# Data Structures

CH4 Linked Lists

Prof. Ren-Shuo Liu

NTHU EE

Spring 2018
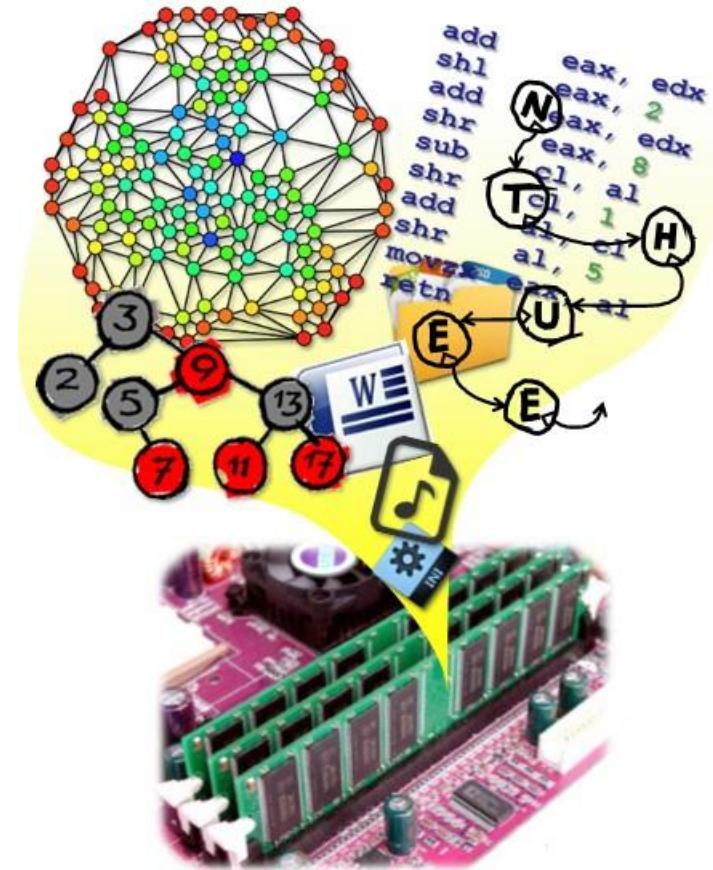
Outline

- 4.1-4.3 Basic singly linked lists and chains

- 4.4-4.5 Circular lists

- 4.6-4.9 Linked stacks, queues, polynomials, and sparse matrices

- 4.10 Doubly linked lists

- 4.11 Generalized lists

Sequential Representation of a List

- Insertion and deletion of arbitrary elements cause excessive data movement

list of three-letter
words ending in AT

0	BAT
1	CAT
2	EAT
3	FAT
4	HAT
5	JAT
6	

Move data

0	BAT
1	CAT
2	
3	EAT
4	FAT
5	HAT
6	JAT

0	BAT
1	CAT
2	DAT
3	EAT
4	FAT
5	HAT
6	JAT

Insert "DAT"
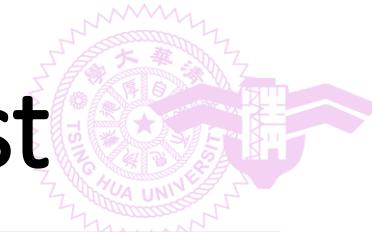
Linked Representation of a List

- Insertion and deletion of arbitrary elements are simplified

Index denoting
the next element

0	BAT	1
1	CAT	2
2	EAT	3
3	FAT	4
4	HAT	5
5	JAT	0
6		

0	BAT	1
1	CAT	2
2	EAT	3
3	FAT	4
4	HAT	5
5	JAT	0
6	DAT	

| 0 | BAT | 1 |
1	CAT	6
2	EAT	3
3	FAT	4
4	HAT	5
5	JAT	0
6	DAT	2

Adjust
indices
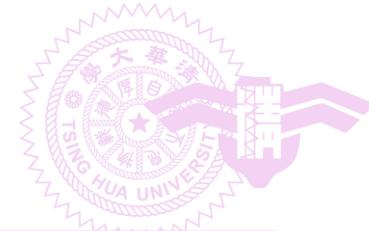
Singly Linked List (Chain)

- Insertion
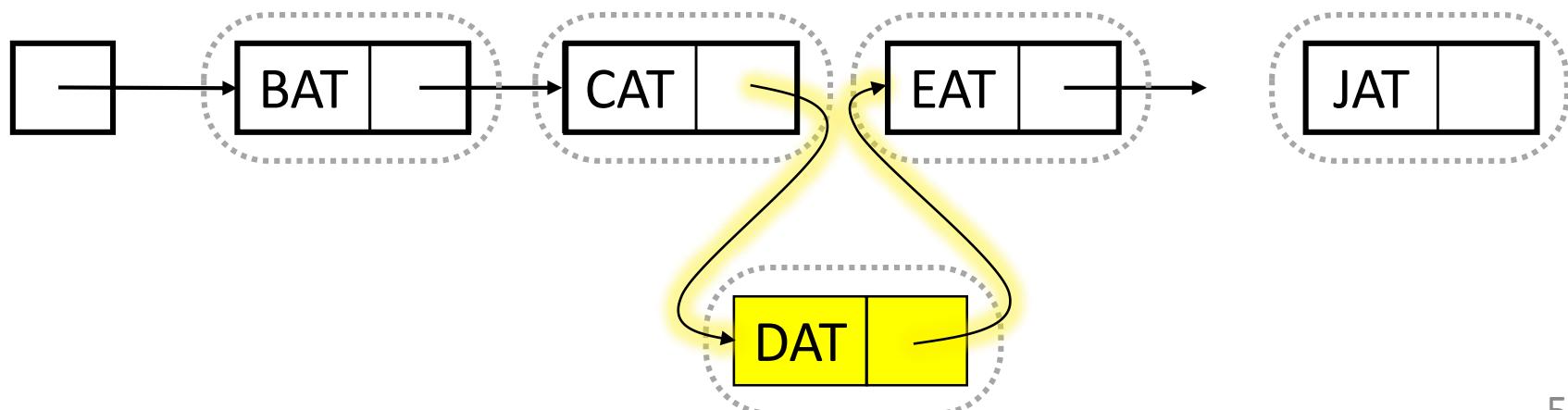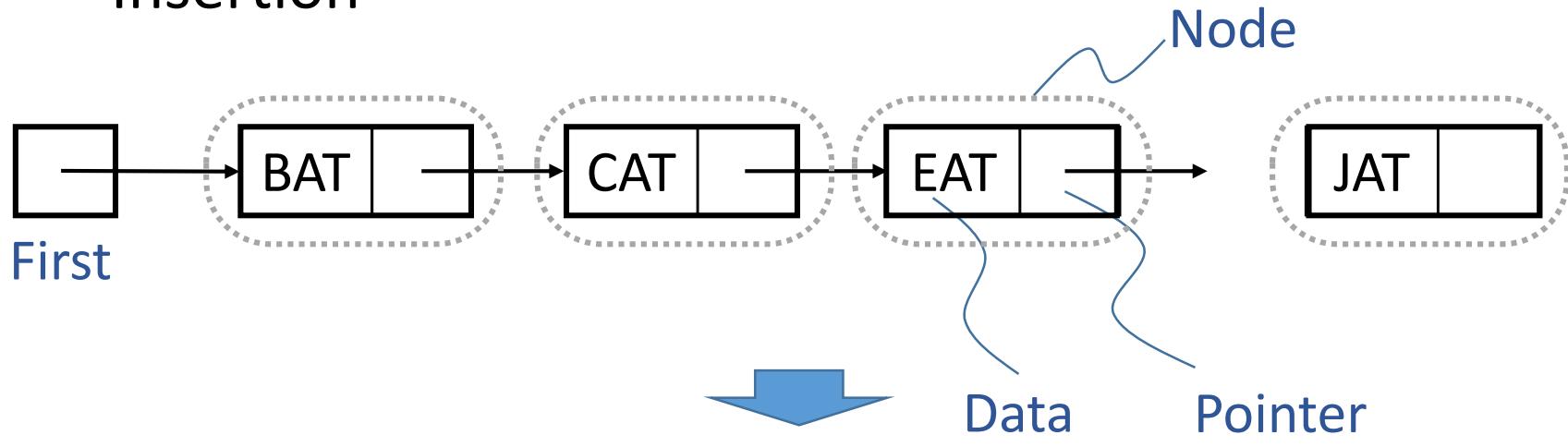
# Chain of Three-Letter Words (Composite Class Style)
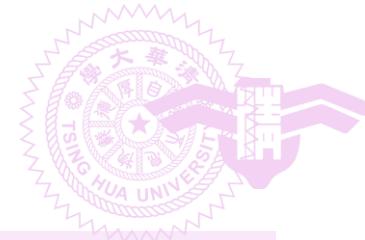
```
class ThreeLetterNode;
// forward declaration
```

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

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

*first

first->data   first->link

first->data[2]

first->data[1]

first->data[0]
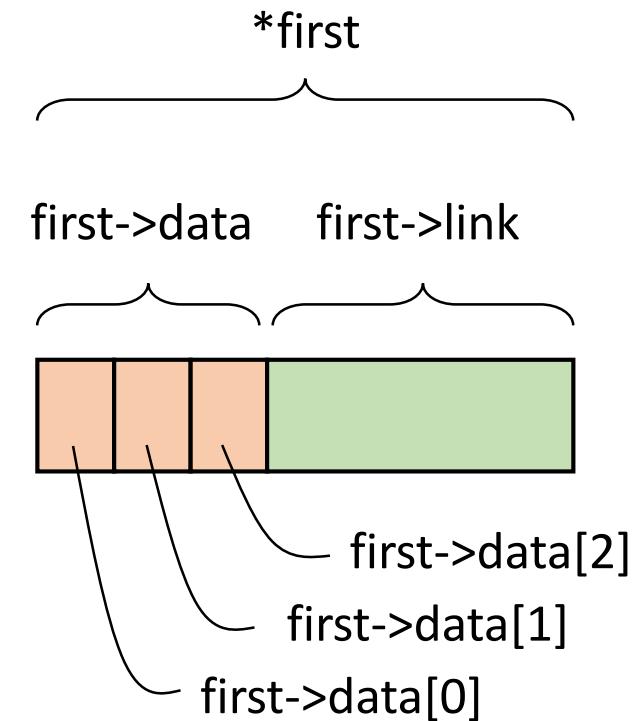
Chain of Three-Letter Words (Nested Class Style)
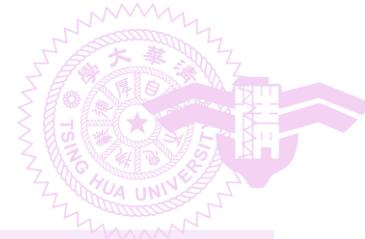
```cpp
class ThreeLetterChain {
public:
    // chain manipulation operations
    …
private:

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

    ThreeLetterNode *first;
};
```

*first

first->data   first->link

first->data[2]

first->data[1]

first->data[0]
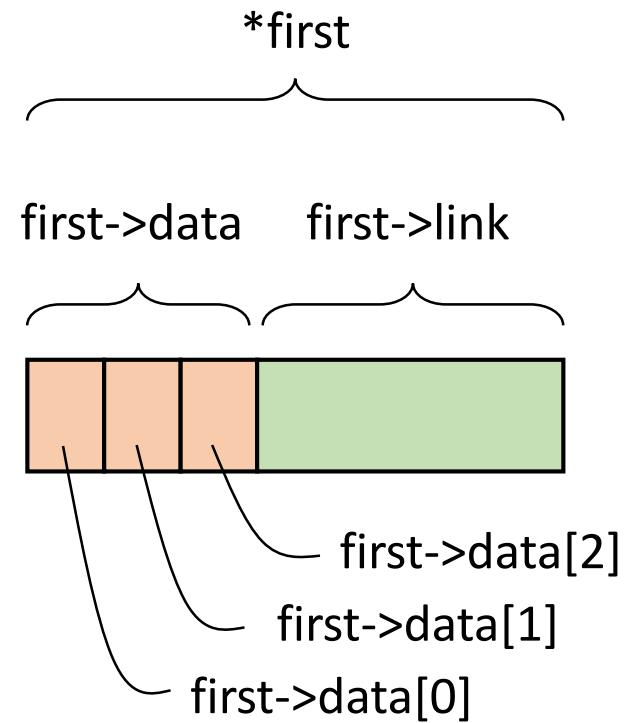
Pointer Manipulation Example

```
ThreeLetterNode * x;
ThreeLetterNode * y;
x = new ThreeLetterNode;
y = new ThreeLetterNode;

y->data[0] = 'B';
y->data[1] = 'A';
y->data[2] = 'T';
y->link = x;

*x = *y;

x = y;

x = y->link;
delete x;
delete y;
```


(For exampling purpose only)
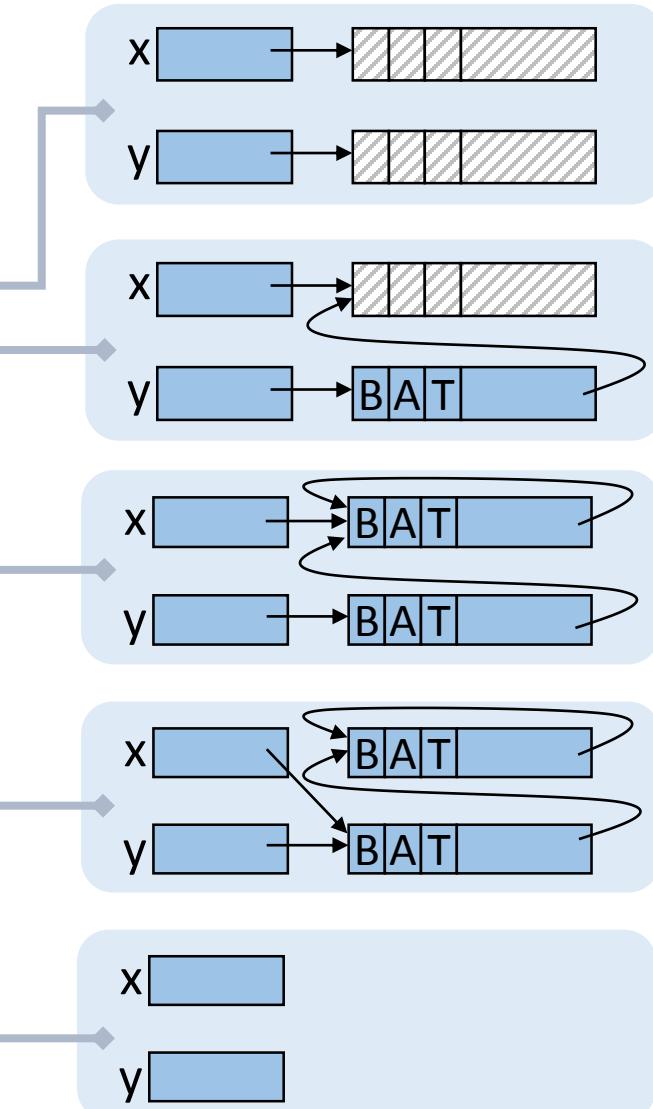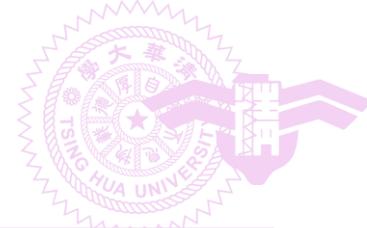
Chain of Integers in C++

```cpp
class ChainNode; // forward declaration
class Chain
{
public:
    // chain manipulation operations
    …
private:
    ChainNode *first;
}

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

Chain Manipulation Operations

- Example: create a chain with two nodes

