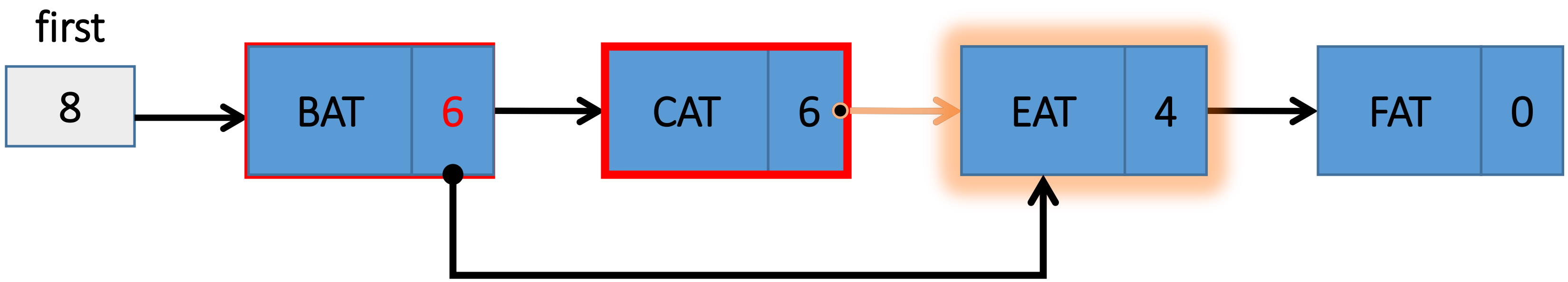
2) Set the *data* field of *a* to EAT.

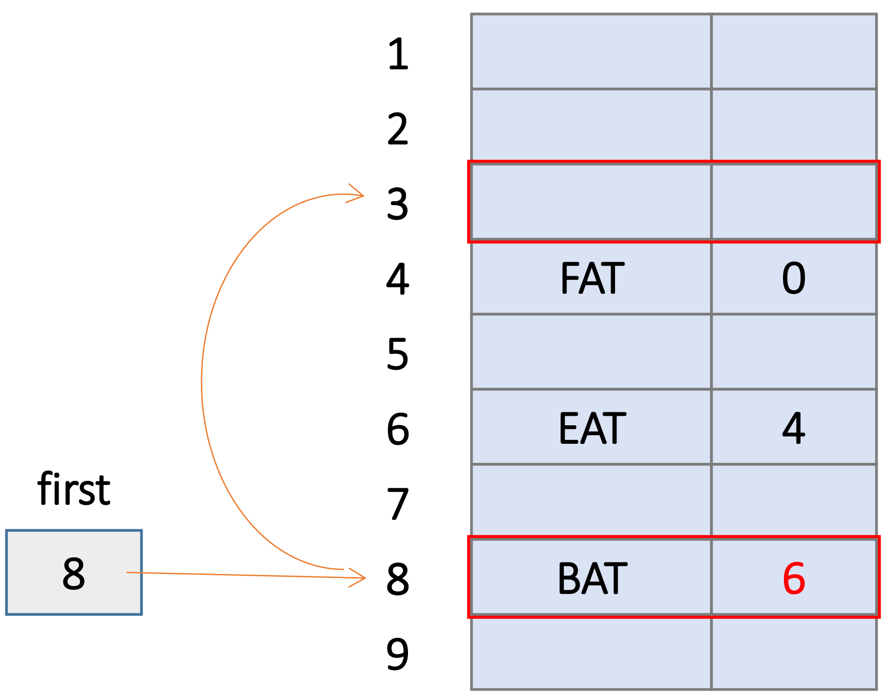
3) Set the *link* field of *a* to point the node after CAT, which contains FAT.

4) Set the *link* field of the node containing CAT to *a*.

Deletion

- Remove CAT from the linked list





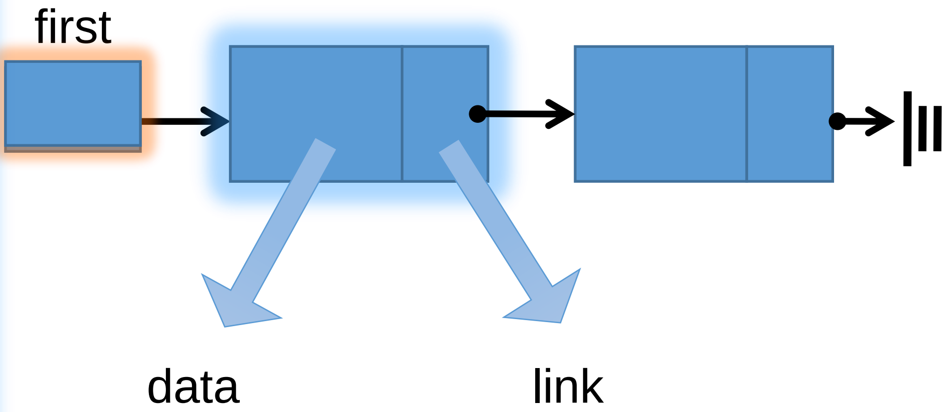
Find the address of CAT

1) Set the *link* of BAT to EAT.

2) Deallocate CAT

Representation of a Linked List

```
class ThreeLetterNode
{
    friend class LinkedList;
    public:
        ThreeLetterNode();
        ThreeLetterNode(DataField value);
        ~ThreeLetterNode();
    private:
        char data[3];
        ThreeLetterNode *link;
};
```

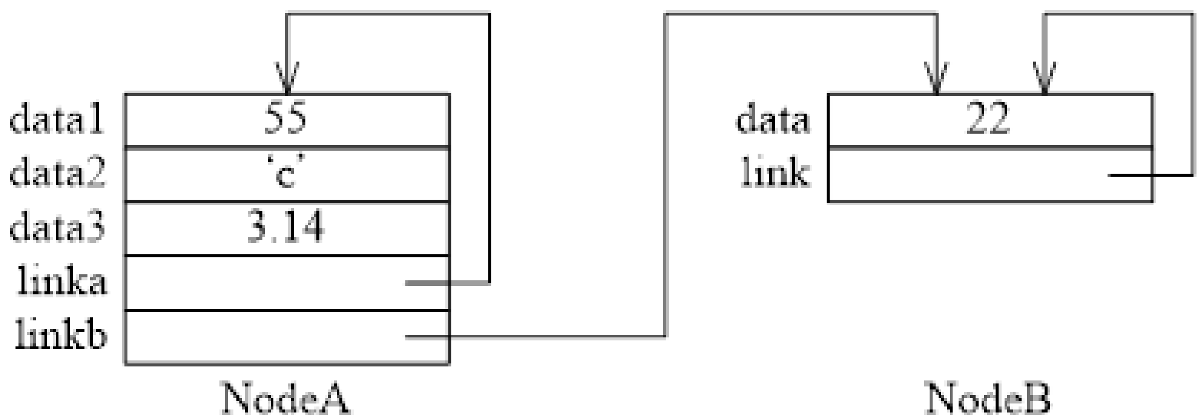



```
class LinkedList {
    private:
        ThreeLetterNode * first;
};
```

Example

- A complex list structure

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

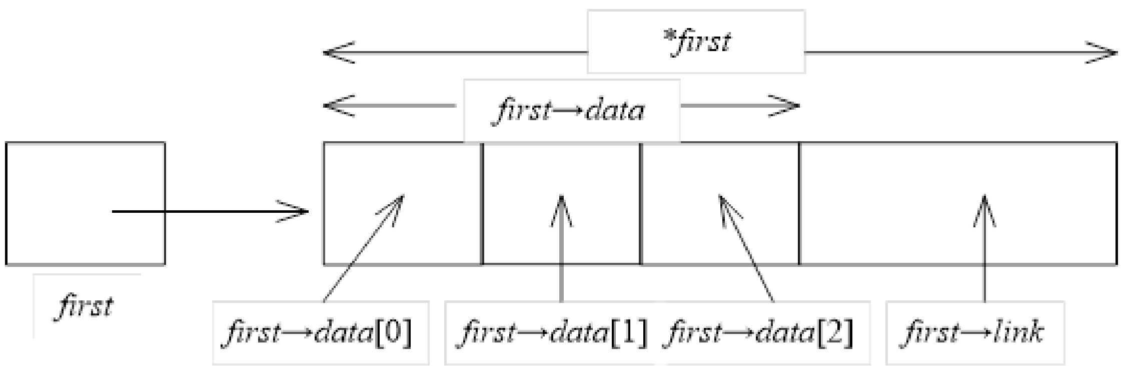


Designing a Chain Class in C++

- Design Attempt 1:
  - Use a **global variable** ***first*** which is a pointer of *ThreeLetterNode*.

    *ThreeLetterNode * first;*
  - To access data members of a node pointed to by ***first***

    first->*data* , *first*->*link*



- Flaw: Unable to access to **private** data members: data and link.

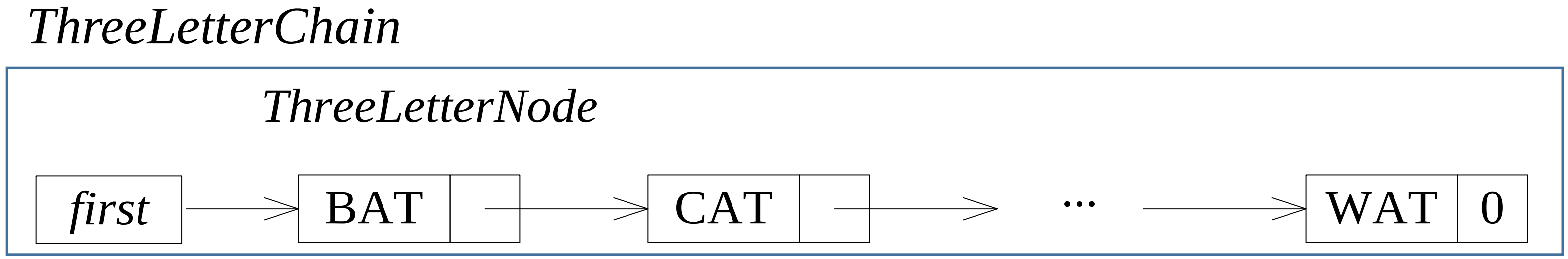
Data encapsulation principle

Designing a Chain Class in C++ (contd.)

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

Designing a List in C++ (contd.)

- Design Attempt 3:
  - Composition Class (HAS-A)
    - A class contains the items of another objects of another class.
    - Other than *ChainNode*, creating a class to represent the linked list. E.g., Use of two classes
      - *ThreeLetterNode* contains the data
      - *ThreeLetterChain* contains member functions that carry out list manipulation



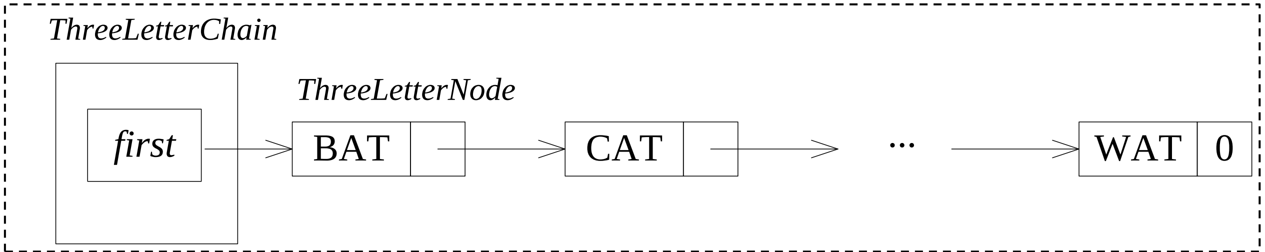
Composition Class (HAS-A)

- HAS-A
  - 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. E.g.,
    - *Computer* **HAS-A** *Processor*
    - *Book* **HAS-A** *Page*
- Composition Class
  - Friend Classes.
  - Nested Classes.

Friend Classes

can access *ThreeLetterNode*'s private and protected members

```
class ThreeLetterChain; // forward declaration

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

class ThreeLetterChain {
public:
    // Chain manipulation operations
…
private:
    ThreeLetterNode *first;
};
```



C++ Friend Function

- **friend** member function　　讓函數可存取 **private** 成員

```
class Box {
private:
   double width;
public:
   friend void printWidth( Box box );
   void setWidth( double wid );
};
// Member function definition
void Box::setWidth( double wid ) {
   width = wid;
}

//printWidth() is not a member function
of any class.
void printWidth( Box box ) {
   cout <<"Width of box: "<< box.width
<<endl;
}
```

```
int main( ) {
   Box box;

   box.setWidth(10.0);


   printWidth( box );

   return 0;
}
```

C++ Friend Class

- **friend** class　讓其他類別也可存取 **private** 成員

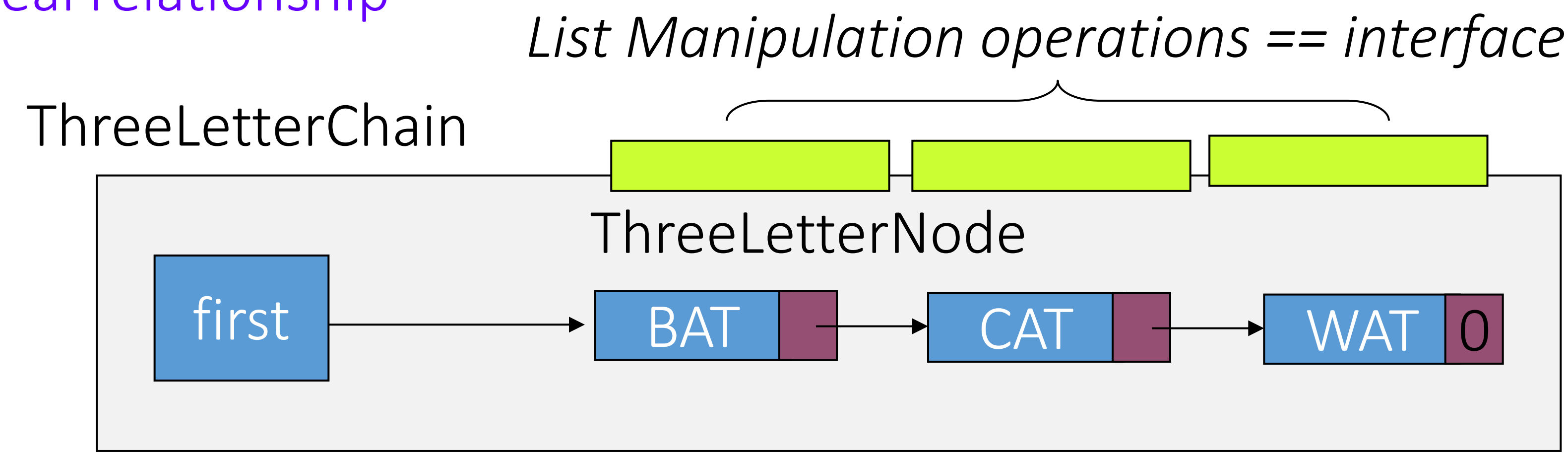
```
class Demo {
public:
    Demo(int pa, int pb) {
        a = pa;
        b = pb;
    }

    int do_something(Demo& d) {
        return d.a + d.b;
    }

    friend class Demo2;

private:
    int a;
    int b;
};
```

```
class Demo2 {
public:
    int do_something2(Demo& d) {
        return d.a + d.b;
    }
};
```

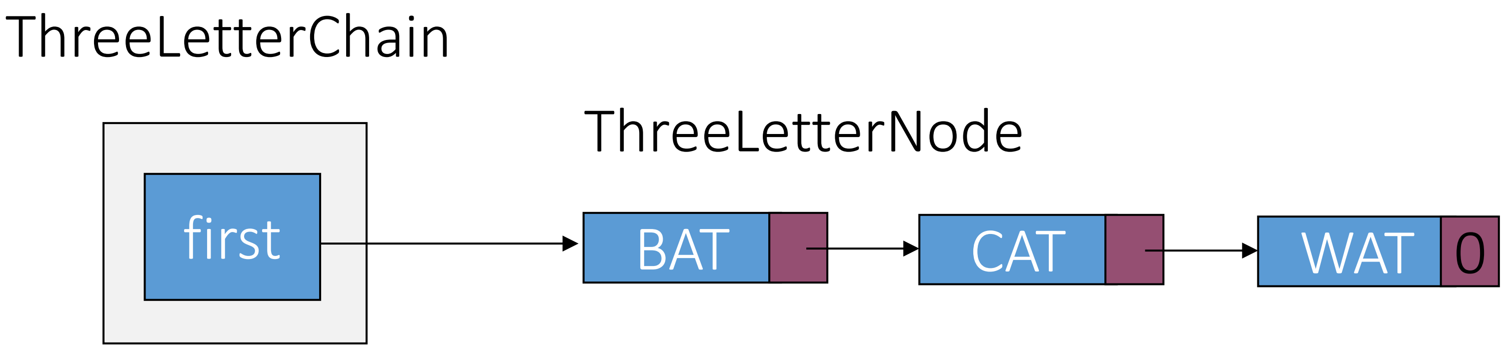
```
int main(void) {
    Demo d(32, 22);
    std::cout << d.do_something(d) << std::endl;
    Demo2 d2;
    std::cout << d2.do_something2(d) << std::endl;
    return 0;
}
```

Composite Classes (contd.)

- Ideal relationship



- Actual relationship



Nested Classes

- One class is defined inside the definition of another.
- For example, class *ThreeLetterNode* is defined inside the private portion of the definition of class *ThreeLetterChain*
  - This ensures that *ThreeLetterNode* objects cannot be accessed outside class *ThreeLetterChain*

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

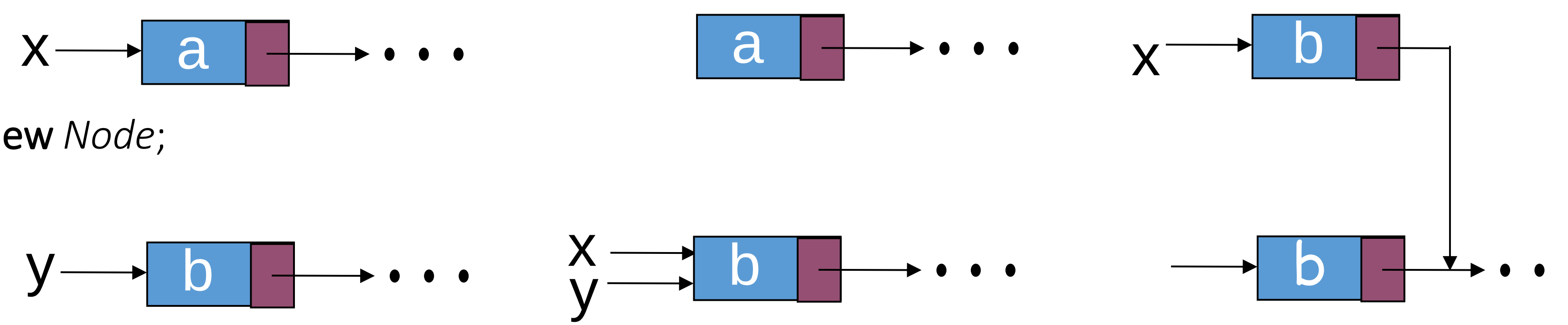
Pointer Manipulation in C++

- *ThreeLetterNode *f* ;
  - *f is of type ThreeLetterNode ** ;
  - *f* = new *ThreeLetterNode* ;

f is a pointer, storing the address of a Node

* *f* denotes the Node

- Two pointer variables of the same type can be compared.
  - *x* == *y*, *x* != *y*, *x* == 0
- Effect of pointer assignments
  - x = y, *x = *y



x = **new** *Node*;

y = **new** *Node*;

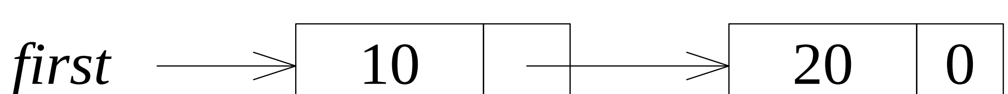
x = y

*x = *y

Chain Manipulation Operations

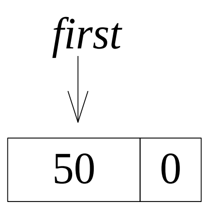
```
class ChainNode {

friend class Chain;
public:
        ChainNode (int element =0, ChainNode *next=0)
        // Chain manipulation operations
        {data = element; link = next;}
private: ;
      int data;
      ChainNode *link;
};
```

```
void Chain::Create2()
{
      // create a linked list with two nodes
      ChainNode* second = new ChainNode(20,0);

      // create and set fields of first node
      first = new ChainNode(10,second);
}
```

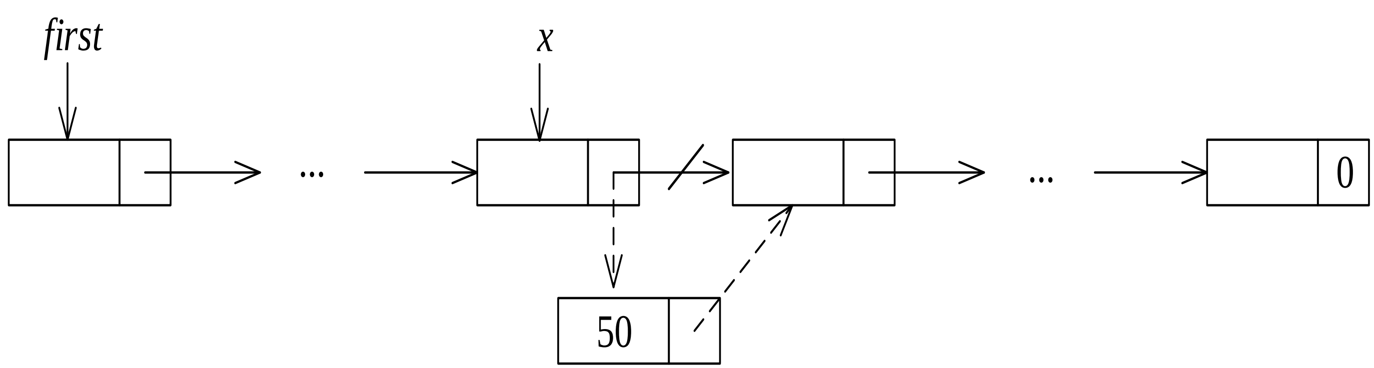


List Insertion

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



Insert into an empty list

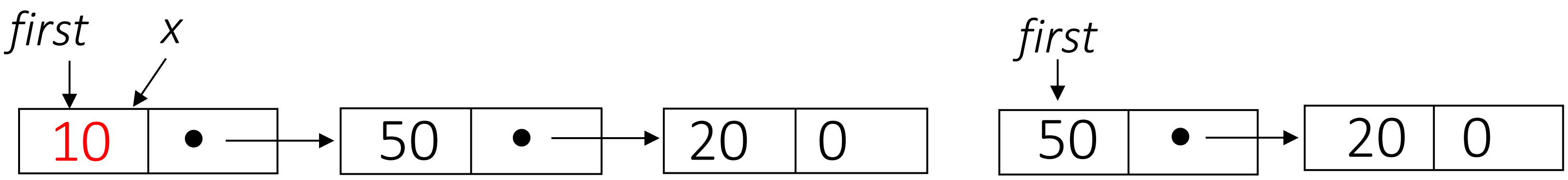


Insert into an nonempty list

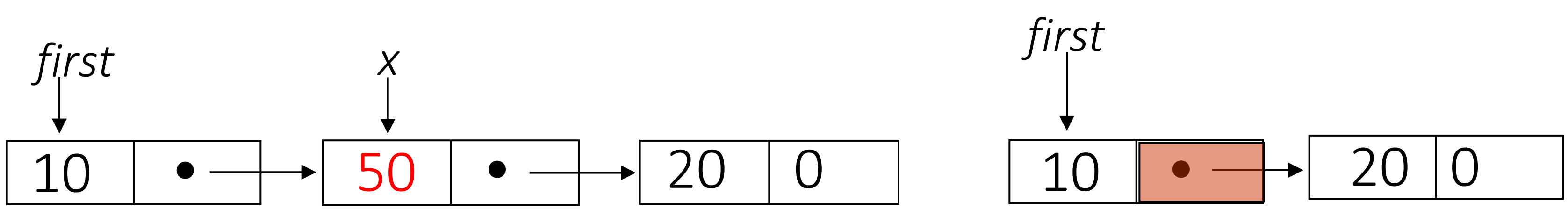
List Deletion

```
void Chain::Delete(ChainNode* x, ChainNode *y)
{
    if(x = = first) first = first→link;
    else y→link = x→link;
    delete x;
}
```

- Delete the first node



- Delete node other than the first node.