```
void Chain::func1()
{
    // 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);
}
```
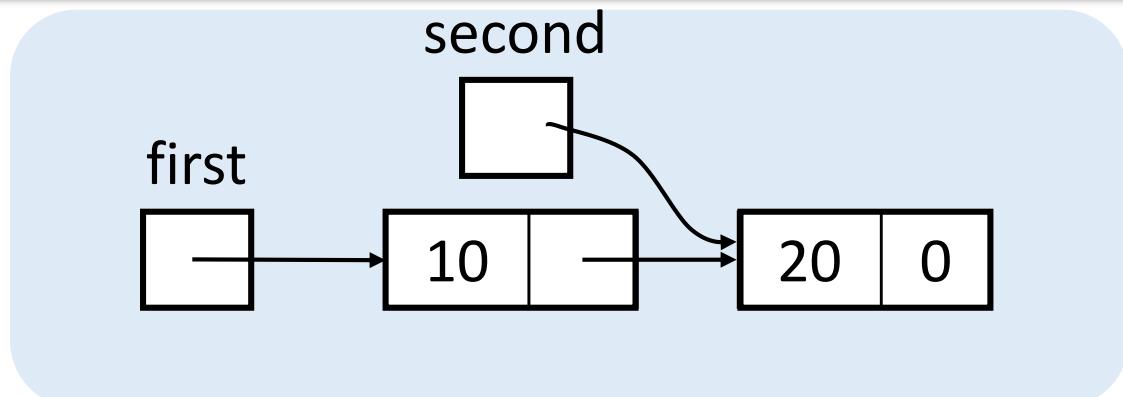
Chain Manipulation Operations

- Insert a node with value 50

```
void Chain::func2(ChainNode* x)
{
    if(first == 0) // insert into an empty chain
        first = new ChainNode(50);
    else            // x is the inserting point
        x->link = new ChainNode(50, x->link);
}
```


empty case

non-empty case

11
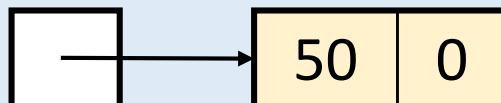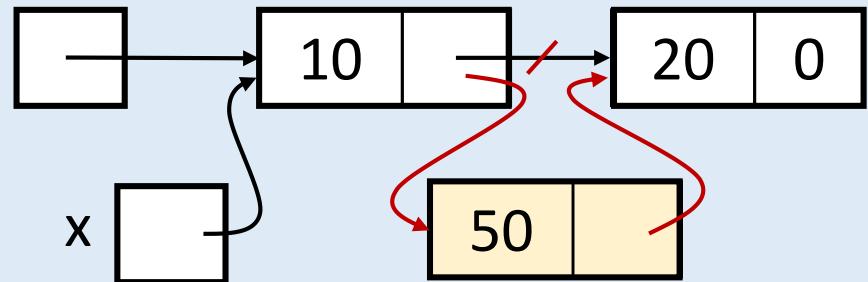
Chain Manipulation Operations

- Delete/remove a node

```
void Chain::Delete(ChainNode* todel, ChainNode* prev)
{
    if(todel == first)
         first = first->link;
    else
        prev->link = todel->link;
    delete todel;
}
```
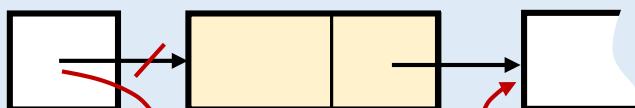

deleting first case
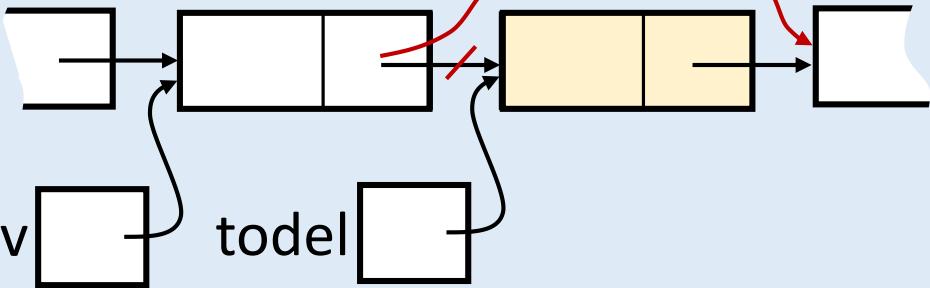
deleting others case

Template Class Chain

- It is a waste of time for programmers to develop different types of chains and their associated algorithms over and over
  - Three-letter word chain
  - Integer chain
  - …

- It is desirable for programmers to have a reusable chain class
  - Template and iterators can help
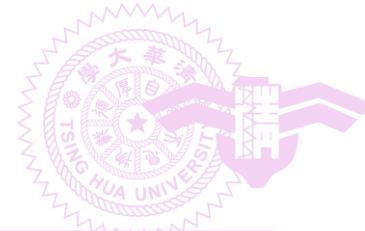
Template Class Chain

```cpp
template <class T> class ChainNode;   // forward declaration
```

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

```cpp
template <class T>
class ChainNode {
friend class Chain <T>;
private:
    T data;
    ChainNode<T>* link;
};
```

```cpp
Usage:

Chain<int> i_list;
Chain<float> f_list;
Chain<Rectangle> r_list;
…
```

Motivation of Chain Iterators

- Consider the following tasks for a container containing integers
  - Print out all integers
  - Find the max, min, median, or mean of all integers
  - Find the sum, product, or sum of squares of all integers
  - Find an integer that maximizes a function

- All these require us to examine all elements of the container

Motivation of Chain Iterators

- Basic method to print out all elements

```
1  for each item in C   // C is a container
2  {
3      k = current item of C;
4      cout << k
5  }
```

- First drawback of the naïve approach: implementation is container-dependent
  - For an array

```
1  for(int i = 0; i < n; i++)
3      k = a[i];
```

  - For a list

```
1  for(ChainNode<int>* ptr = first; ptr != 0; ptr = ptr->link)
3      k = ptr->data;
```

Motivation of Chain Iterators

- Second drawback of the naïve approach : operations need to be member functions as they access private data
  - However, not all operations are meaningful for all classes
    - e.g., sum is meaningful for Chain<int> but not for Chain<Rectangle>
  - Number of member functions can become large
  - Infeasible for the class designer to predict all the operations
  - Class users have to learn how the container class is implemented when they want to add member function

C++ Iterators

- Object type that mimics pointers
- Typically, a nested class of a container class
- Designed to systematically access the elements of a container class one by one
- User can employ iterators to implement their own functions

- Design philosophy
  - Implementing iterators is tedious for designers
  - Users' jobs are simplified

C++ Iterators

- Support pointer-like operations
  - Using operator overloading

- Iterators are classified into five categories according to the supported operations
  - All categories support ==, !=, and * (dereferencing)
  - Input iterators: support read access to the elements pointed at and ++
  - Output iterators: support write access to the elements pointed at and ++
  - Forward iterators: support both input and output
  - Bidirectional iterators: additionally support --
  - Random access iterators: additionally support jumps by arbitrary amounts

input   output

forward

bidirectional

Random access

19
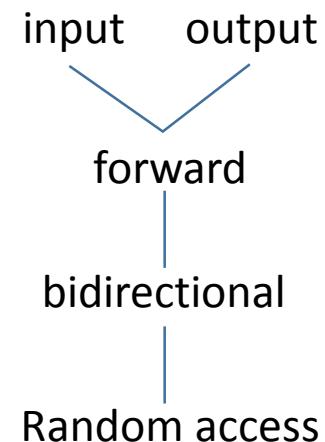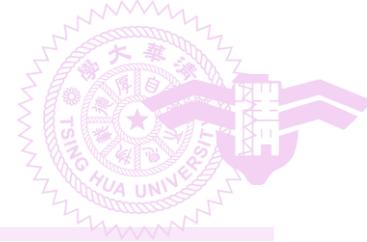
Use of Iterators

- Print-out function for a chain of integers

```
for(Chain<int>::Iterator i = C.begin(); i!=C.end(); i++)
{
    k = *i;
    cout << k;
}
```

- Traverse chain nodes by incrementing an iterator
- Access chain nodes through dereferencing
- Users do not need to directly handle chain operations

Combine Iterators with Template

- Generic print-out function for any container with iterators

```cpp
template <class T>   // T is an iterator type
void print(T begin, T end)
{
    while(begin != end) {
        cout << *begin;
        begin++;
    }
}
```

- Similar to STL sort() and find(), which are also generic functions
- Iterators enable us to decouple the algorithms from containers

Chain Operations

- Design philosophy for choosing which operations to include in an object
  - Provide enough operations so that the class can be used in many applications
  - Not to include too many operations
    - Class becomes bulky and hard to understand

- Example for a chain object
  - Inserting at the back of a list
  - Concatenating two chains
  - Reversing a chain

Inserting at the Back of a List

```cpp
template <class T>
void Chain<T>::InsertBack(const T& e)
{
    if (first) { // nonempty chain
        let last points to the last node
        last->link = new ChainNode<T>(e);
    }
    else first = new ChainNode<T>(e);
}
```
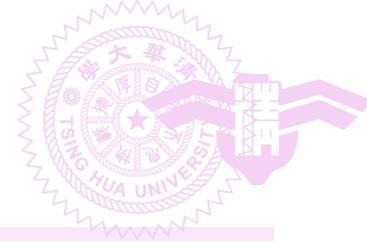
Concatenating Two Chains

```
template <class T>
void Chain <T>::Concatenate(Chain<T>& b)
{
    if (first) { // *this is nonempty
        let last point to the last node
        last->link = b.first;
    }else{
        first = b.first;
    }
    b.first = 0;
}
```
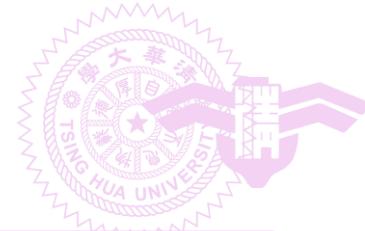
Reversing a Chain