Software Reusability

- Software reuse is a technique to reduce the cost without scarifying the quality
- At the initial design and develop stage, we should try to make it possible to reuse the software in the future
- To make the *Chain* class more reusable
  - To handle different data types, we can implement chains with *Template*
  - To access the elements of a container class one by one, an *Iterator* class is needed
    - Allow users to reuse the class to develop other functions

Template Definition of Chains

- *Chain* <*T*> is declared as a **friend** of *ChainNode* <*T*>.
  - *ChainNode* <*int*> can be accessed by members of *Chain* <*int*>
- *first* is declared to be a pointer to an object of type *ChainNode* <*T*>
  - An object of type *Chain* <*int*> only consists of nodes of type *ChainNode* <*int*>
- An empty chain of integers *intlist*

  Chain <*int*> *intlist*;
- An empty chain of float *floatlist*

  Chain <*float*> *floatlist*;

Template Definition of Chains

```
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;
}
```

Iterator

- A chain iterator is an object that is used to traverse all the elements of a container class ***C***.
  - Output all integers in ***C***
  - Obtain the max, min, mean, median, or mode of all integers in ***C***
  - Obtain the sum, product, or sum of squares of all integers in ***C***
  - Obtain all integers in ***C*** that satisfy some property *P*
    - *P* could be a positive integer, a square of an integer, ….
  - Obtain the integer *x* from ***C*** such that, for some function *f*, function *f*(*x*) *is maximum*

Iterator

- The above problems require to examine all elements of the container class (Chain)

```
Initialization steps;
for each item in C
{
    currentItem = current item of C
    do something with currentItem

postprocessing step;
```

For example,

```
//Pseudo-code for Computing Max Element
int x = std::numeric_limits<int>::min( );
// initialization, must include <limits>
for each item in C
{
    currentItem = current item of C
    x = max(currentItem,x);              // do something
}
return x;                                 // postprocessing step
```

Iterator

- Implementation 1

  for (**int** *i*= 0**;** *i* < n**;** *i* + +)
  currentItem=*a*[*i*]**;**

- Implementation 2

  for (*ChainNode*<*T*> **ptr* = *first***;** *ptr* !=0**;** *ptr*= *ptr*->*link*)
  currentItem=*ptr*->*data***;**

- Design considerations
  - *Chain* <*T*> is a template class, all of its operations should be independent of the type
  - Member functions of *Chain* <*T*> can become quite large and make it unwieldy
  - Adding member functions should entail learning how the contained class is implemented

=> Using *Iterator*, which is a pointer to an element of an object

Iterator

```
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;
}
```

```
for (int i= 0; i != 3; i++)
    Cout<<x[i] <<" ";
```

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

Possible code for STL ***copy***

```
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++; }
}
```

ChainIterator

```
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;
        }
    // 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;
}
```

Chain Operations

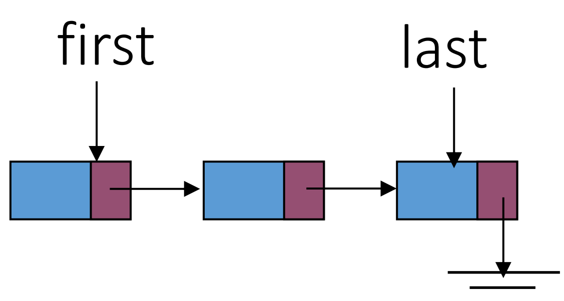
- The most important of designing a reusable class is choosing which operations to include
  - Goal: to provide enough operations to improve the reusability for various applications

    Not too many=>bulky
- Operations includes in most reusable classes
  - Constructor
  - Destructor
  - Copy
  - Assignment =
  - TFE ==
  - Input <<
  - Output >>
- Chain
  - Insert (insertFront)
  - InsertBack
  - Concatenate
  - Reverse
  - Delete

Inserting at the Back of a List

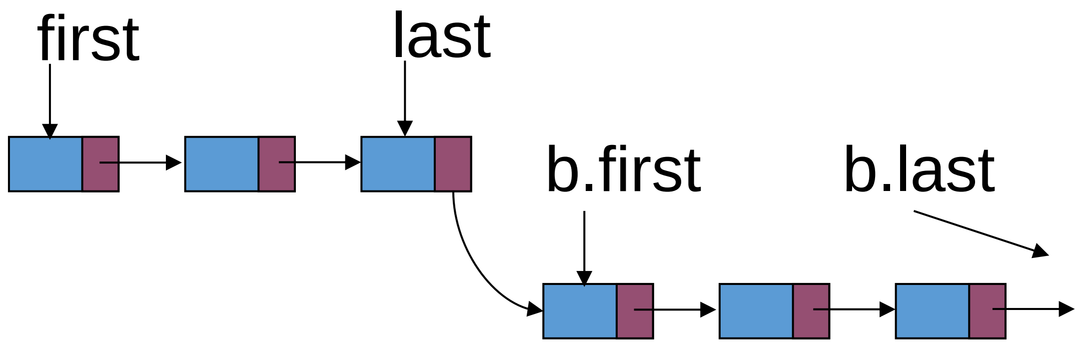
```
template < class T >
void Chain<T>::InsertBakc( const T& e)
{
    if (first) { // nonempty chain
        last→link = new ChainNode<T>(e);
        last = last→link;
    }
    else first = last = new ChainNode<T>(e);
}
```

A last pointer pointing to the last node is added for better efficiency



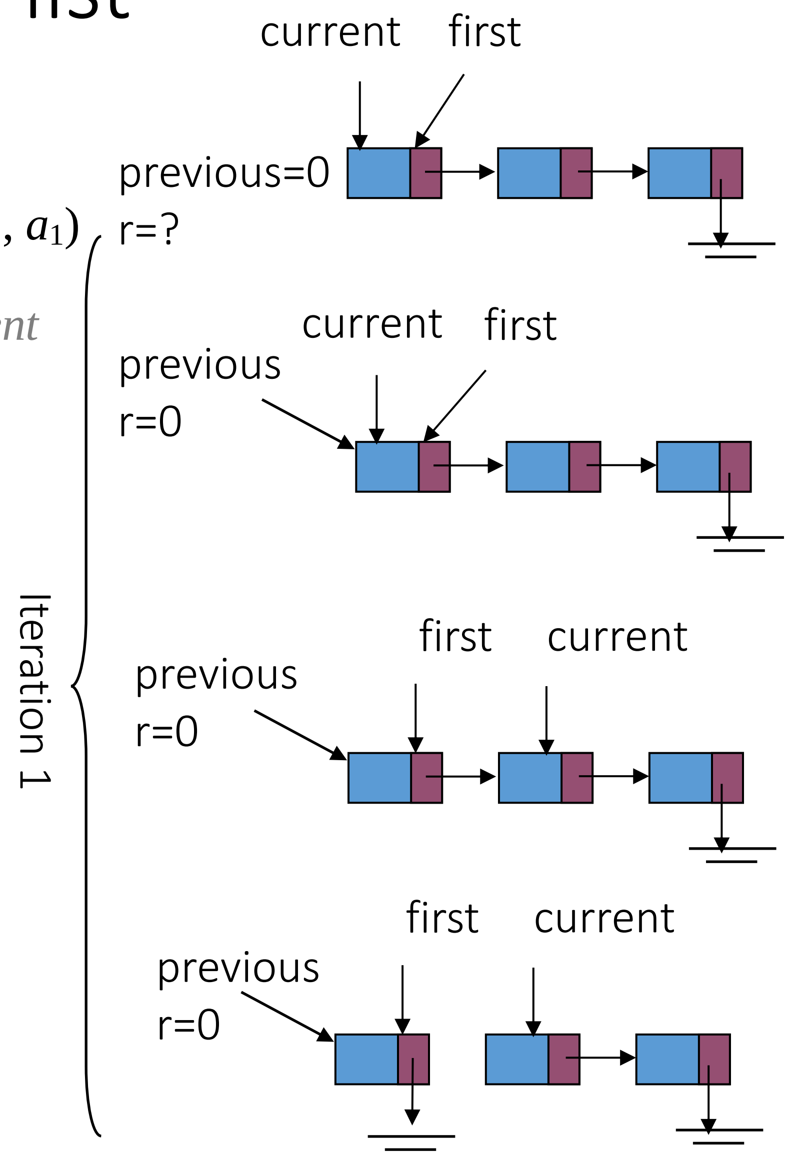
Concatenating two chains

```
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;
}
```



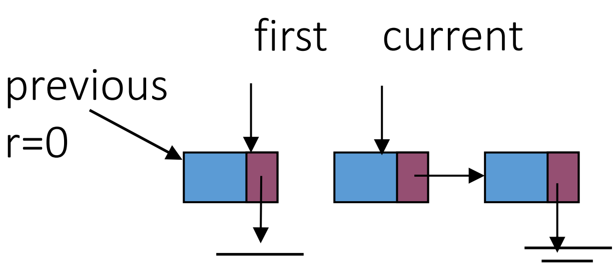
Reversing a list

```
template <class T>
void Chain<T>::Reverse()
{// a chain is reversed so that   (a1, …, an) becomes (an, …, a1)
    ChainNode<T> *current = first,
                 *previous = 0; // previous trails current
    while (current) {
        ChainNode<T> *r = previous;
        previous = current;
          // r trails previous
        current = current → link;
          // current move to next node
        previous → link = r;
          // link previous to preceding node
    }
    first = previous;
}
```

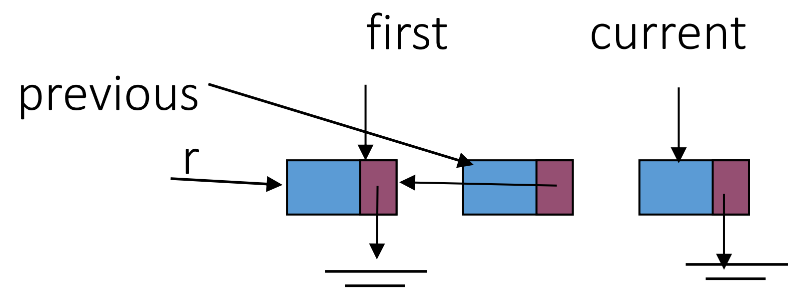


```
template <class T>
void Chain<T>::Reverse()
{// a chain is reversed so that   (a1, …, an) becomes (an, …, a1)
      ChainNode<T> *current = first,
                          *previous = 0; // previous trails current
      while (current) {
            ChainNode<T> *r = previous;
            previous = current;
              // r trails previous
            current = current → link;
              // current move to next node
            previous → link = r;
              // link previous to preceding node
      }
      first = previous;
}
```

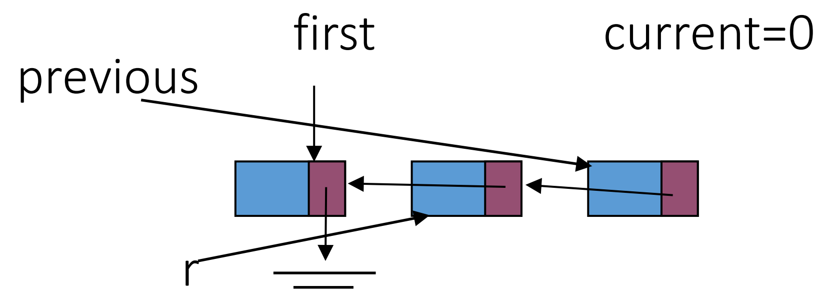
Iteration 1

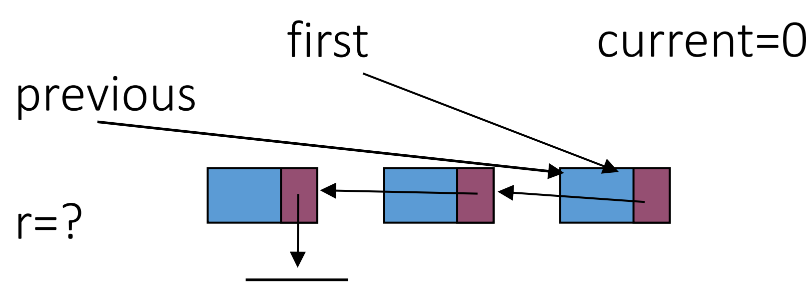


Iteration 2



Iteration 3

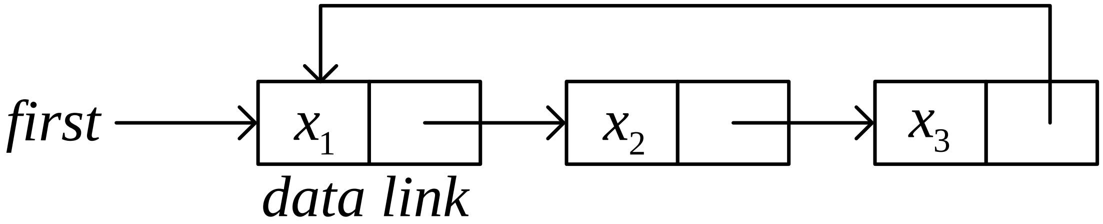




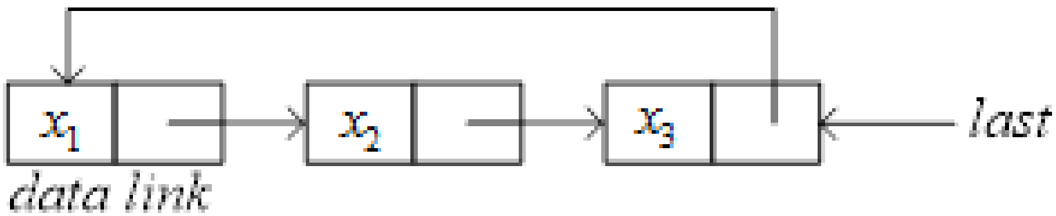
When Not To Reuse A Class

- "**not reuse but rather design a new one**" scenarios
  - Reusing a class to design another may results in a less efficient class
    - E.g., queue and stack in Section 4.6
  - Operations that are complex and specialized, not likely to be implemented in a reusable class
    - E.g., Finding equivalence classes