```
template <class T>
void Chain<T>::Reverse()
{
  ChainNode<T> *curr = first,
  ChainNode<T> *prev = 0;                             initializing
  ChainNode<T> *r = 0;
  while (curr) {
    r = prev;          // r trails prev
    prev = curr;       // prev trails curr        advancing
    curr = curr->link; // curr moves to the next
    prev->link = r;    // link prev to r          re-linking
  }
  first = prev;                                        finalizing
}
```
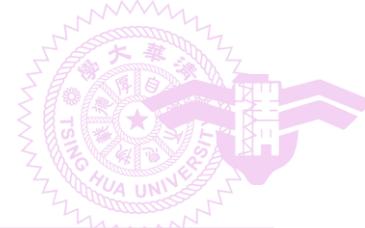
Illustrating Reversing a Chain

Initializing

first

r prev   curr

| 1 | | → | 2 | | → | 3 | | → | 4 | | → | 5 | 0 |

Advancing

first

r   prev   curr

| 1 | | → | 2 | | → | 3 | | → | 4 | | → | 5 | 0 |

Re-linking

first

r   prev   curr

| 1 | 0 | | 2 | | → | 3 | | → | 4 | | → | 5 | 0 |

Illustrating Reversing a Chain

Advancing

first → [1 | 0] [2 | —] → [3 | —] → [4 | —] → [5 | 0]

r   prev   curr

Re-linking

first → [1 | 0] [2 | —] → [3 | —] → [4 | —] → [5 | 0]

r   prev   curr

Advancing + re-linking

first → [1 | 0] [2 | —] [3 | —] → [4 | —] → [5 | 0]

r   prev   curr
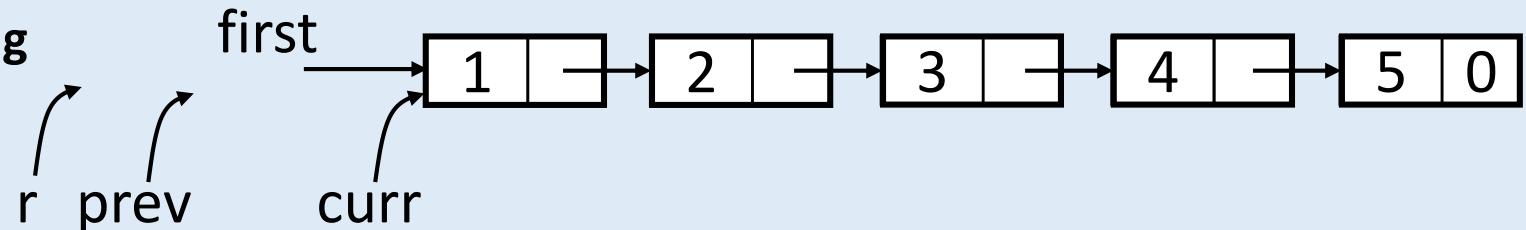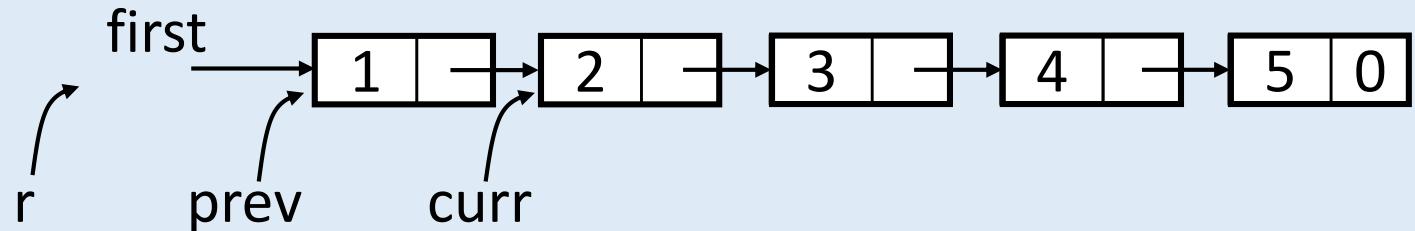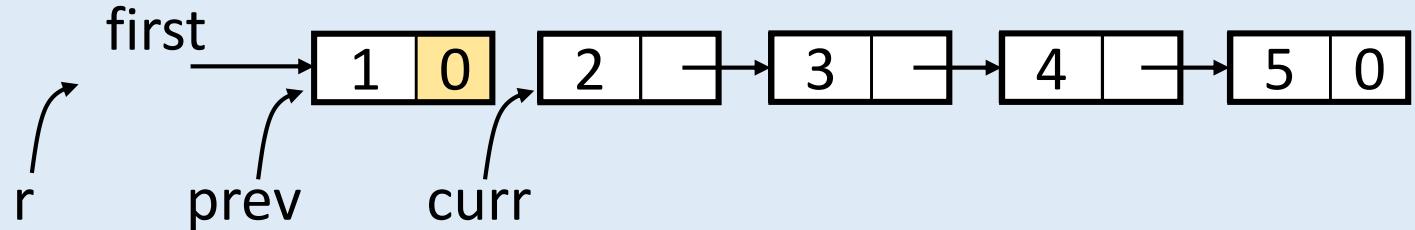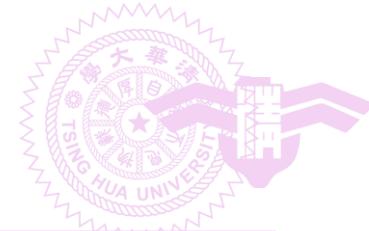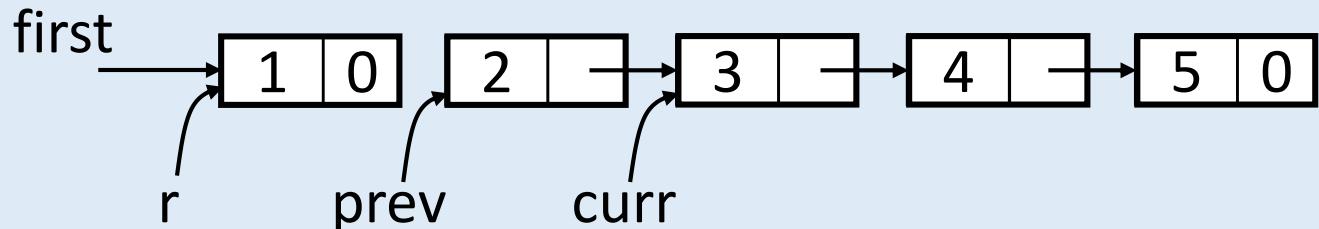
Illustrating Reversing a Chain

Advancing + re-linking
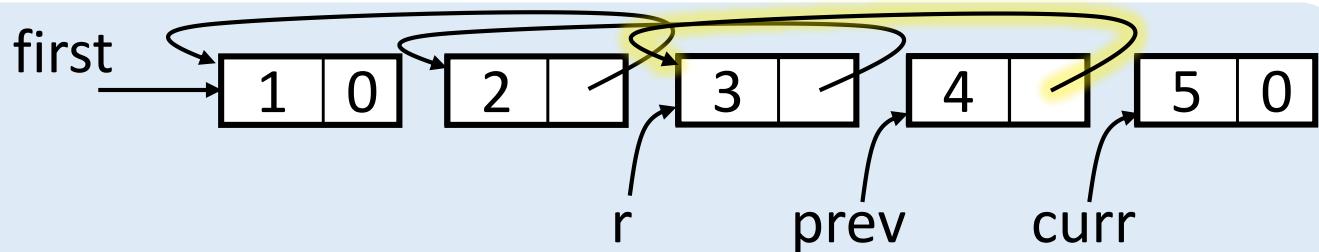
first → | 1 | 0 | | 2 | | | 3 | | | 4 | | | 5 | 0 |

r   prev   curr

Advancing + re-linking

first → | 1 | 0 | | 2 | | | 3 | | | 4 | | | 5 | |

r   prev   curr

Finalizing

| 1 | 0 | | 2 | | | 3 | | | 4 | | | 5 | |

r   first   prev   curr
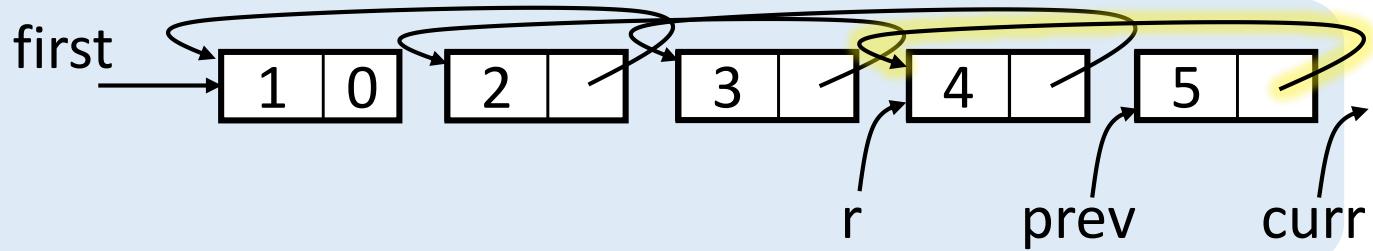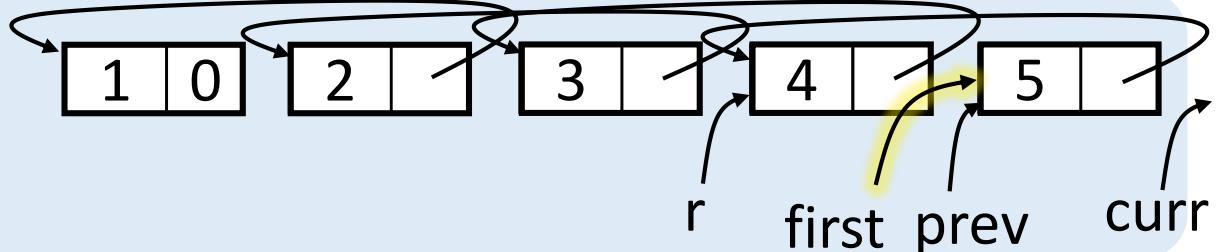
Outline

- 4.1-4.3 Basic singly linked lists and chains

- **4.4-4.5 Circular lists**

- 4.6-4.7 Linked stacks, queues, polynomials, and sparse matrices

- 4.10 Doubly linked lists

- 4.11 Generalized lists

Several Variants of Linked Lists

- Circular list
  - Access pointer pointing to the first node
  - Access pointer pointing to the last node
- Circular list with a header nodes
- Circular list with an available pool

Circular Lists

- Link the last node with the first node
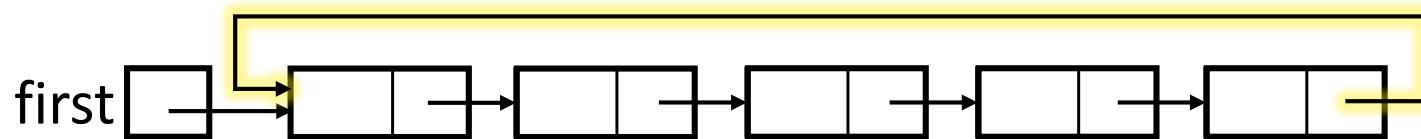
- Advantages
  - Capability of traversing a list with only a pointer pointing to a node in the list
  - Easy to rotate a list

- Potential disadvantages (can be solved)
 - Inserting a node becomes less efficient
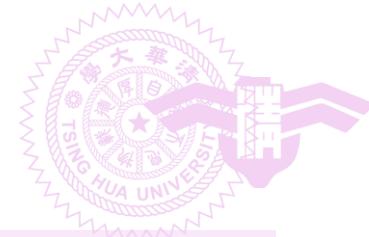
Insert at the Front

- If (first == 0) // empty list

first 0

first ③

tmp ① ②
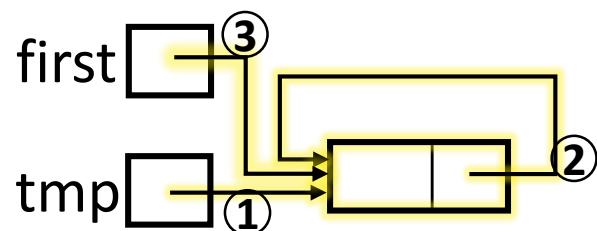
```
① tmp = new node
② tmp->link = tmp
③ first = tmp
```
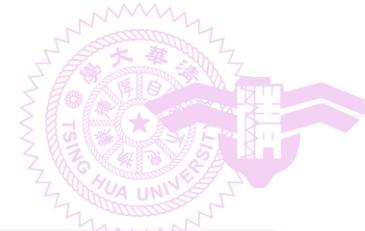
Insert at the Front

- If (first != 0) // non-empty list
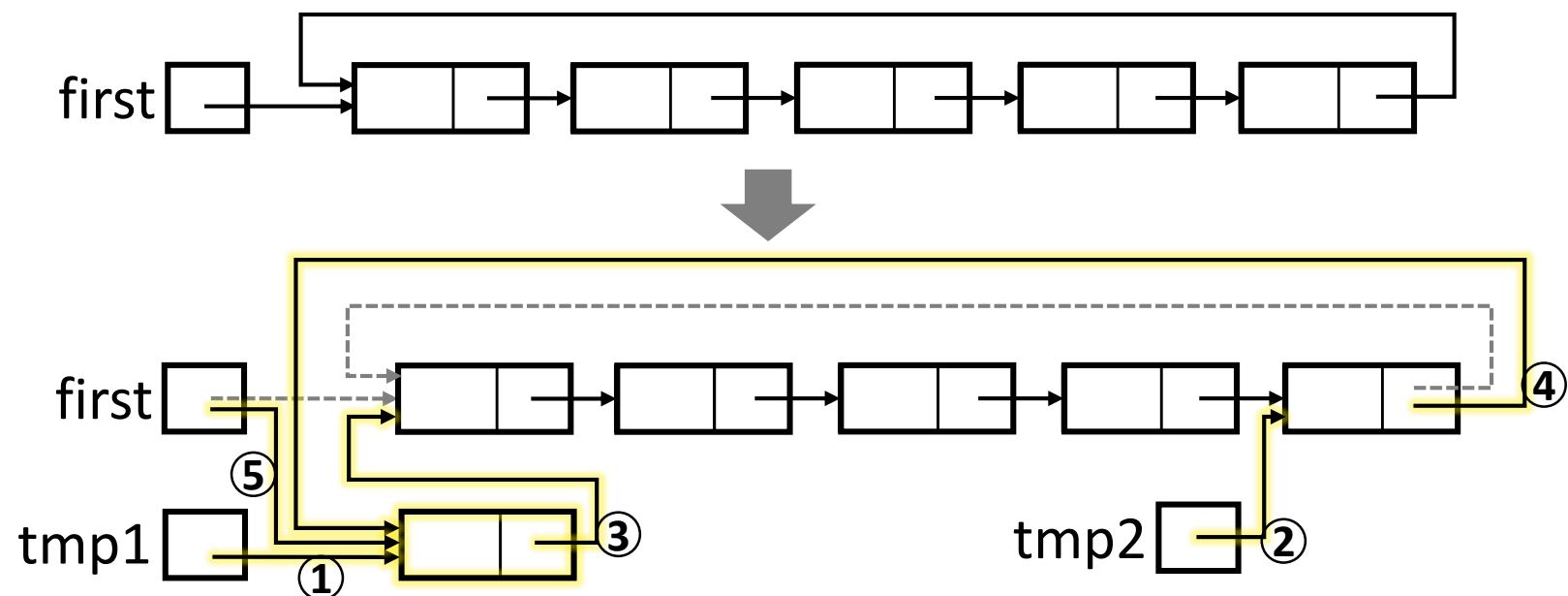
① tmp1 = **new** node
② traverse the whole list
   (Θ(n)) to let tmp2 point
   to the last node.

③ tmp1->link = first
④ tmp2->link = tmp1
⑤ first = tmp1

Quick Summary

- Drawback of circular lists with only an access pointer pointing to the first node
  - Inserting at the front becomes inefficient
    - Last node is involved
    - Time complexity is O(n)
    - Worse than the original non-circular lists

- Solution
  - Let access pointer point to the last node
 - By doing so, inserting at the front consumes constant time

# Insert at the Front

- If (last == 0) // empty list

last ☐0

last ③

tmp ② ①
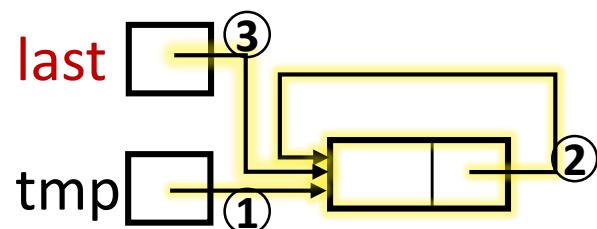
```
① tmp = new node
② tmp->link = tmp
③ last = tmp
```

Insert at the Front

- If(last != 0)

last

tmp ① ② last ③

① tmp = **new** node
② tmp->link = last->link
③ last->link = tmp
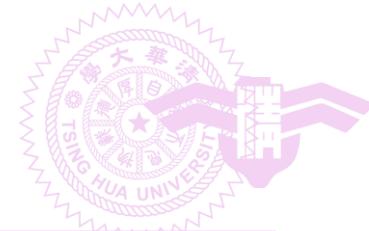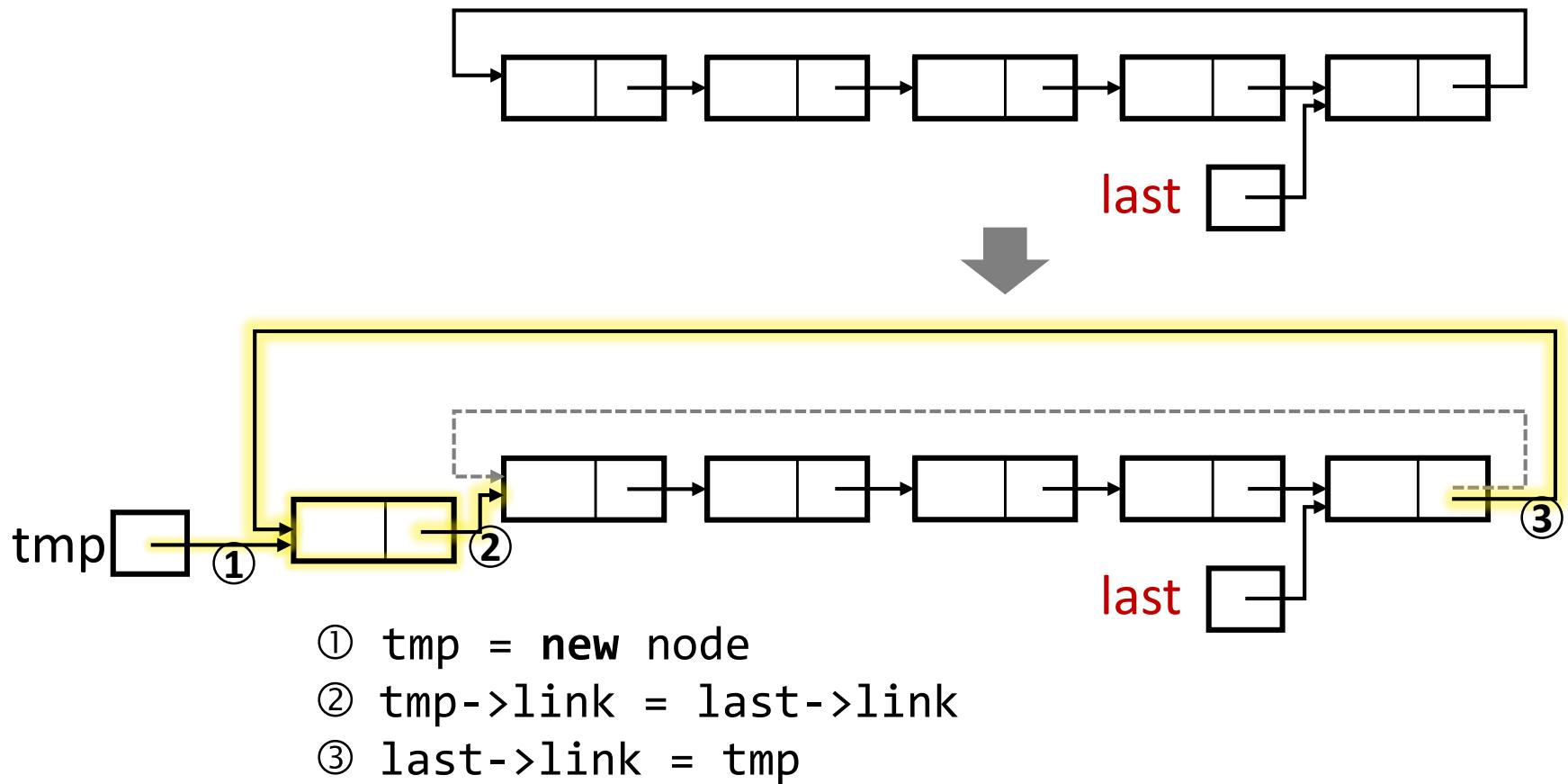
Lists with a Header Node

- Improve data structure regularity
- Avoid special case handling
  - No need to check whether the access pointer is zero
  - Same insertion procedure for both empty and non-empty lists
- Data field of the header node can be left unused or store information about the list (if appropriate)

Empty list

head   →   -
         header

Non-empty list
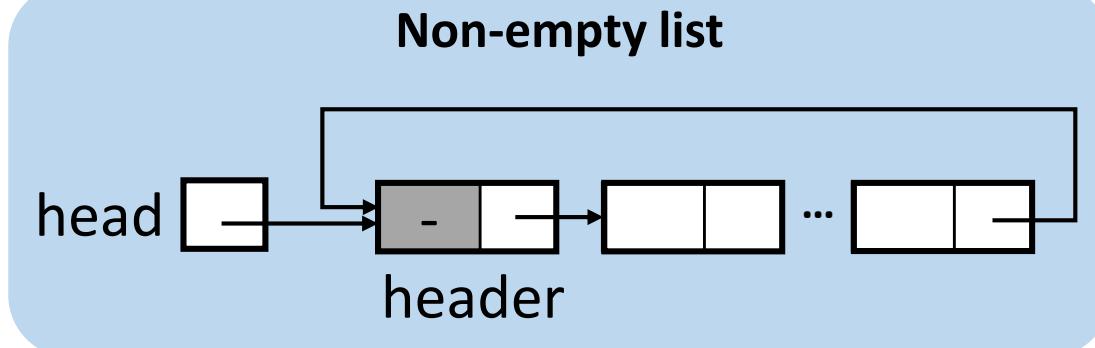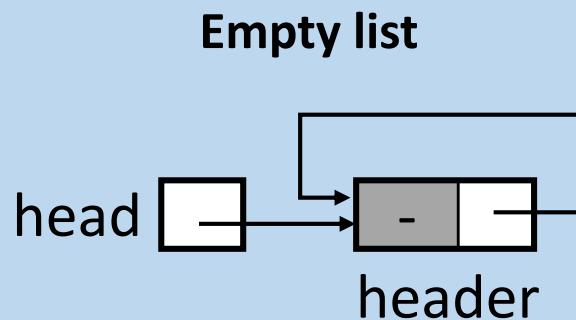
head   →   -   →   □   ···   □
         header
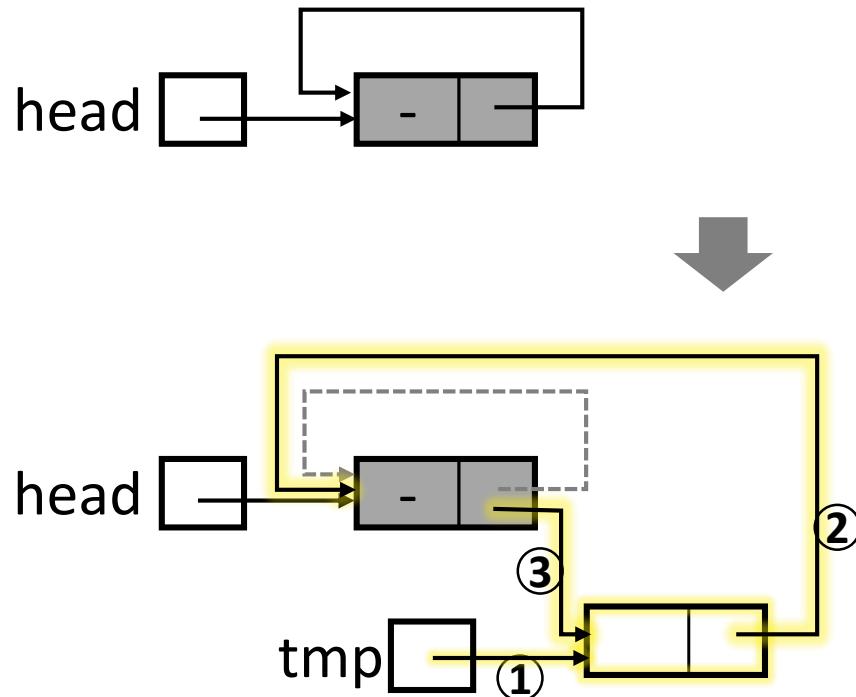
Insert at the Front

① `tmp = new node`
② `tmp->link = head->link`
③ `head->link = tmp`
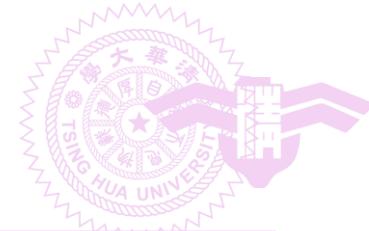
Insert at the Front
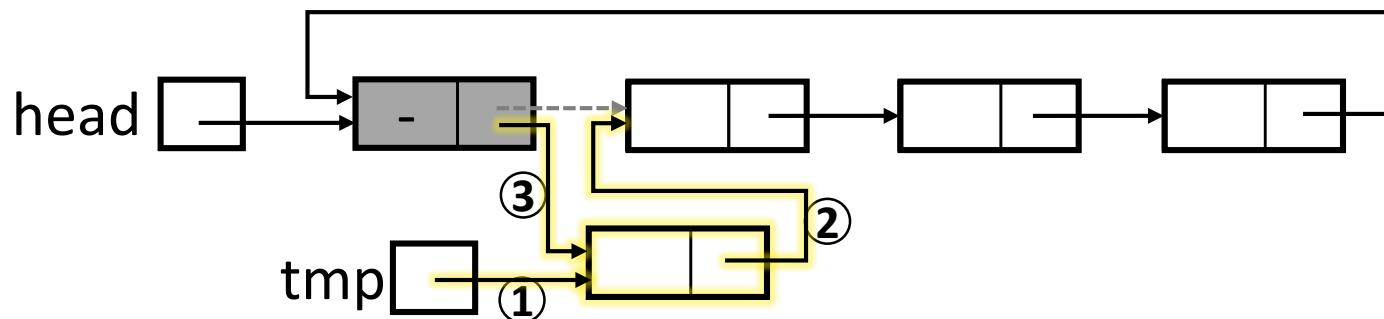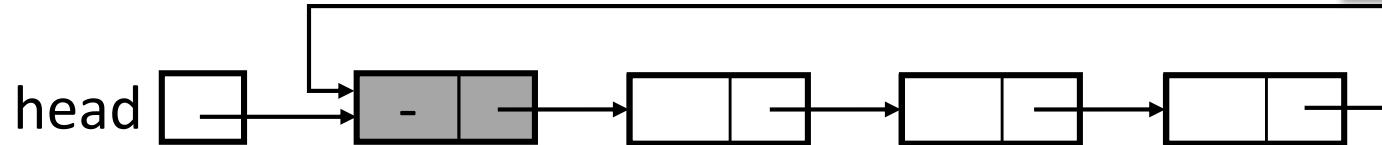
```
① tmp = new node
② tmp->link = head->link
③ head->link = tmp
```
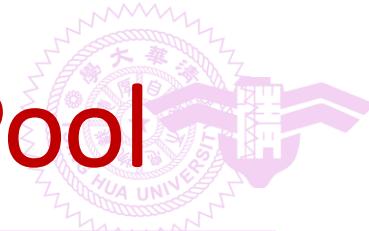
Lists with an Available Node Pool

- Use a pointer to hold pre-allocated and freed nodes
  - The pointer is shared by all the objects of the same class (i.e., the pointer is a **static** class member)
  - Expose GetNode() and RetNode() instead of **new** and **delete** (or one can overload **new** and **delete**)
  - Original **new** and **delete** are used only when necessary

- Benefits
  - Reduces frequent **new** and **delete**, which are costly operations
  - Chain can be deleted in O(1) time
    - Deleting an original chain is of O(n) time

Available Space Lists

List1

List2

List3, List4 …

Available Space Lists

List1

List2

last

av

List2.GetNode()

last

List3, List4 ...

Available Space Lists

List1

List1.RetNode()

last

av - - - - - - - 0

List2

last

List3, List4 …
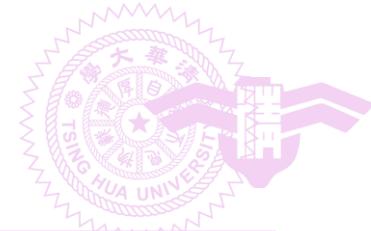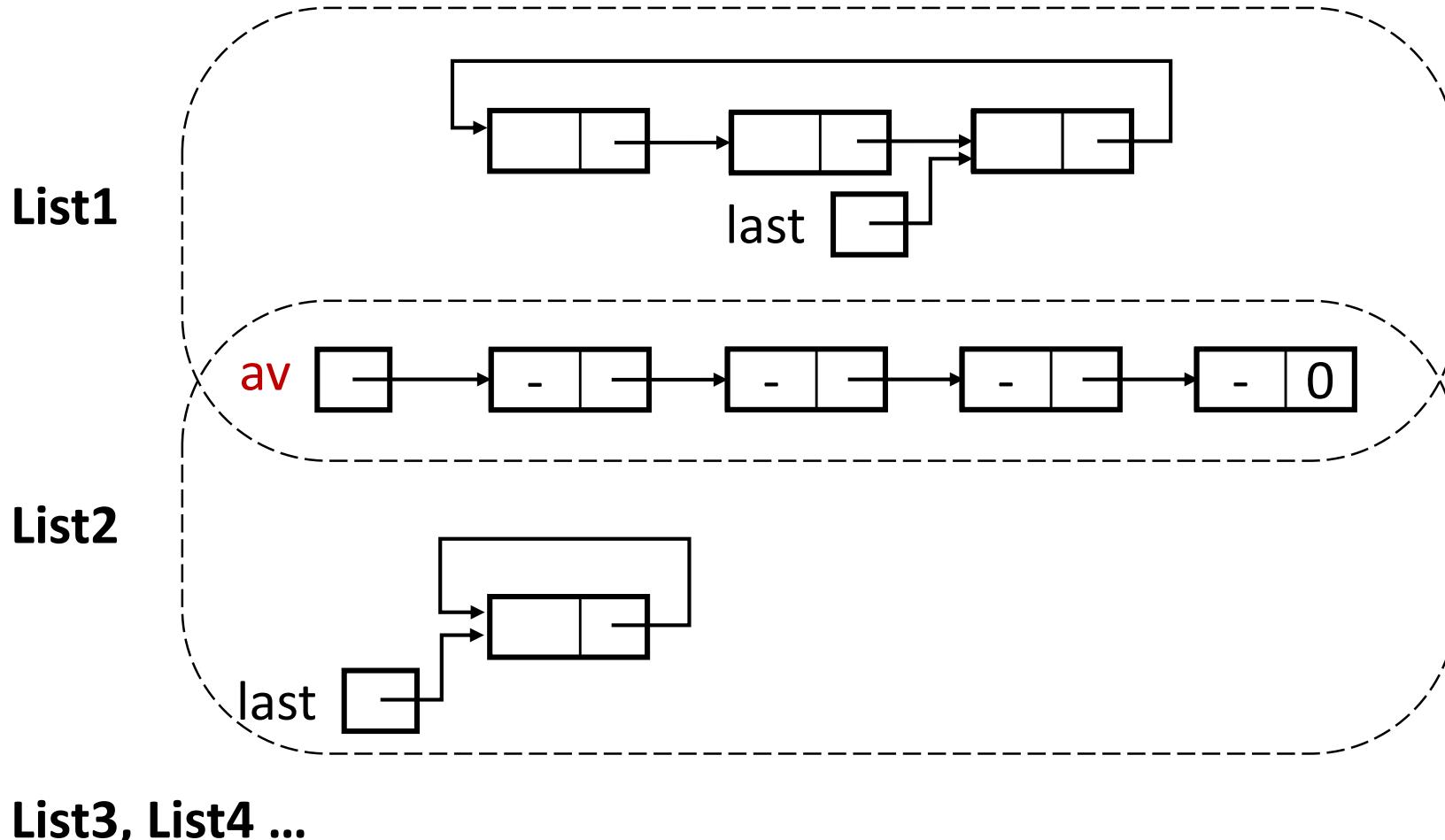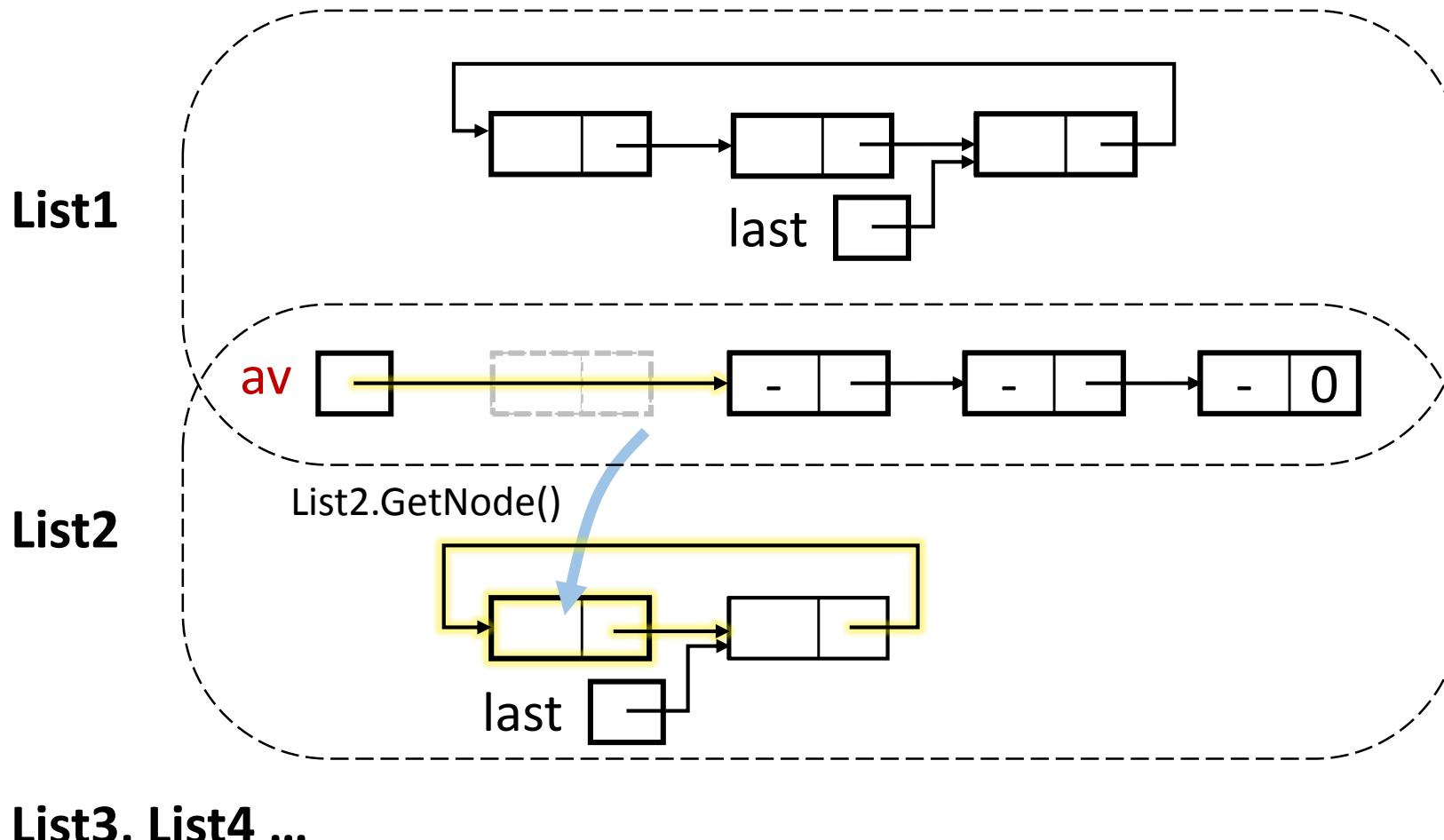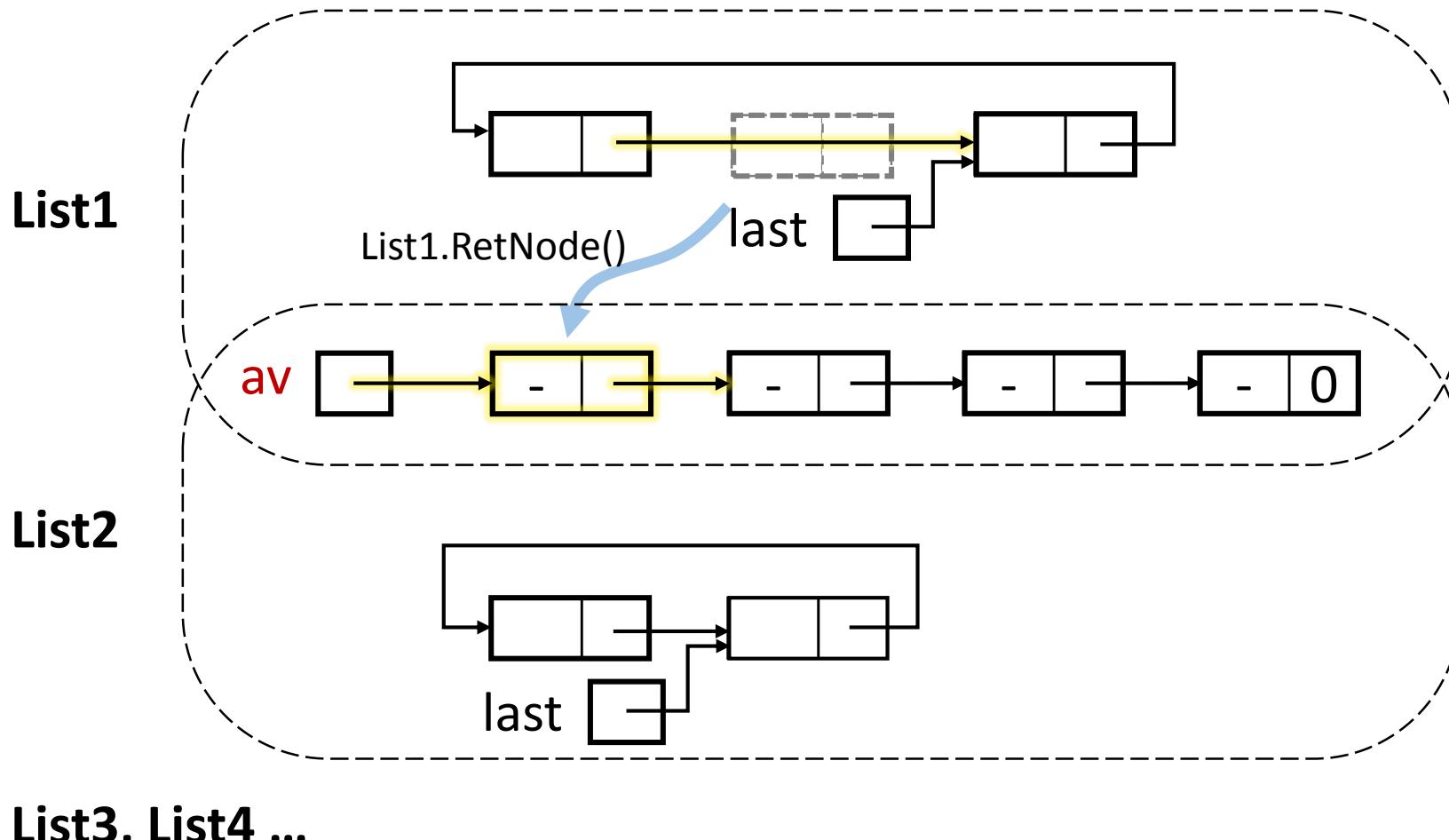
Getting a Node

```
template <class T>
ChainNode<T>* CircularList<T>::GetNode()
{ // Provide a node for use
    ChainNode<T>* x;
    if (av){
        x = av;
        av = av->link;
    }else{ // out of available nodes
        x = new ChainNode<T>;
    }
    return x;
}
```
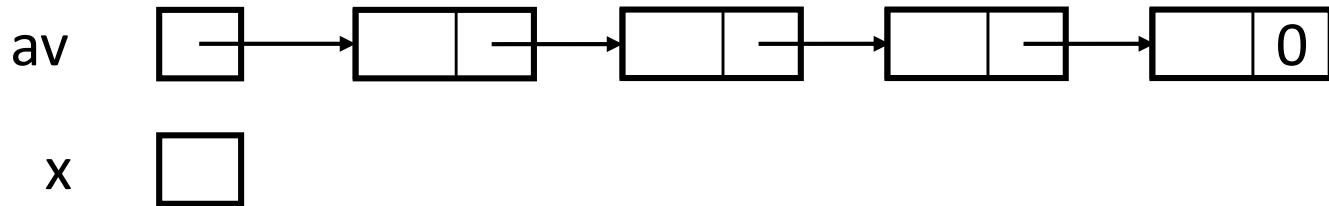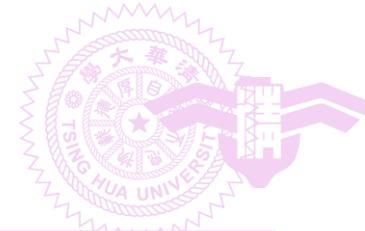
Returning a Node

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

reference to a pointer

x is cleared after being returned. This could prevent a freed node from still being used.

av

x
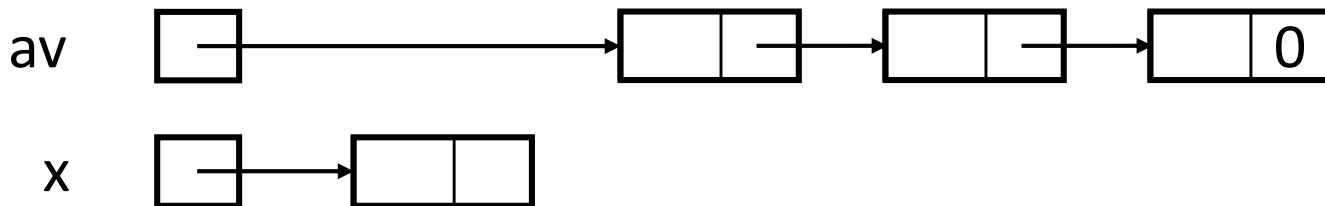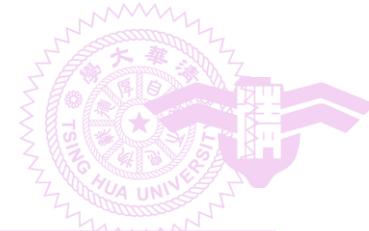
Clearing a List