Circular Lists



Check Last Node: *current->link == first;*

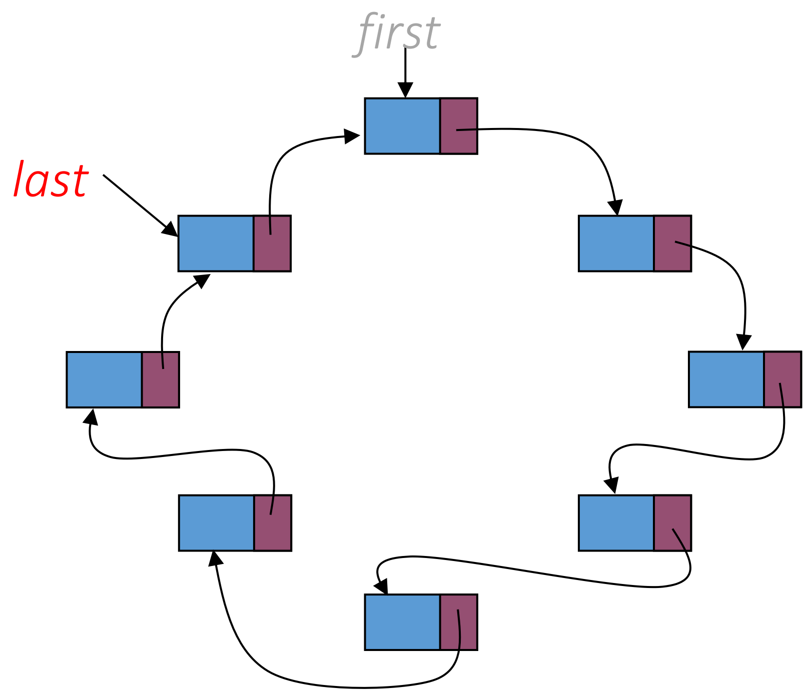


Use the last node instead of the first node

More convenient o inset a new node at the front or at the back of the list O(1)

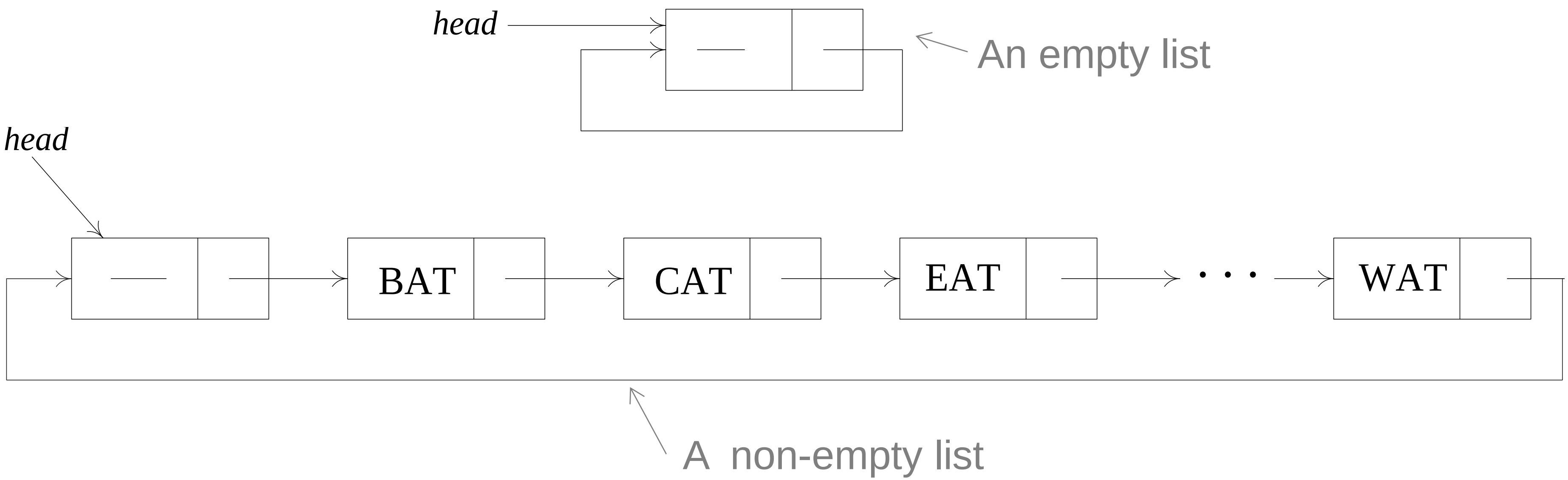
Circular Lists - *InsertFront*

```
template <class T>
void CircularList <T>::InsertFront(const T& e)
{// Insert the element e at the “front” of a 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;
    }
}
```



It is more convenient if the access pointer of a circular list points to the last node rather than to the first node

Using **Dummy Node** to Represent Empty list




```
template <class T>
void CircularList <T>::InsertFront(const T& e)
{ // insert e into the front of the circular list
   // last point to the last node of the list
      ChainNode <T> *newNode = new ChainNode <T>(e);
      newNode → link = last → link;
      last → link = newNode;
}
```

Maintain an Available Space List

- The memory a program uses is typically divided into a few different areas, called **segments**:
  1. The **code** segment (also called a text segment), where the compiled program sits in memory. The code segment is typically read-only.
  2. The **bss** segment (also called the uninitialized data segment), where zero-initialized global and static variables are stored.
  3. The **data** segment (also called the initialized data segment), where initialized global and static variables are stored.
  4. **The heap, where dynamically allocated variables are allocated from.**
  5. The call **stack,** where function parameters, local variables, and other function-related information are stored.

Maintain an Available Space List

- To avoid memory leak, delete a node directly in the destructor

Return it to heap

```
template <class T>
Chain<T>::~Chain(){
//Free all nodes in the chain
      ChainNode<T>* next;
      for (; first; first = next) {
            next = first->link;
            delete first;
      }
}
```

Time Complexity O(n)

Maintain an Available Space List

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

Maintain an Available Space List

```
template <class T>
ChainNode <T>* CircularList <T>::GetNode()
{// 提供一個節點供使用
    ChainNode<T>* x;
    if (av) {x = av; av = av→link;}
    else x = new ChainNode<T>;
    return x;
}
```

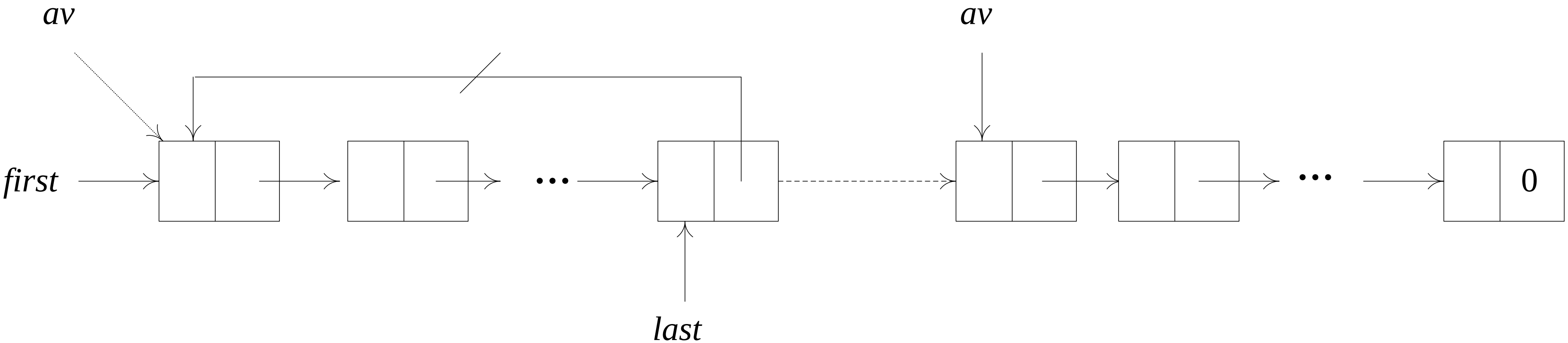
```
template <class T>
void CircularList<T>::RetNode(ChainNode<T>*& x)
{// 釋回 x 所指向的節點
    x→ link = av;
    av = x;
    x = 0;
}
```

Maintain an Available Space List

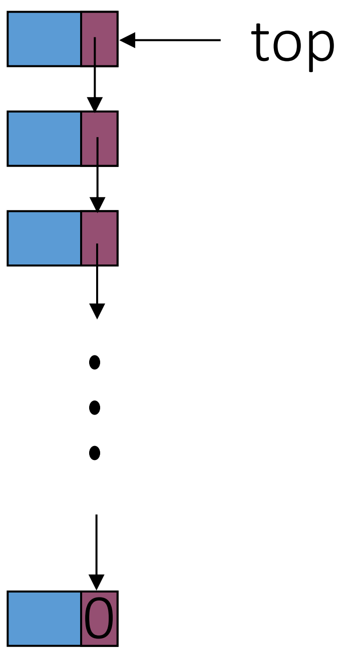
- Example-Circular List Destructor

```
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;      // first node of list becomes front of av list
        last = 0;
    }
}
```

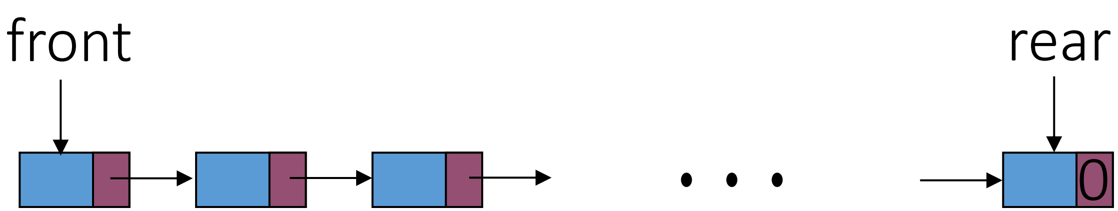
Place the list in front of the available list



Linked Stacks and Queues

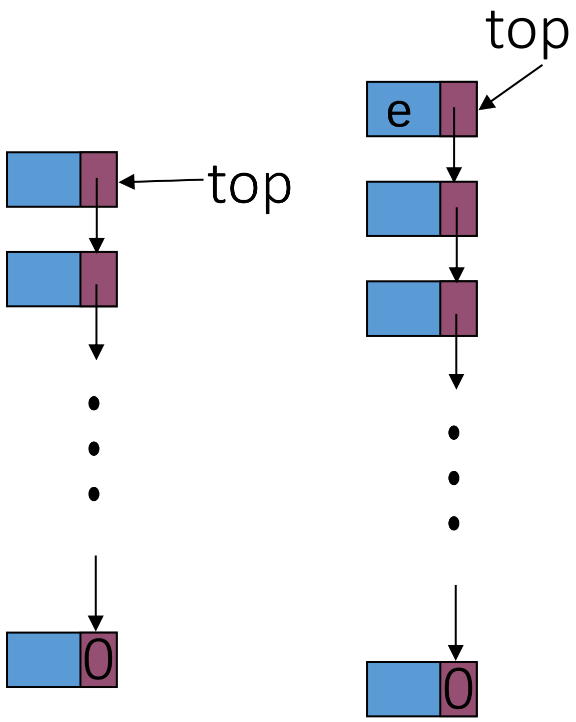


Linked Stack



Linked Queue

Linked Stacks



- Adding to a linked stack

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

- Deleting from a linked stack

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

Linked Queues

- Adding to a linked queue

```
template <class T>
void LinkedQueue <T>:: Push(const T& e)
{
    if (IsEmpty( )) front = rear = new ChainNode(e,0); //empty queue
    else rear = rear→link = new ChainNode(e,0); // attach node and update
rear

}
```

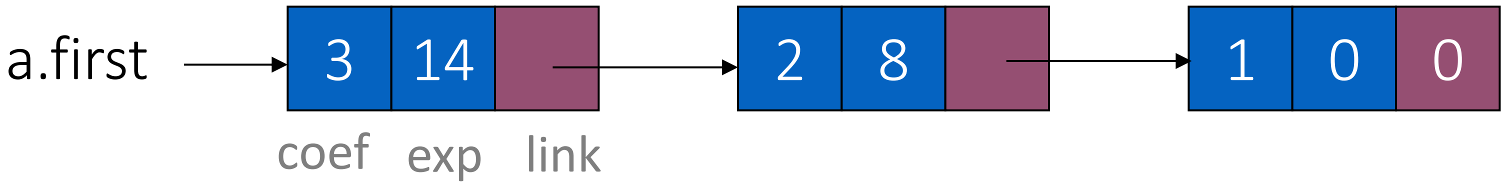
- Deleting from a linked queue

```
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;     // remove the first node from Chain
    delete delNode;         // free the node
}
```

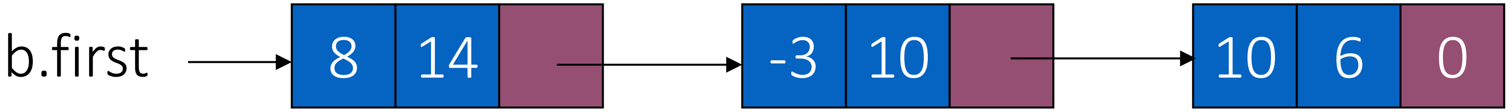
Polynomials

- Representing each term of a polynomial as a node containing coefficient and exponent fields, as well as a pointer to the next term.

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



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



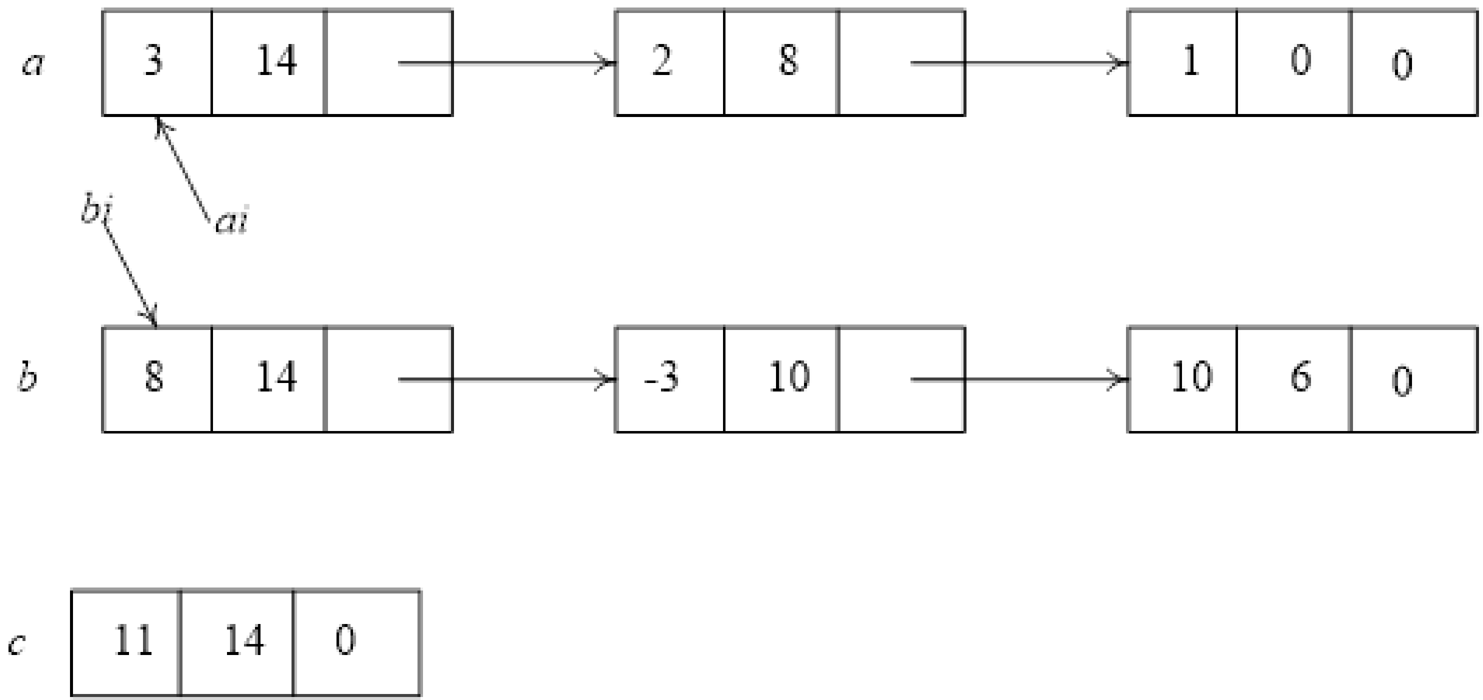
Polynomial Representation

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

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

Operating on Polynomials

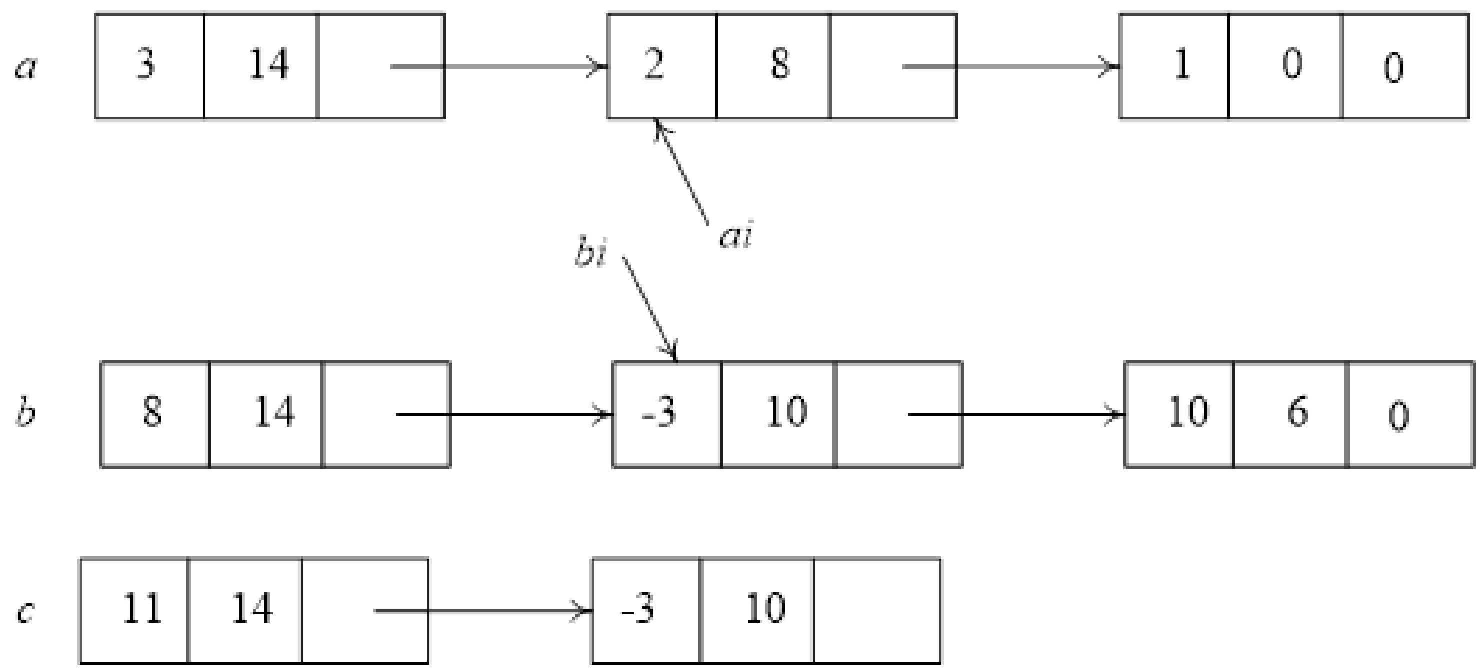
- With linked lists, it is much easier to perform operations on polynomials such as adding and deleting.
  - E.g., adding two polynomials *a* and *b*
- To add two polynomials, we examine their terms starting at the nodes pointed to by *a* and *b*.
  - **If the exponents of the two terms are equal**
    1. add the two coefficients
    2. create a new term for the result.
  - **If the exponent of the current term in *ai* is less than *bi***
    1. create a duplicate term of *b*
    2. attach *b* to the result, i.e., *Polynomial* object *c*
    3. advance the pointer to the next term in *b*.
  - **Similarly, if *ai->exp* > *bi->exp***

Adding Polynomials



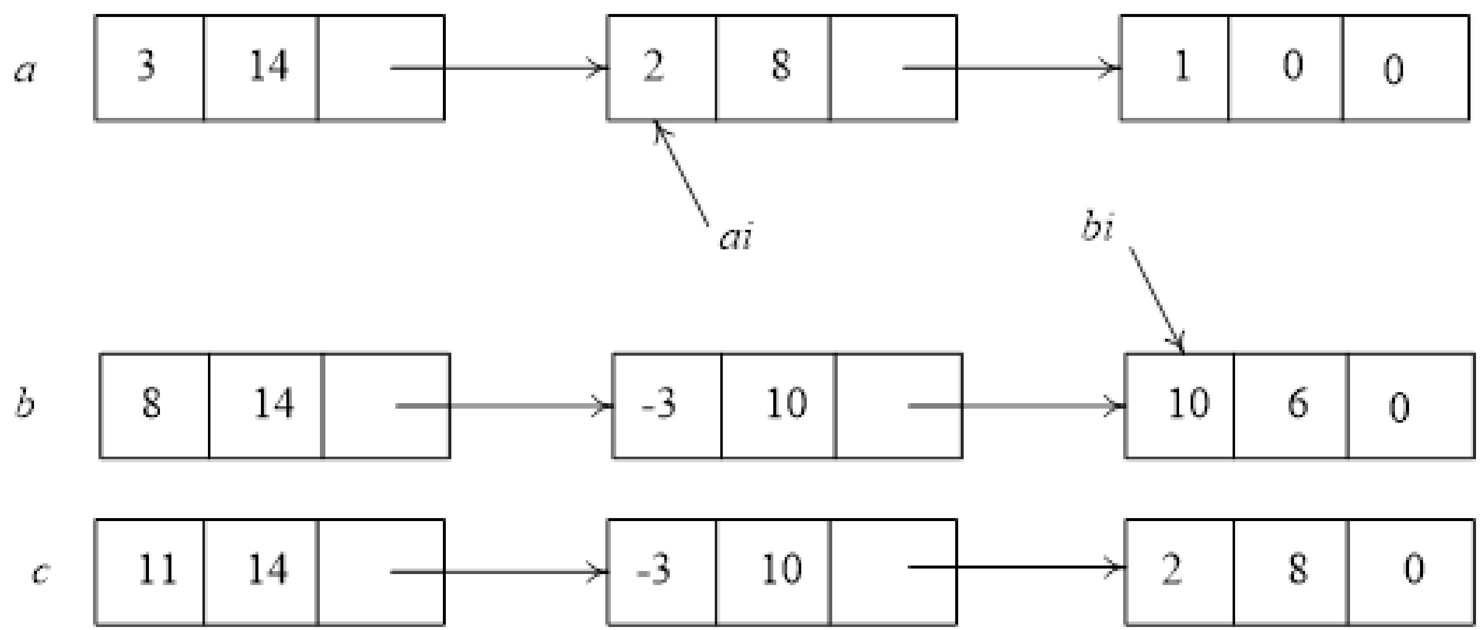
(i) $ai \rightarrow exp == bi \rightarrow exp$

Adding Polynomials



(ii) $ai \rightarrow exp < bi \rightarrow exp$

Adding Polynomials



(iii) $ai \rightarrow exp > bi \rightarrow exp$

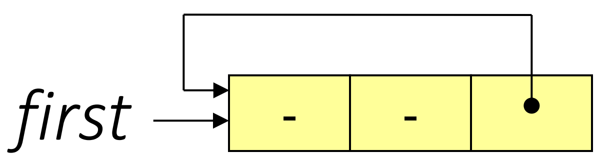
```
Polynomial Polynomial::operator+(const Polynomial& b) const
{// Polynomias *this（a）and b are added and the sum returned
 Term temp;
 Chain <Term>::ChainIterator ai = poly.begin(),
                                             bi =b. poly.begin();
 Polynomial c;
 while (ai&&bi) { // current node are not null
        if (ai→exp = = bi→exp) {
               int sum = ai→coef + bi→coef;
               if (sum) c.poly.InsetBack (temp.Set(sum, ai→exp));
               ai++; bi++; //  前 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
        }
 }
```

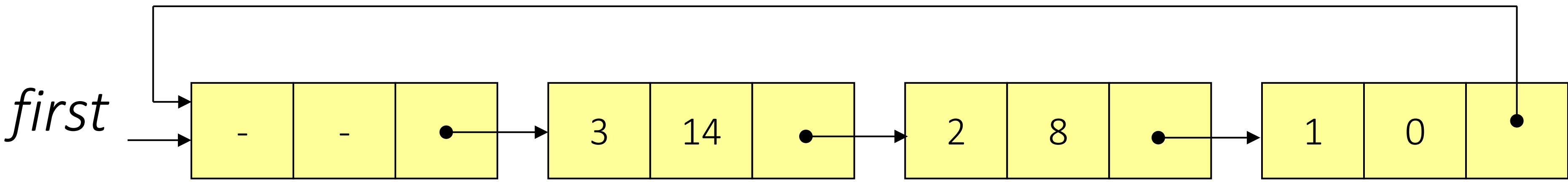
```
    while (ai) { // copy rest of a
        c.poly.InsertBack(temp.Set(ai→coef,ai→exp)) ;
        ai++;
    }
  while (bi) { // copy rest of b
        c.poly.InsertBack (temp.Set(bi→coef,bi→exp)) ;
         bi++ ;
     }
    return c;
}
```

Polynomials

- Using dummy head to avoid special case

Zero polynomial





$3x^{14} + 2x^{8} + 1$ with dummy head to avoid special case

Equivalence Relations

- Properties of Relations: For an arbitrary relation by the symbol ≡
  - **Reflexive**
    - If $x \equiv x$
  - **Symmetric**
    - If $x \equiv y$, then $y \equiv x$
  - **Transitive**
    - If $x \equiv y$ and $y \equiv z$, then $x \equiv z$
- 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*.