```cpp
template <class T>
void CircularList<T>::~CircularList()
{// Delete/clear the circular list
    if(last) {
        ChainNode<T>* x = last->link;
        last->link = av; // last node linked to av
        av = x;
        // first node of list becomes front of av list
        last = 0;
    }
}
```
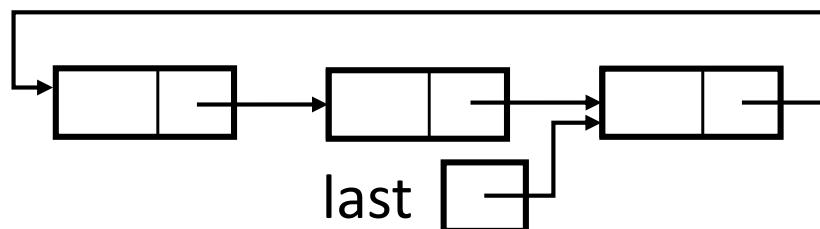
Quick Summary

- Different list implementations have their pros and cons
  - No one-size-fit-all solution
  - No best solution

- Even the vanilla (singly, non-circular) list implementation has its suitable use
  - E.g., managing the available nodes of the available-space lists
  - Usage scenarios in which a list is usually get inserted and deleted at the front and seldom traversed

Outline

- 4.1-4.3 Basic singly linked lists and chains

- 4.4-4.5 Circular lists

- **4.6-4.9 Linked stacks, queues, polynomials, equivalence classes, and sparse matrices**

- 4.10 Doubly linked lists

- 4.11 Generalized lists

Linked Stacks and Queues

- Stack
  - Last in first out

- Queue
  - First in first out

please note the direction
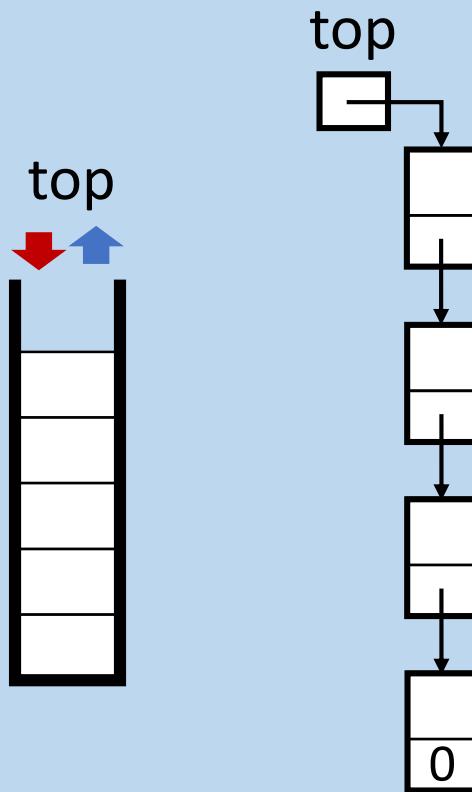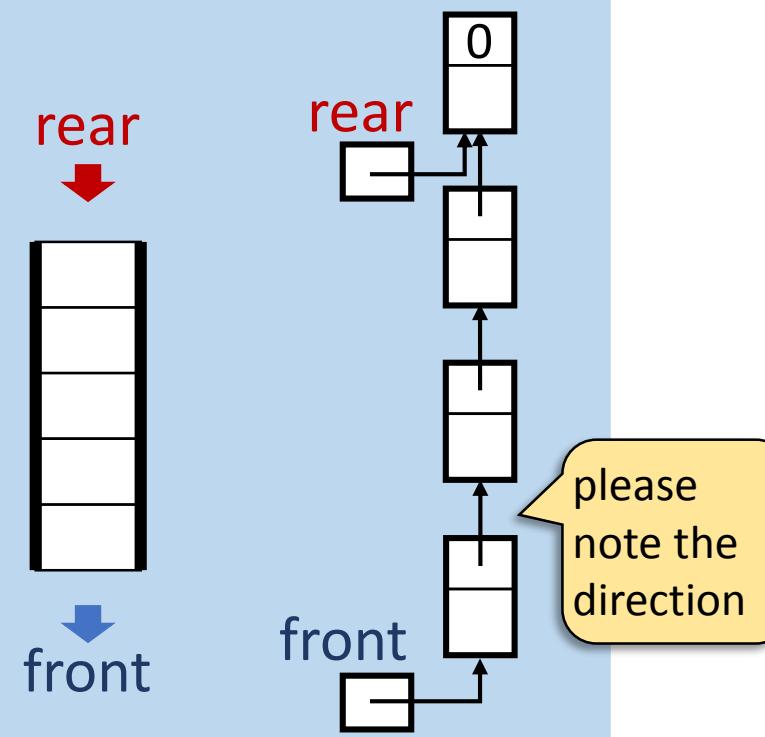
Linked Stack Operations

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

```cpp
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 top node
  delete delNode;  // free the node
}
```
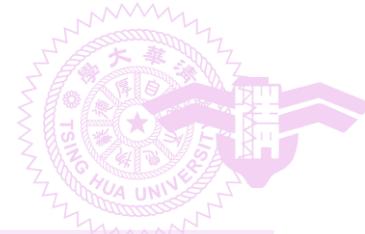
Linked Queue Operations

```
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);
    //attach node and update rear
}
```

```
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 first node from chain
    delete delNode;
}
```

Linked Polynomials

- We can have Polynomials that IS-IMPLEMENTED-IN-TERMS-OF Lists (instead of arrays)

- Definition
    - A data object of Type A IS-IMPLEMENTED-IN-TERMS-OF a data object of Type B if the Type B object is central to the implementation of the Type A object
    - This relationship is usually expressed by declaring the Type B object as a data member of the Type A object
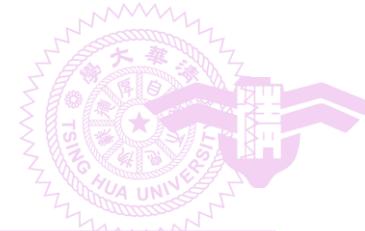
Polynomial Classes