Examples

- The "equal to" ($=$) relationship is an equivalence relation since
  - $x = x$
  - $x = y$ implies $y = x$
  - $x = y$ and $y = z$ implies $x = z$
- An equivalence relation is to partition the set S into equivalence classes such that two members x and y of S are in the same equivalence iff $x \equiv y$
- Example, given 12 polygons numbered $0 \sim 11$

  $0 \equiv 4, 3 \equiv 1, 6 \equiv 10, 8 \equiv 9, 7 \equiv 4, 6 \equiv 8, 3 \equiv 5, 2 \equiv 11, 11 \equiv 0$

  Three equivalent classes $\{0,2,4,7,11\}, \{1,3,5\}, \{6,8,9,10\}$

Finding Equivalence Classes

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

Phase 1

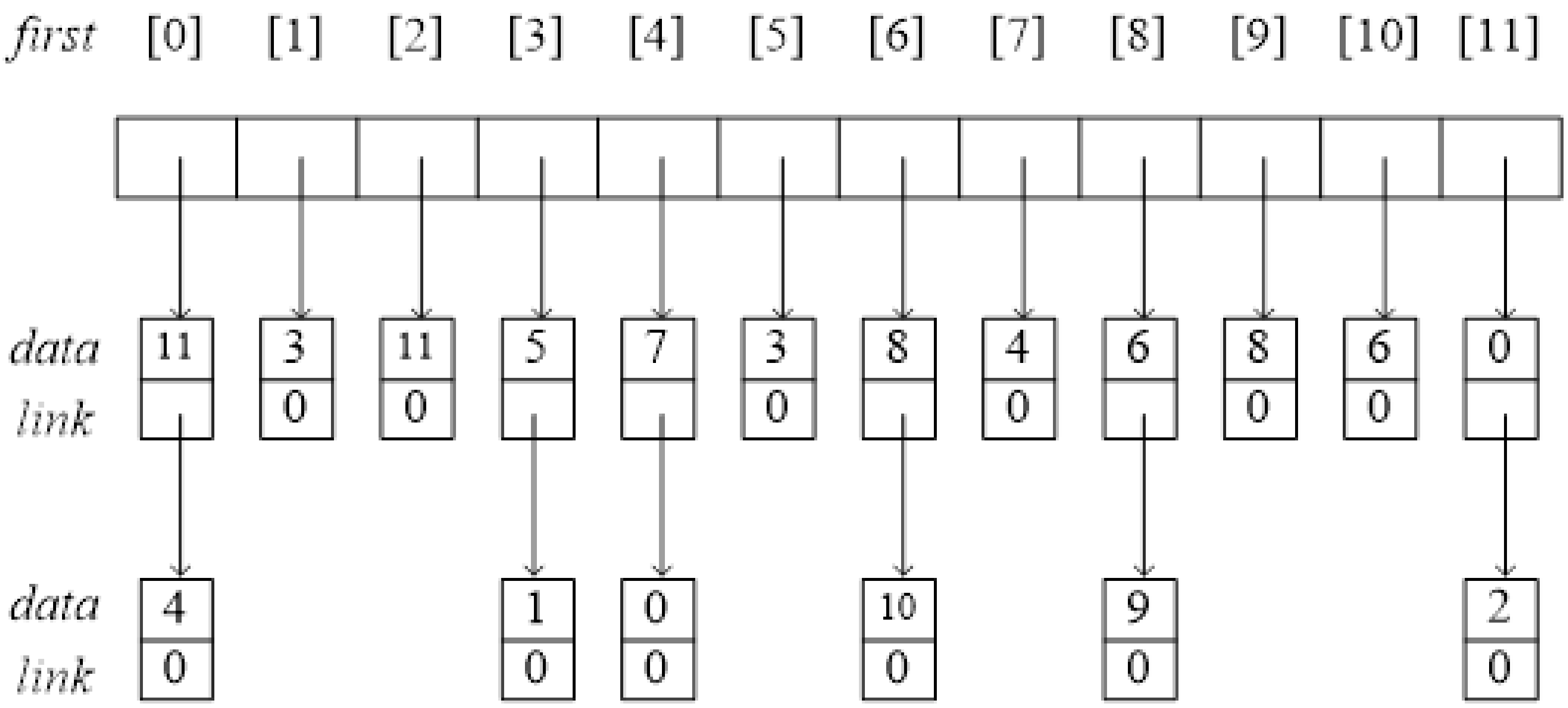
Phase 2

What kinds of data structures are adopted?

Example

Lists after pairs have been input

$0 \equiv 4$
$3 \equiv 1$
$6 \equiv 10$
$8 \equiv 9$
$7 \equiv 4$
$6 \equiv 8$
$3 \equiv 5$
$2 \equiv 11$
$11 \equiv 0$



Finding Equivalence Classes

- Two phases to determine equivalence
  - **First phase:**
    - The equivalence pairs (i, j) are read in and stored.
  - **Second phase:**
    - Begin at 0 and find all pairs of the form $(0, j)$.
    - Continue until the entire equivalence class containing 0 has been found, marked, and printed.
    - Next find another object not yet output, and repeat the above process.

Finding Equivalence Classes

1. 用linked list的array記住每個數跟其他那些數”相等”
2. 印出來的時候記得那些數已經印過了(不要重複)
3. 利用stack的概念

Finding Equivalence Classes

void *Equivalence*()
{
read *n*； // read in number of objects
initialize *first*[0:*n*-1] to 0，*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*]； direct equivalence
}
for (*i* = 0**;** *i* < *n***;** *i*++)
if (!*out*[*i*]) **{**
out[*i*] = **true**；
output the equivalence class that contains object *i*；
}
compute indirect equivalence
using transitivity
}

```
class ENode {
friend void Equivalence( );
public:
      ENode(int d = 0)    // constructor
            {data = d; link = 0;}
private:
      int data;
      ENode *link;
};

void Equivalence( )
{// input equivalence pairs and outpu the equivalene classes
      ifstream inFile( "equiv.in", ios::in); // "equiv.in" is the input file
      if (!inFile) throw "Cannot open input file.";
      int i, j, n;
      inFile >> n;    // read no. of objects
      // initialize first and out
      ENode **first = new ENode* [n];
      bool *out = new bool[n];
      // use STLfunction 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 EOF
                  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 e output
        cout << endl << "A new class: " << i;
        out[i] = true;
        ENode *x = first[i]; ENode *top = 0; // initialize stack
        while (1) {    // scan 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; // delete
        } // while(1) ends
    } // if (!out[i]) ends
```

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

Summary of Equivalence Algorithm

- Two phases to determine equivalence class
  - **Phase 1**: Equivalence pairs (i, j) are read in and adjacency (linked) list of each object is built.
  - **Phase 2**: Trace (output) the equivalence class containing object i with stack (depth-first search). Next find another object not yet output, and repeat.
- Time complexity: $\Theta(m+n)$
  - n: # of objects
  - m: # of pairs (relations)

Sparse Matrices

- Inadequate of sequential schemes
  - # of nonzero terms will vary after some matrix computation
  - Matrix just represents intermediate results
- New scheme
  - Each column (row): a circular linked list with a head node

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

Example: Sparse Matrix

- Circular linked list representation of a sparse matrix has two types of nodes:
  - **head node**: head, down, right, and next
  - **entry node**: head, down, row, col, right, value
- Head node i is the head node for both row i and column i.
  - Each head node is belonged to three lists: **a row list**, **a column list**, and a **head node list**.
- For an $n \times m$ sparse matrix with r nonzero terms, the number of nodes needed is $\max\{n, m\} + r + 1$.

Node Structure for Sparse Matrix

<table>
<tr><td colspan="2">next</td></tr>
<tr><td>down</td><td>right</td></tr>
</table>

Header node

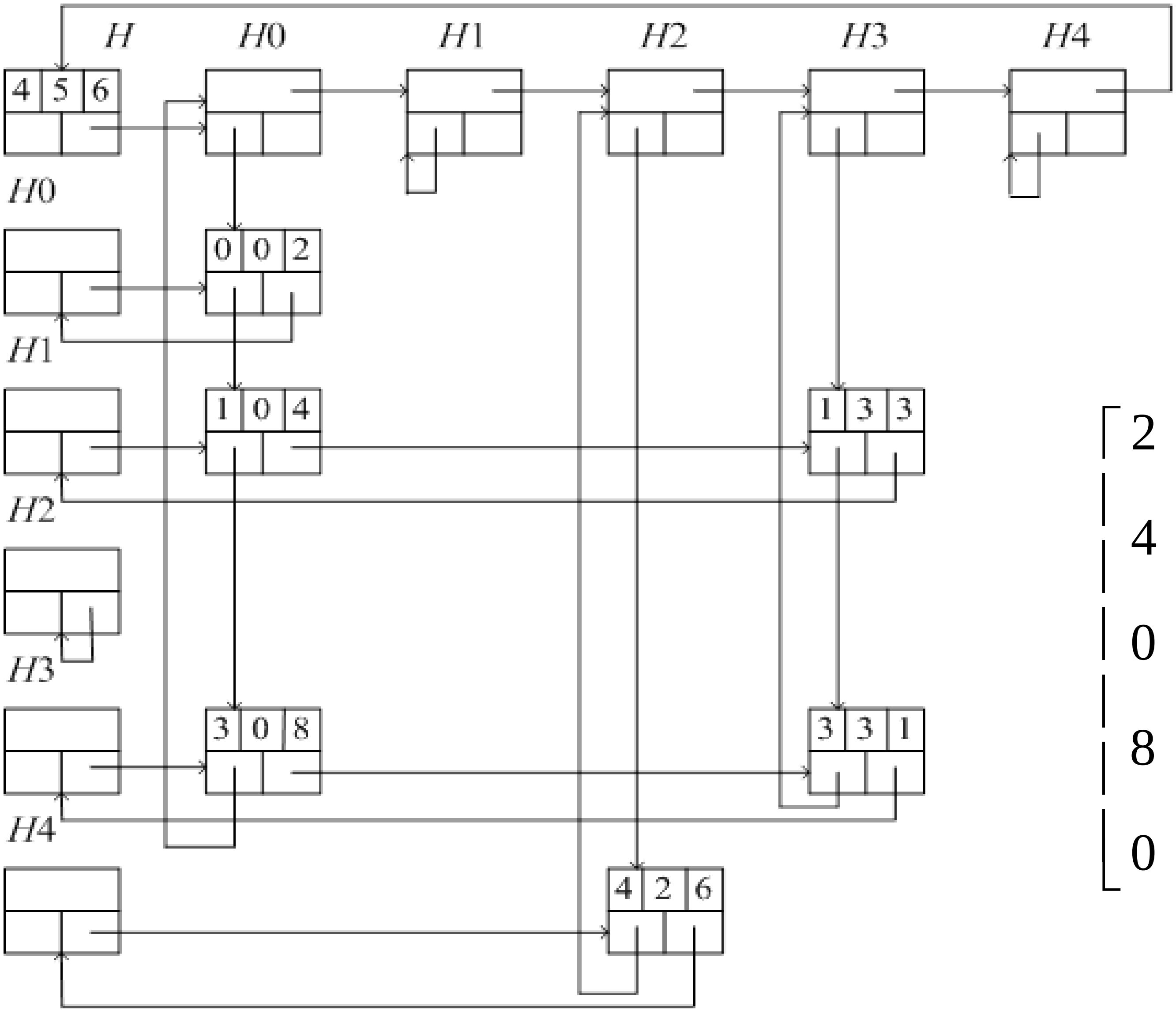
<table>
<tr><td colspan="2">row</td><td colspan="2">col</td><td colspan="2">value</td></tr>
<tr><td colspan="3">down</td><td colspan="3">right</td></tr>
</table>