```
struct Term {
    int coef;
    int exp;
    Term Set(int c, int e)
    { coef=c; exp=e; return *this; }
};
class Polynomial{
public:
    // Polynomial operations
private:
    Chain<Term> poly;
}
```

a.poly.first

| 3 | 14 | → | 2 | 8 | → | 1 | 0 | 0 |

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

Term link
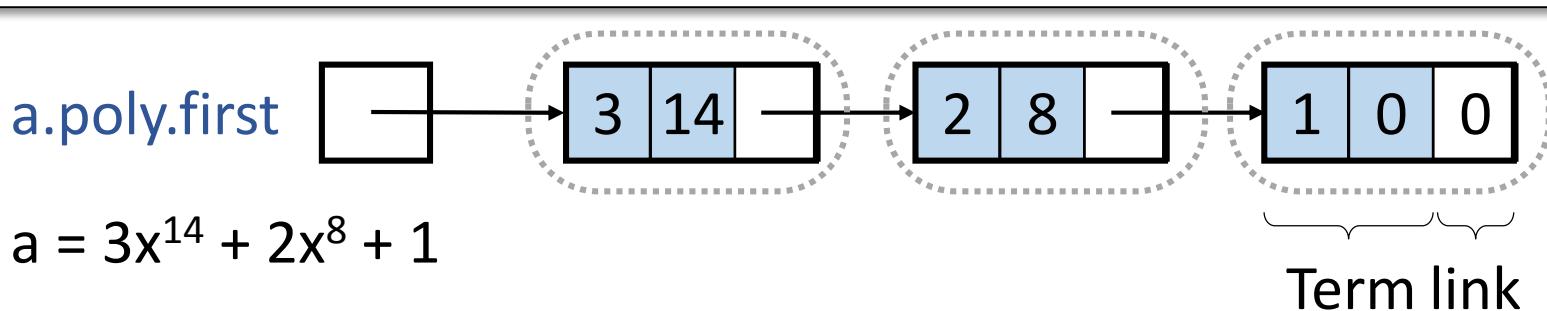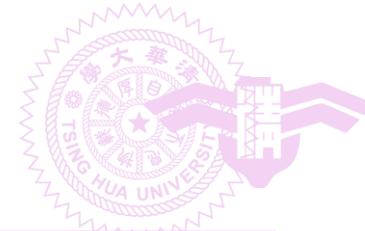
Adding Polynomials

$$3x^{14} + \qquad 2x^8 \qquad + 1$$

$$+ \quad 8x^{14} + -3x^{10} + \qquad 10x^6$$

$$11x^{14} + -3x^{10} + 2x^8 + 10x^6 + 1$$

①　　　　③　　　　⑤

| 3 | 14 | | → | 2 | 8 | | → | 1 | 0 | 0 |

| 8 | 14 | | → | -3 | 10 | | → | 10 | 6 | 0 |

①　　　　②　　　　④
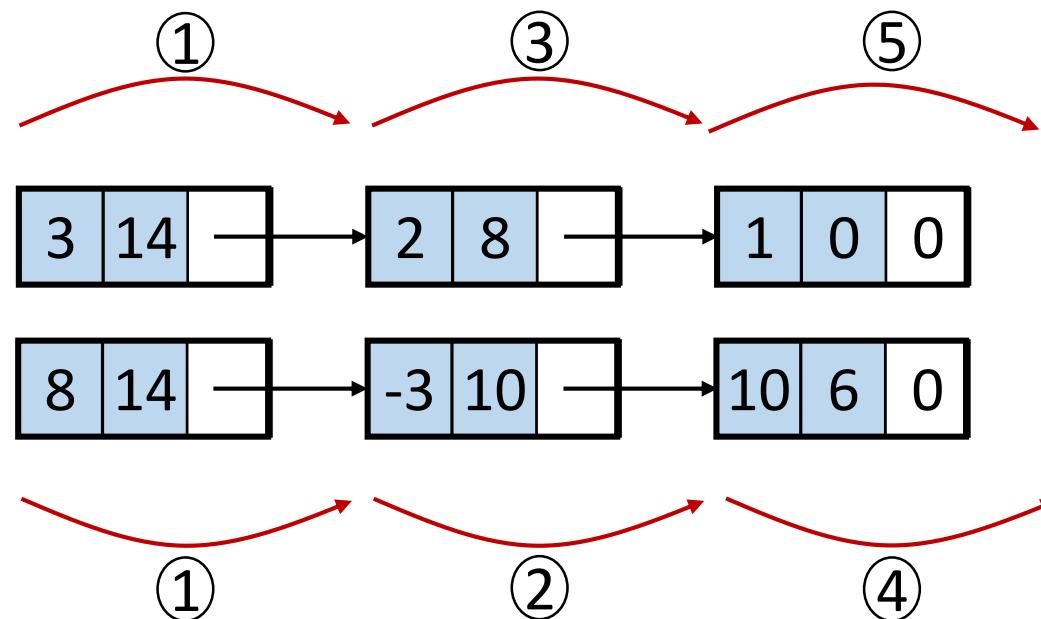
```cpp
Polynomial Polynomial::operator+(const Polynomial& b) const
{
    Term temp;
    Chain <Term>::ChainIterator ai = poly.begin(), bi = b.poly.begin();
    Polynomial c;
    while (ai && bi) {
        if (ai->exp == bi->exp) {
            int sum = ai->coef + bi->coef;
            if (sum) c.poly.InsetBack (temp.Set(sum, ai->exp));
            ai++; bi++;
        } else if (ai->exp < bi->exp) {
            c.poly.InsertBack(temp.Set(bi->coef, bi->exp)) ;
            bi++;
        } else {
            c.poly.InsertBack(temp.Set(ai->coef , ai->exp)) ;
            i++;
        }
    }
    while (ai) {
        c.poly.InsertBack(temp.Set(ai->coef,ai->exp));      ai++;
    }
    while (bi) {
        c.poly.InsertBack (temp.Set(bi->coef,bi->exp));     bi++;
    }
    return c;
}
```

Outline

Equivalence Class Problem

- Integrated circuit (IC) fabrication involves exposing a silicon wafer using a series of masks (光罩)

- Each mask consists of several polygons that define metal lines

- Connected lines form power/signal net

masks

silicon

http://en.wikipedia.org/wiki/Standard_cell
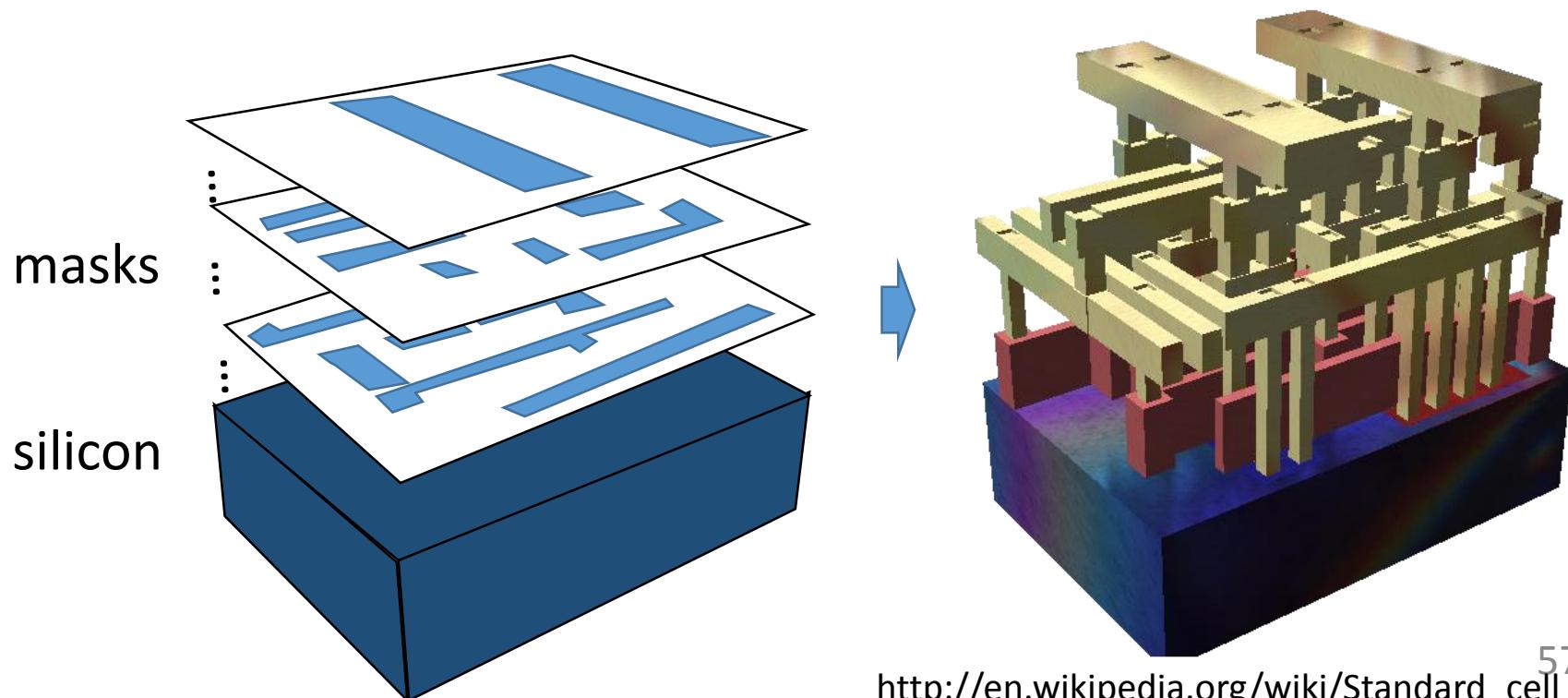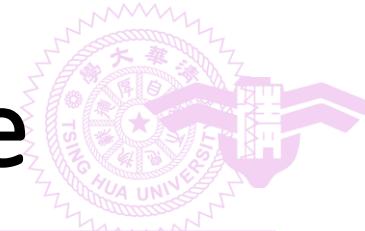
Basic IC Fabrication Procedure

(Supplementary information)
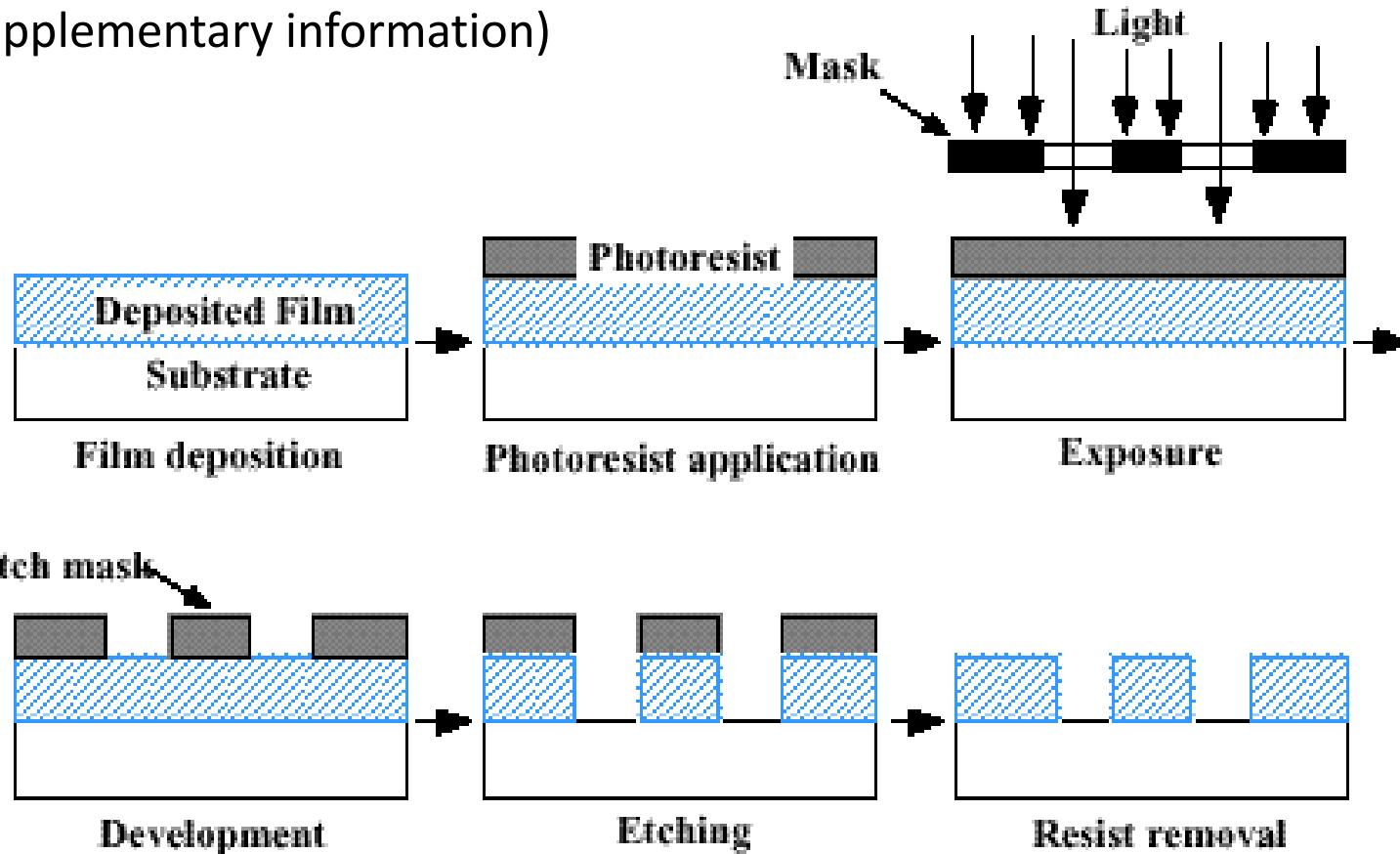
The Fabrication of Integrated Circuits
https://www.youtube.com/watch?v=35jWSQXku74
http://www.hitequest.com/Kiss/VLSI.htm
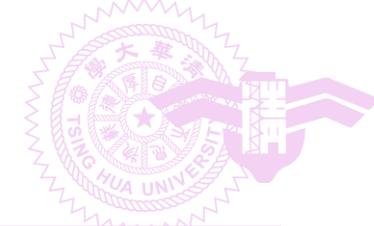
Gallery

(Supplementary information)

Highly magnified view of copper tracks (insulating layers have been removed)
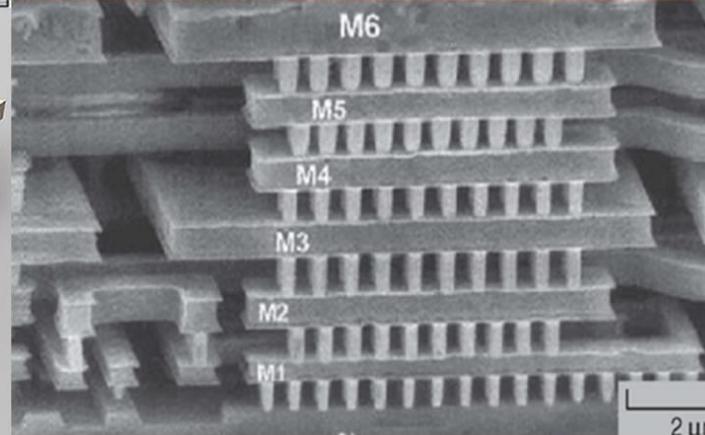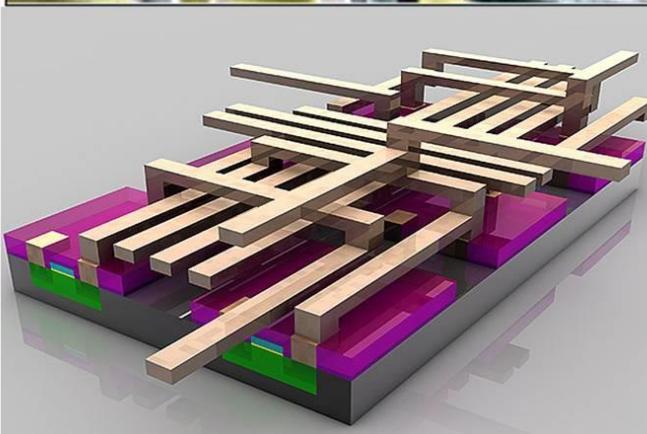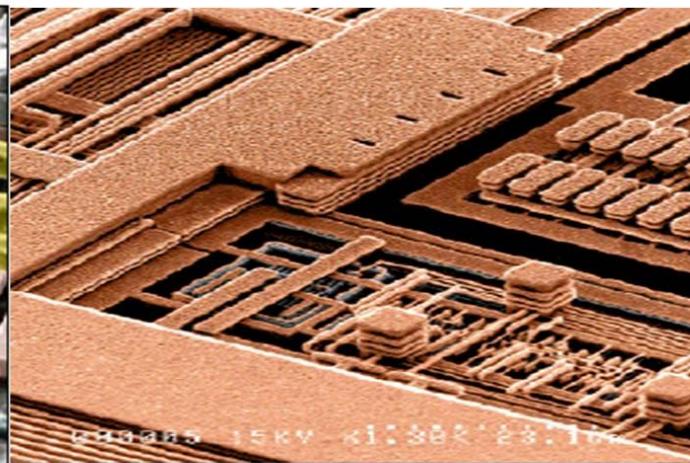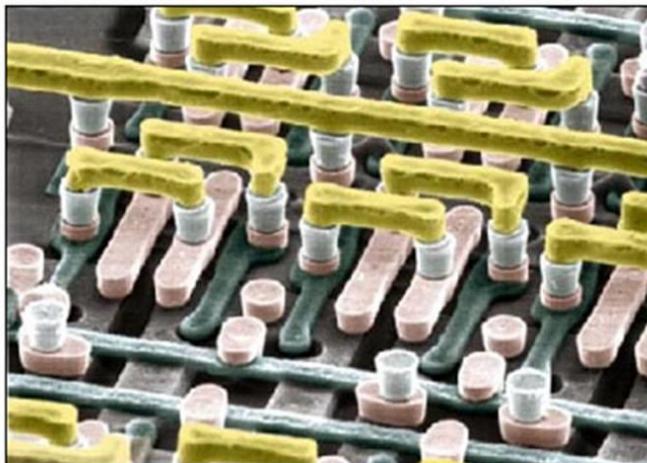
state of the art has a 0.02 μm line pitch

http://www.cense.iisc.ernet.in/nanotwiki.html

Equivalence Class Problem

- We are given polygons and overlap/equivalence pairs (denoted by '≡') among polygons

- We want to partition the polygons into equivalence classes

- For example
  - Input: 0 ≡ 4, 3 ≡ 1, 6 ≡ 10, 8 ≡ 9, 7 ≡ 4, 6 ≡ 8, 3 ≡ 5, 2 ≡ 11, and 11 ≡ 0
  - Output: {0, 2, 4, 7, 11}; {1, 3, 5}; {6, 8, 9, 10}

Properties

- '≡' is reflexive
  - For any polygon x, x ≡ x

- '≡' is symmetric
  - For any two polygons x and y, if x ≡ y, then y ≡ x

- '≡' is transitive
  - For any three polygons x, y, and z, if x ≡ y and y ≡ z, then x ≡ z

Algorithm

- Phase 1
  - Read and store all equivalence pairs (i, j)
- Phase 2
  - Begin at polygon 0 and find all pairs of the form (0, j)
    - 0 and these j's are of the same class
  - Find all pairs of the form (j, k)
    - By transitivity, k is in the same class as 0 and j
  - Continue in this way until the entire equivalence class containing 0 has been found and output
  - Find an object not yet output, which is in a new equivalence class
  - Find and output the new equivalence class as before
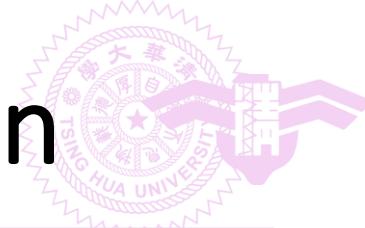
Data Structure Design Decision

- For storing all equivalence pairs (i, j)
  - n×n array
    - e.g., bool pairs[n][n]
    - Result in $\Theta(n^2)$ time complexity
  - Chain representation

0 ≡ 4, 3 ≡ 1, 6 ≡ 10,
8 ≡ 9, 7 ≡ 4, 6 ≡ 8,
3 ≡ 5, 2 ≡ 11, 11 ≡ 0
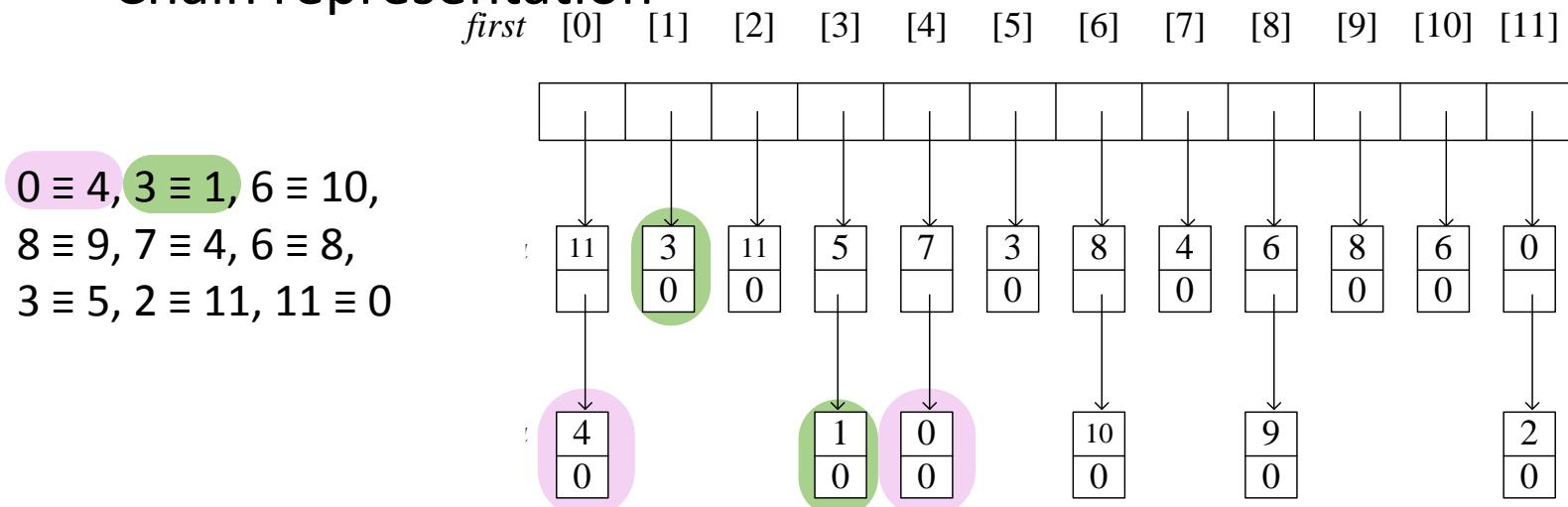
- n-element array, e.g., bool out[n], is used for recording whether objects are output

Example

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

data	11	3	11	5	7	3	8	4	6	8	6	0
link		0	0			0		0		0	0	

data	4			1	0		10		9			2
link	0			0	0		0		0			0
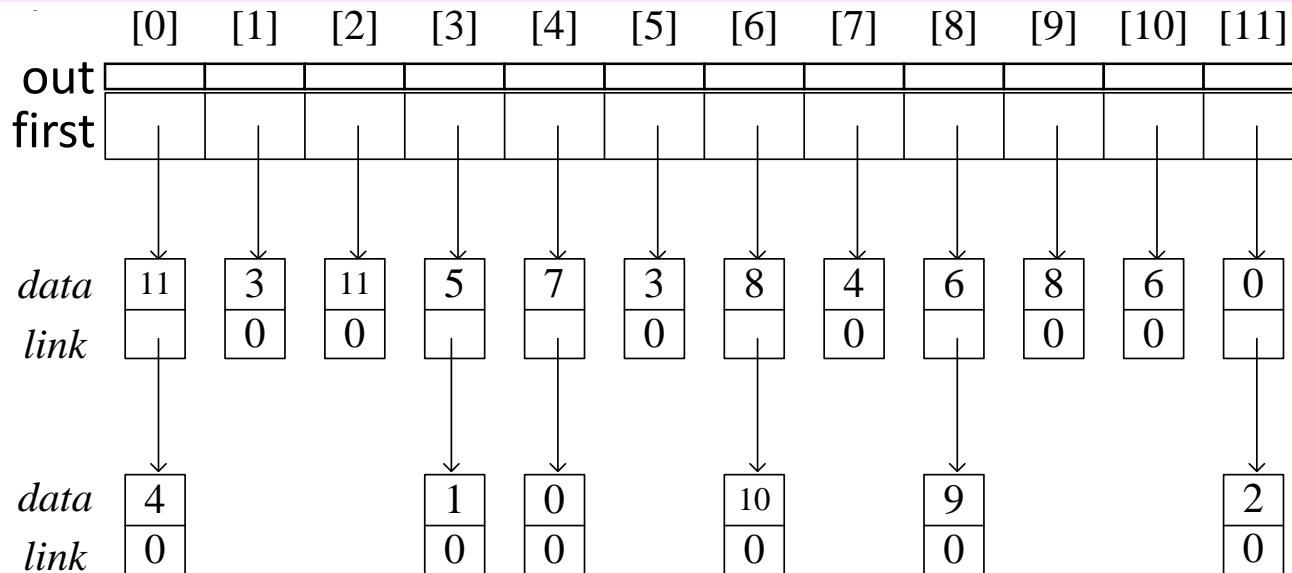
1. (Begin at 0)   (Add 0 into a container, e.g., a stack)
2. (Get 0 from the container)   Output 0   (Add 11 and 4 into the stack)
3. (Get 4 from the container)   Output 4   (Add 7 into the stack)
4. (Get 7 from the container)   Output 7   (No new objects are found)
5. (Get 11 from the container)   Output 11   (Add 2 into the stack)
6. (Get 2 from the container)   Output 2   (No new objects found)

64

Algorithm in C++

```cpp
void Equivalence( )
{
    ifstream inFile("equiv.in", ios::in);
    if (!inFile) throw "Cannot open input file.";
    int i, j, n;
    inFile >> n;  // number of objects
    // initialize first and out
    bool *out = new bool[n];  // an array of bool
    ENode **first = new ENode* [n];  // an array of pointers
    // 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;
    }
```

```cpp
for (i = 0; i < n; i++) // Phase 2
    if (!out[i]) { // this object has not been output yet
        cout << endl << "A new class: " << i;
        out[i] = true;  // mark the object
        stack<Enode> s; // initialize a stack
        ENode *x = first[i]; // obtain the ith list
        while (1) {      // continue processing lists and the stack
            while (x) { // continue processing a list
                j = x->data;
                if (!out[j]) {
                    cout << ", " << j;
                    out[j] = true;
                    s.push(*x); // add *x into the stack
                }else{
                    x = x->link;
                }
            } // end of while(x)
            if (s.isEmpty()) break;
            x = s.top();
            s.pop(); // unstack
        } // end of while(1)
    } // end of if (!out[i])
```
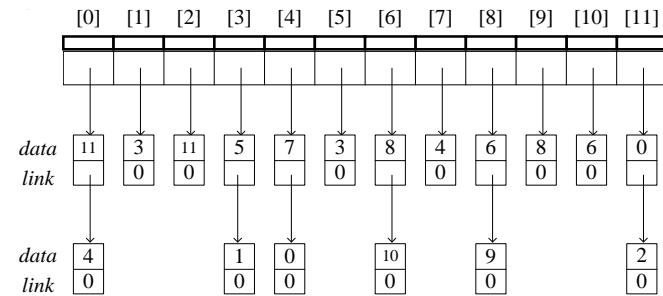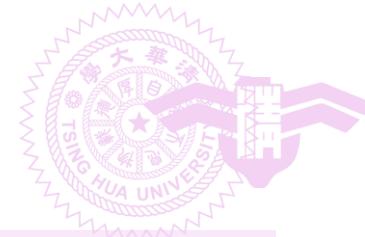
Algorithm in C++ (cont'd)