Element node

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

A 5×4 sparse matrix

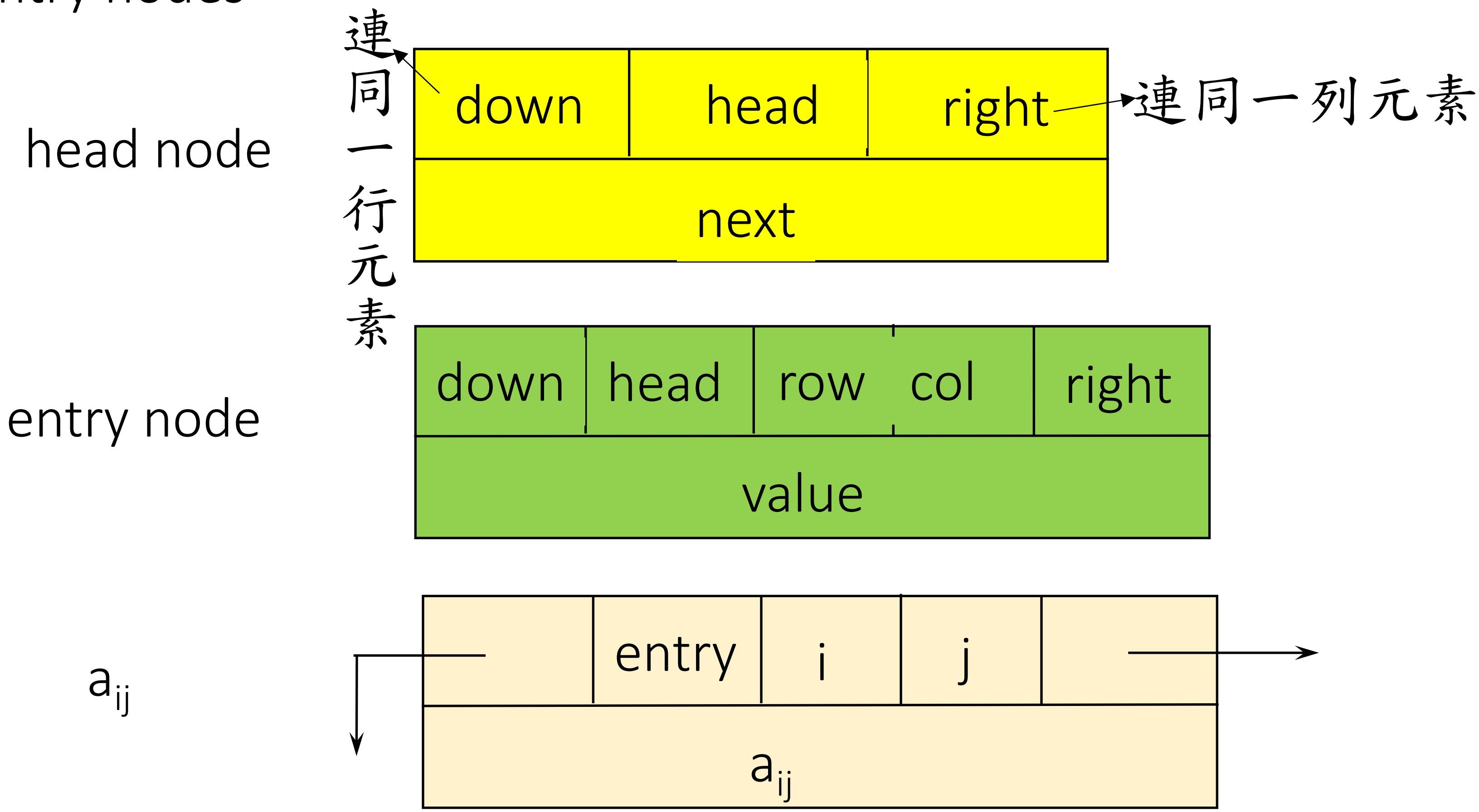
Linked List for Sparse Matrix



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

Revisit Sparse Matrices

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

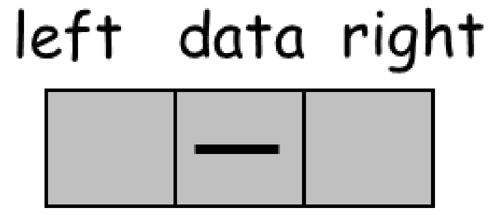


```
struct Triple{int row, col, value;};
class Matrix;    // forward declaration

class MatrixNode {
friend class Matrix;
friend istream& operator>>(istream&, Matrix&); // for reading in a matrix
private:
        MatrixNode *down , *right;
        bool head;
        union { // anonymous unionunion
                MatrixNode *next;
                Triple triple;
        };
        MatrixNode(bool, Triple*); // constructor
}

MatrixNode::MatrixNode(bool b, Triple *t) { // constructor
        head = b;
        if (b) {right = down = this;} // row/column header node
        else triple = *t; // element node or header node for list of header nodes
}
class Matrix{
friend istream& operator>>(istream&, Matrix&);
public:
        ~Matrix(); // destructor
private:
        MatrixNode *headnode;
};
```

Doubly Linked List

- Move in forward and backward direction.
  - Singly linked list (in one direction only)
- How to get the preceding node during deletion or insertion?
  - Using 2 pointers
- Node in doubly linked list
  - left link field (*llink*)
  - data field *(item*)
  - right link field (*rlink*)

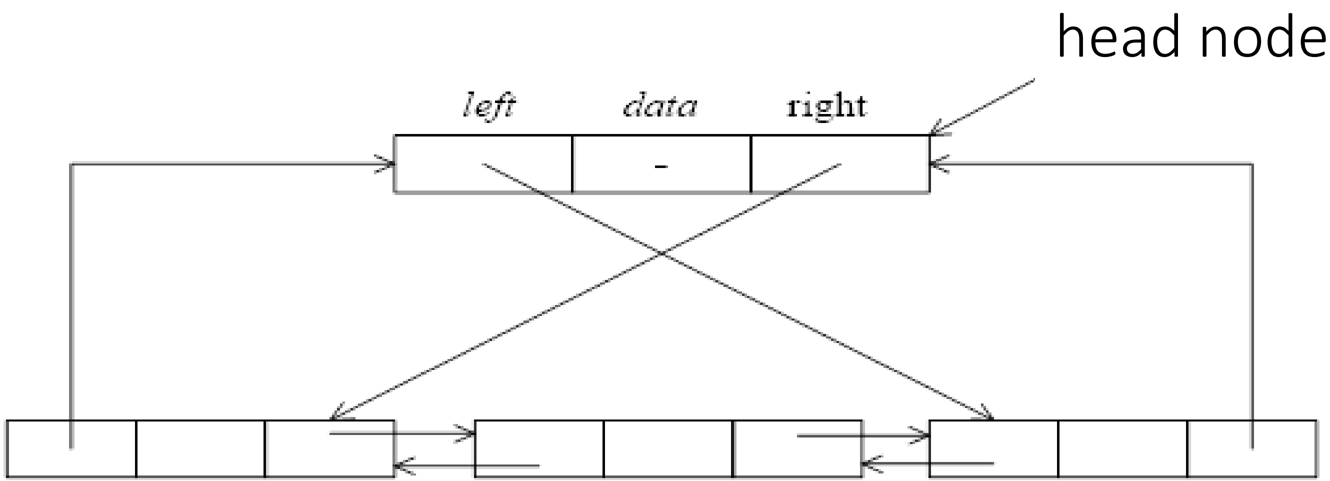
Doubly Linked List

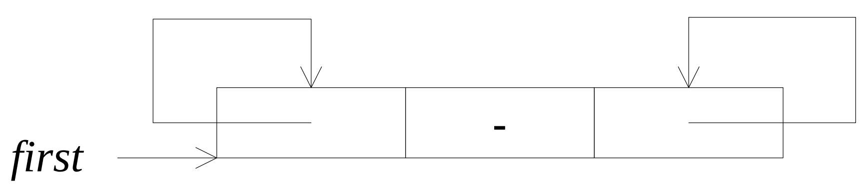



```
class DblList;

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

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

Doubly Linked List (contd.)

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

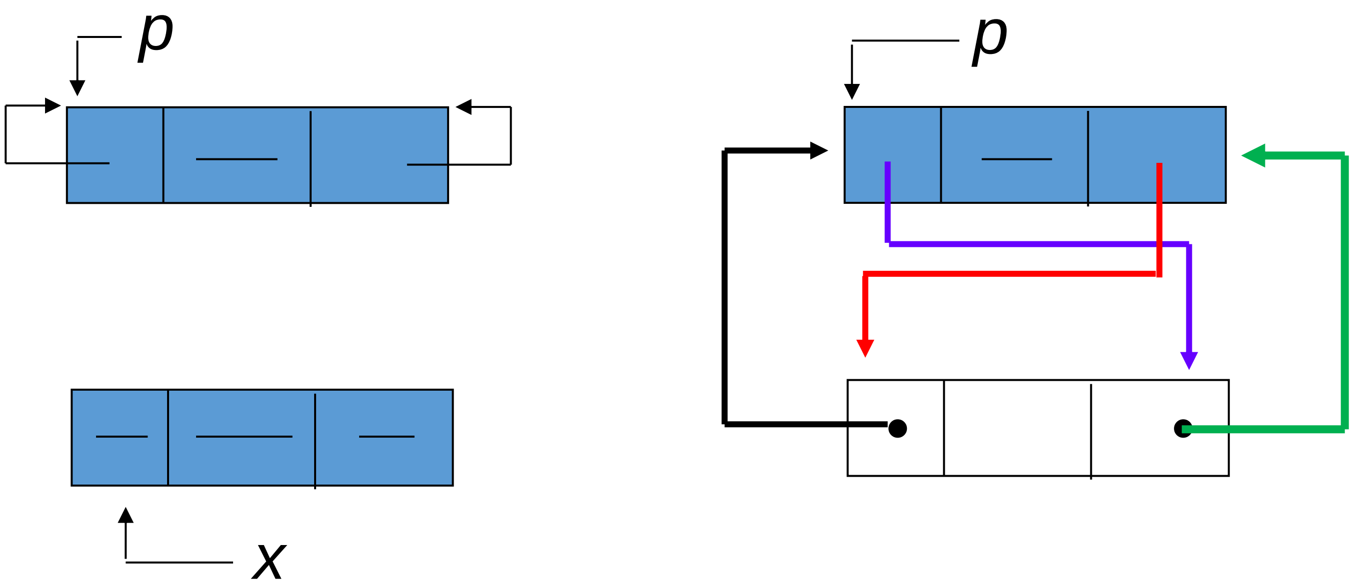




Empty doubly linked list

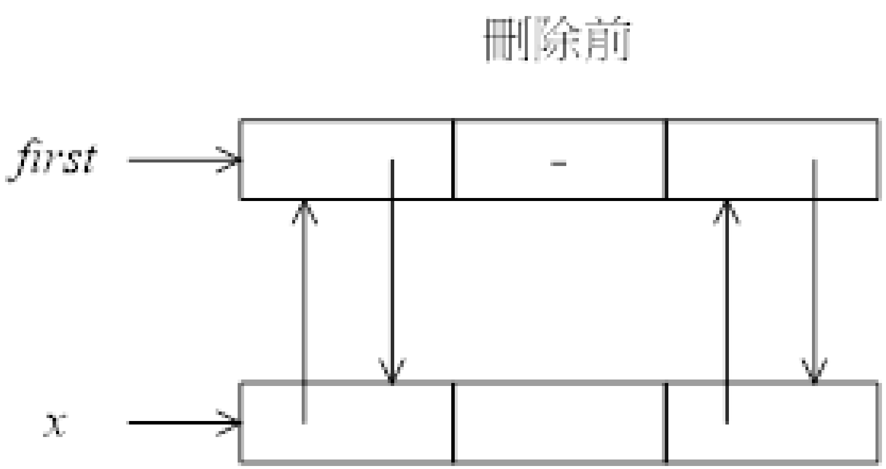
Insertion

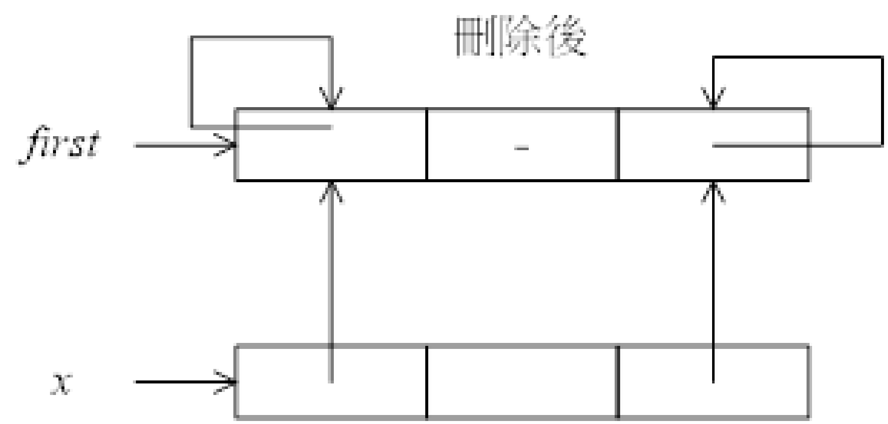
void *DblList* :: *Insert*(*DblListNode* **p*, *DblListNode* **x*)
{ // inset node *p* to the right of node *x*
p→*left* = *x*; *p*→*right* = x→*right*;
x→*right*→*left* = *p*; *x*→ *right* = *p*;
}



Deletion

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



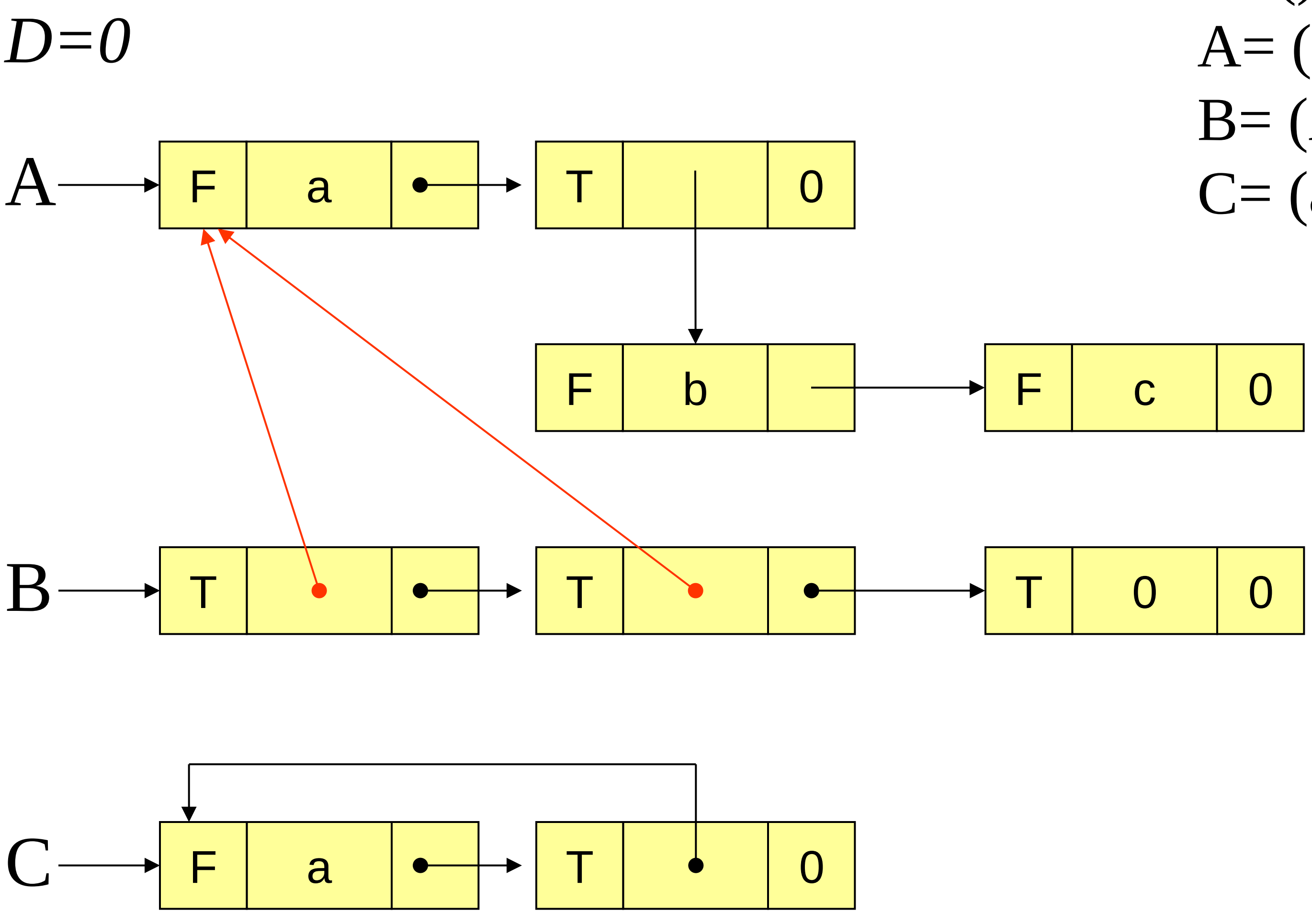


Generalized List

- Generalized List
  - *A generalized list, A, is a finite sequence of* $n \geq 0$ *elements,* $\boldsymbol{a_0, \ldots, a_{n-1}}$*, where* $\boldsymbol{a_i}$ *is either an atom or a list.*
  - *The elements* $\boldsymbol{a_i}$ $0 \leq i \leq n-1$*, that are not atoms are said to be sublists of A.* '
- Examples
  - D=()
  - A= (a, (b, c))
  - B= (A, A, ())
  - C= (a, C)
- Consequences
  - Lists may be empty (Example D)
  - lists may be shared by other lists (Example B)
  - lists may be recursive (Example C)

Generalized List

D=()
A= (a, (b, c))
B= (A, A, ())
C= (a, C)

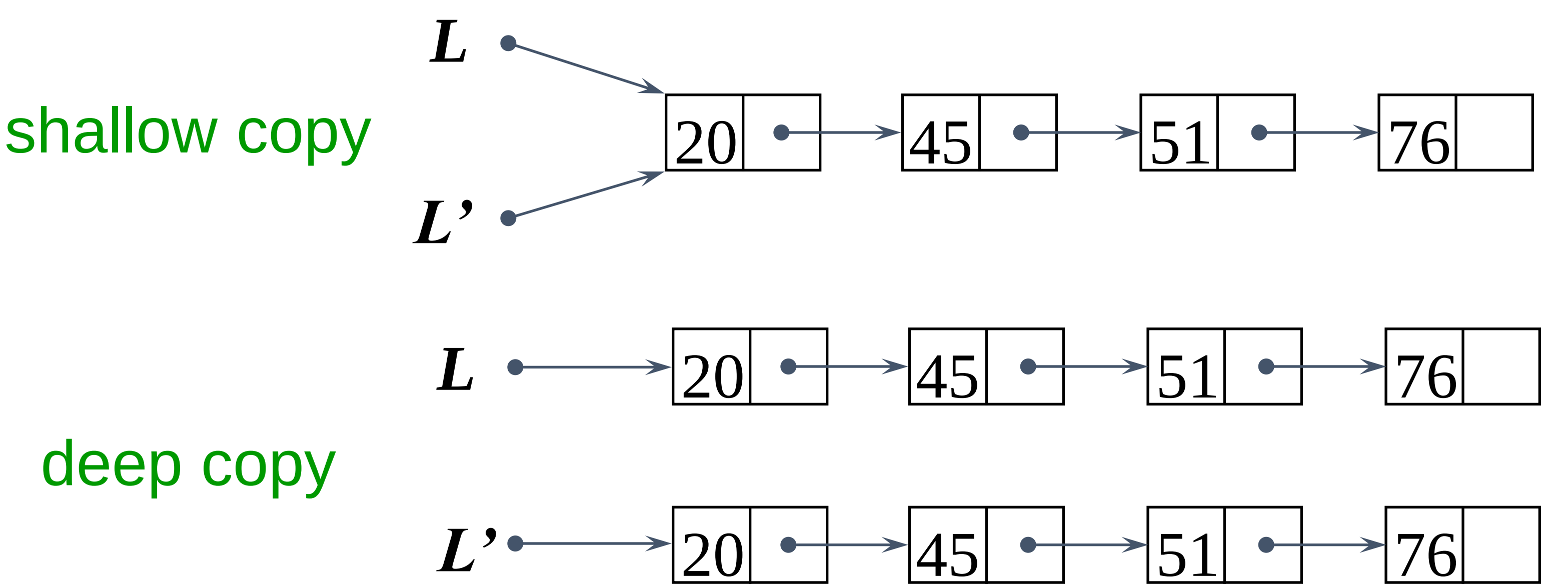


Important Generalized List Functions

- List Copy
  - See textbook pp225-227
  - Program 4.36
- List Equality
  - See textbook pp228
  - Program 4.37
- List Depth
  - See textbook pp229
  - Program 4.38

Lists Copy

- 串列有兩種拷貝的方式：
  - 淺拷貝（shallow copy）：不拷貝資料項目
  - 深拷貝（deep copy）：拷貝資料項目



List Copy with Recursive Algorithms

- Indirect Recursive
- A recursive algorithm consists of two components:
  - The recursive function (the workhorse);
    - Declared as a private function
  - A second function that invokes the recursive function at the top level (the driver);
    - declared as a public function.

List Copy

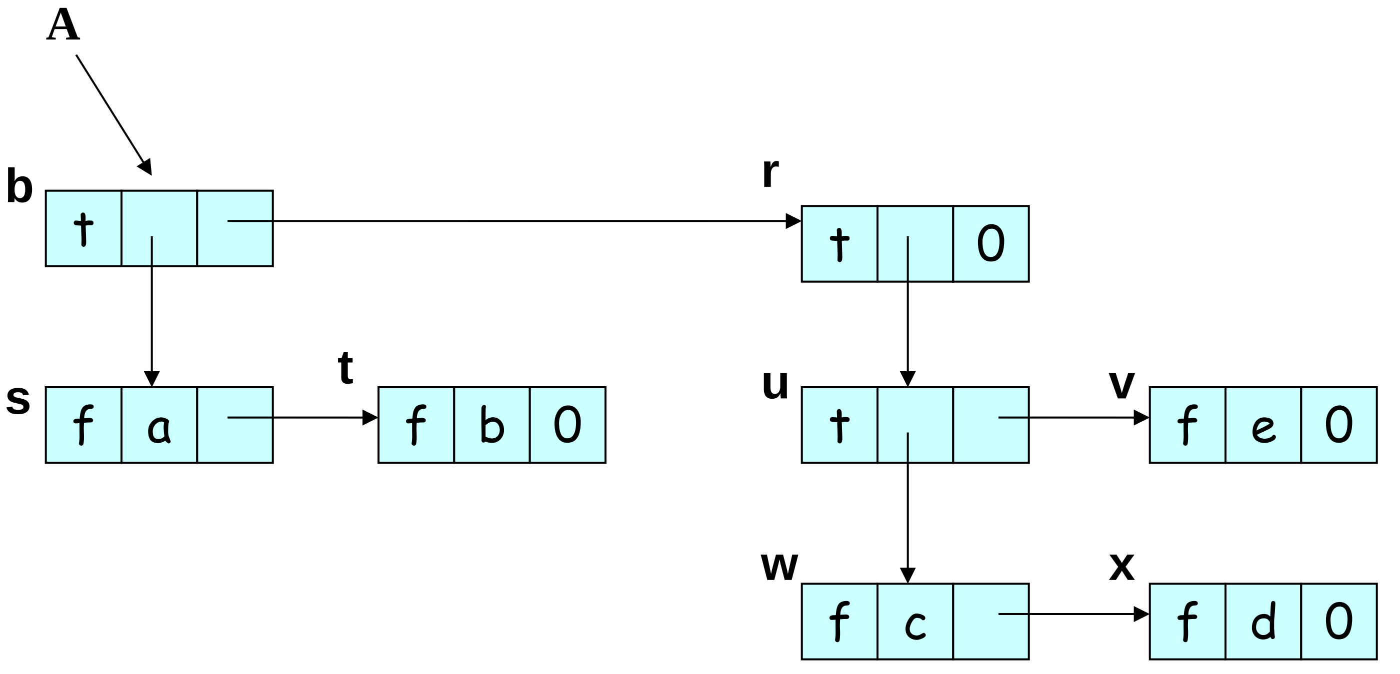
```
void GenList::Copy(const GenList& l)
{
 first = Copy(l.first);
}

GenListNode* GenList::Copy(GenListNode *p)
//Copy the nonrecursive list with no shared sublists pointed at by p
{
 GenListNode *q = 0;
 if (p) {
      q = new GenListNode;
      q->tag = p->tag;
      if (!p->tag)
           q->data = p->data;
      else
           q->dlink = Copy(p->dlink);
      q->link = Copy(p->link);
 }
 return q;
}
```

Driver

Workhorse

$O(M)$

Example: List Copy A (Generalized Lists)



$A((a,b),((c,d),e))$

Example: List Copy A (Generalized Lists)

GenList::Copy(A)

Level of recursion	Value of p	Continuin g level	p	Continuin g 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

List Depth

- The Depth of list is defined as follows.

$$depth(s) = \begin{cases} 0 & if\ \ s\ is\ an\ atom \\ 1 + \max\{depth(x_1), \cdots, depth(x_n)\} & if\ \ s\ is\ the\ list\ (x_1, \cdots, x_n),\ \ n \geq 1 \end{cases}$$

- An empty list has depth 0.