```cpp
for (i = 0; i < n; i++)  // clean up Phase-1-created objected
    while (first[i]) {
        ENode *delnode = first[i];
        first[i] = delnode->link;
        delete delnode;
    }
delete [] first;
delete [] out;
```

- Time complexity = O(m+n)
  - Initialization of first[] and out[] takes O(n) time
    - where n is the number of objects
  - Processing of each input pair in phase takes O(m)
    - where m is the number of input pairs
  - The **for** loop takes O(n) time
  - Evaluating of the 2m pairs takes O(m) time

Outline

- 4.1-4.3 Basic singly linked lists and chains

- 4.4-4.5 Circular lists

- **4.6-4.9** Linked stacks, queues, polynomials, equivalence classes, and **sparse matrices**

- 4.10 Doubly linked lists

- 4.11 Generalized lists
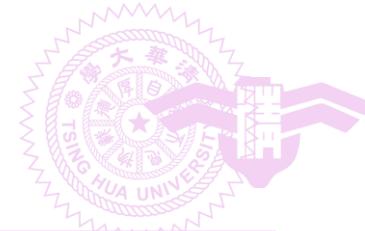
Sparse Matrices

- Array-based representation (we have learned this before)
  - Row access is easy, but column access is difficult

- Linked representation
  - Easy access both by row and column

- Node design (head field will not be shown later)

Handler

#row	#col	#terms
down		right
ishead		

Terms

row	col	val
down		right
ishead		

Row/Col Head

next		
down		right
ishead		

ishead = false

ishead = true

Linked Sparse Matrix

H0 **H1** **H2** **H3** **H4**

5	4	6

H0

0	0	2

H1

1	0	4

1	3	3

H2

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

H3

| 3	0	8

3	3	1

H4

| 4	2	6

70
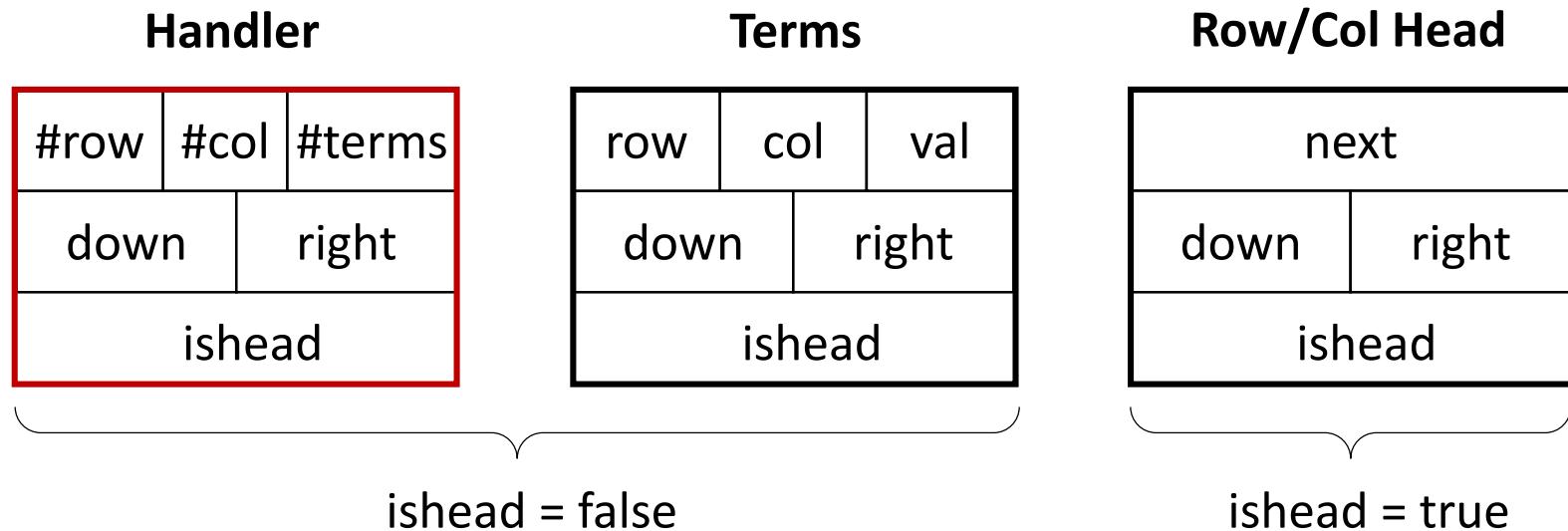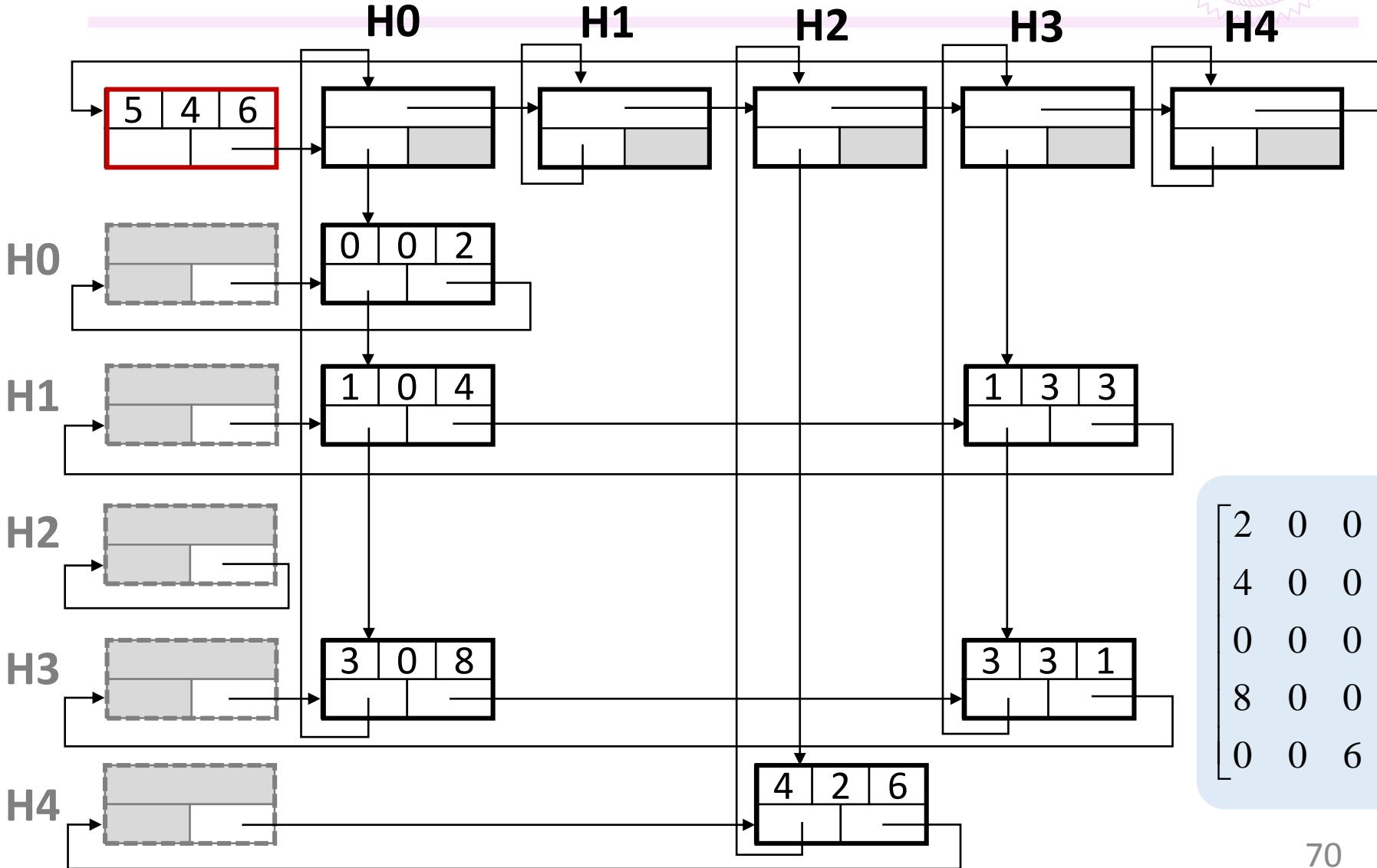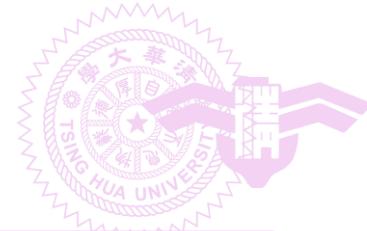
Figure Explanations
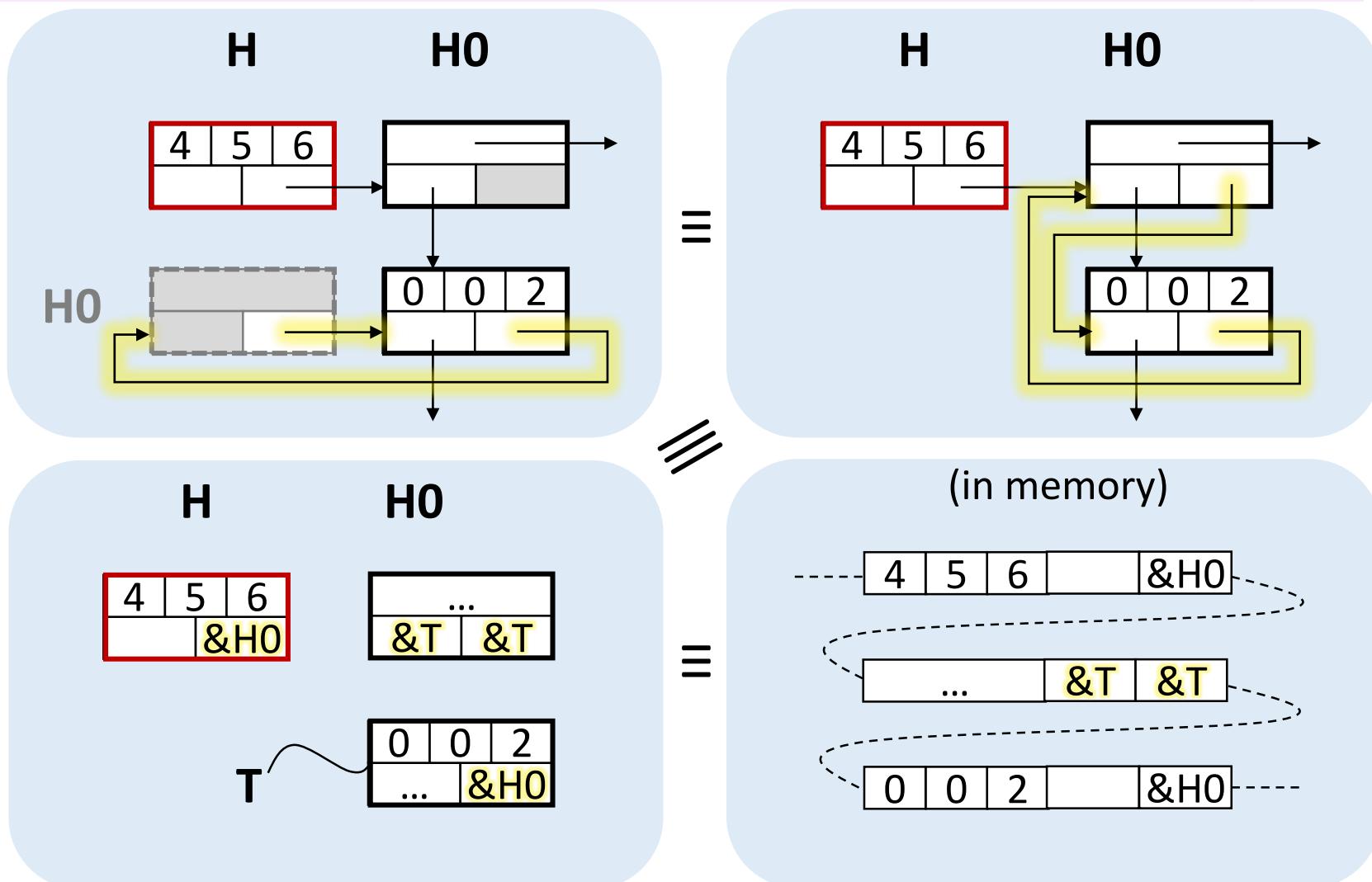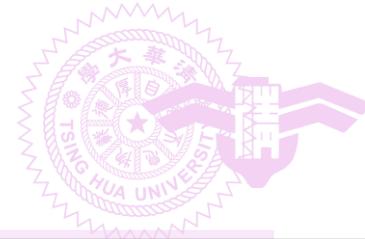
Sparse Matrix Definition

```cpp
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 {                      // union allow multiple objects to
        MatrixNode *next;        // overlap with one another.
        Triple triple;           // Programmers are responsible to
    };                           // take care the use of them.
    MatrixNode(bool, Triple*); // constructor
}
```

Sparse Matrix Definition

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

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

Sparse Matrix Input

- Overloading the >> operator makes sparse matrix more like built-in types

```
int i, j, k;
cin >> i >> j >> k;
Matrix mi, mj, mk;
cin >> mi >> mj >> mk;
```

inputs

```
4 5 6
0 0 2
1 0 4
1 3 3
3 0 8
3 3 1
4 2 6
```
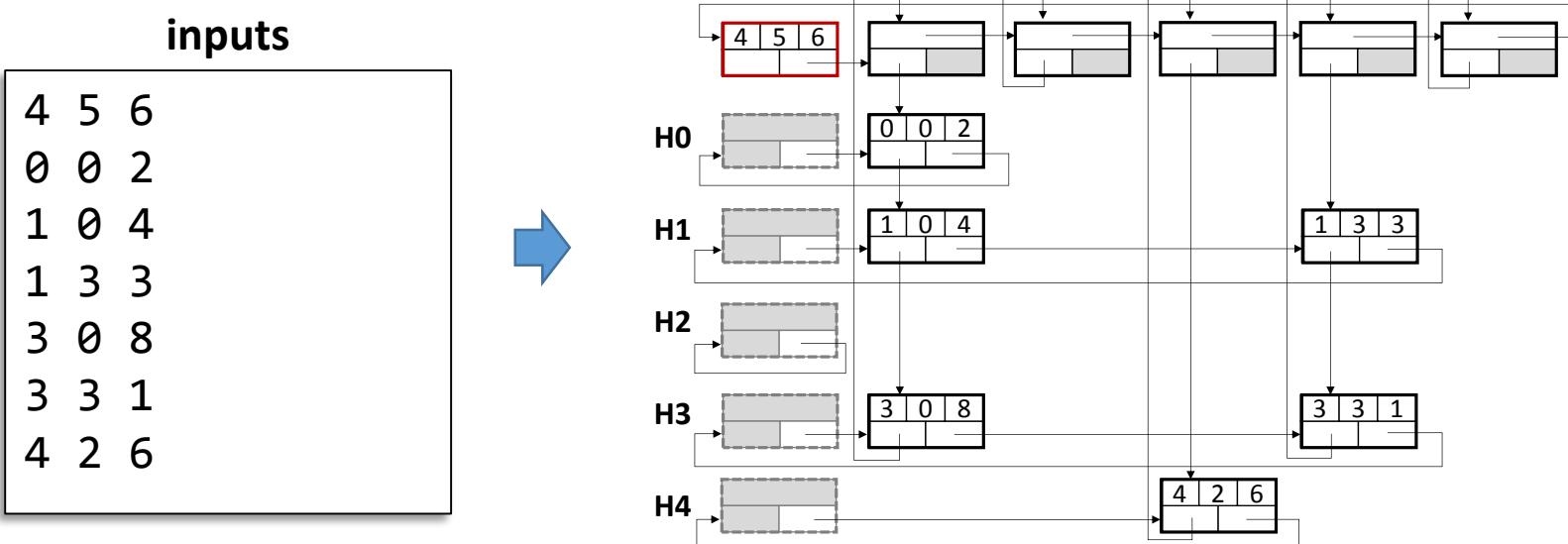
Sparse Matrix Input

```cpp
istream& operator>>(istream& is, Matrix& matrix)
{// Read in a maxtix 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;  // for supporting "cin >> mi >> mj;"
    }
    // at least one row or column
    MatrixNode **head = new MatrixNode* [p];
    for (int i = 0; i < p; i++)
        head[i] = new MatrixNode(true, 0);

    // please continue on the next page
```
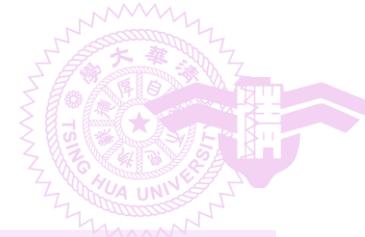
Sparse Matrix Input

```cpp
int currentRow = 0;
int MatrixNode *last = head[0]; // last node in current row
for (i = 0; i < s.value; i++)    // input triples
{
    Triple t;
    is >> t.row >> t.col >> t.value;
    if (t.row > currentRow) { // end of current row
        last->right = head[currentRow]; // close current row
        currentRow = t.row;
        last = head[currentRow];
    } // end of if

    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
// please continue on the next page
```

Sparse Matrix Input

```
        last->right = head[currentRow]; // close last row
        for (i = 0; i < s.col; i++)
            head[i]->next->down = head[i] // close all column lists

        // link the header nodes together
        for (i = 0; i < p; i++)
            head[i]->next = head[i + 1];
        head[p-1]->next = matrix.headnode;
        matrix.headnode->right = head[0];

        delete [] head;
        return is;
}
```
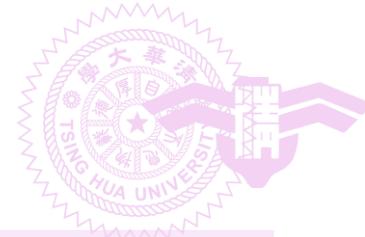
Sparse Matrix Deletion

```
Matrix::~Matrix()
{// Return all nodes to the av list, which is a chain linked
 // via the right field.
 // av is a static variable pointing to the first of the av list.
    if (!headnode )
        return; // no nodes to delete
    MatrixNode *x = headnode->right;

    headnode->right = av;
    av = headnode; // return headnode

    while (x != headnode) { // return nodes by rows
        MatrixNode *y = x->right;
        x->right = av;
        av = y;
        x = x->next; // next row
    }
    headnode = 0;
}
```

Outline

- 4.1-4.3 Basic singly linked lists and chains

- 4.4-4.5 Circular lists

- 4.6-4.9 Linked stacks, queues, polynomials, equivalence classes, and sparse matrices

- **4.10 Doubly linked lists**

- 4.11 Generalized lists

Doubly Linking vs. Singly Linking

- Doubly linked
  - Each node contain pointers to the both direction
 - Traversal can be performed in both direction

left   data   right
(prev)   (next)
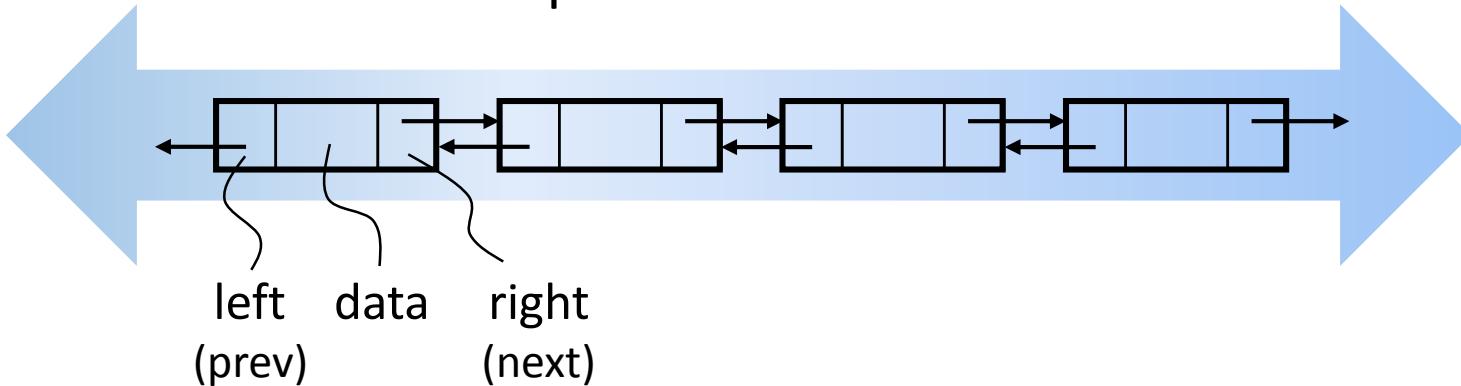
- Singly linked
  - Traversal can only be performed in one direction
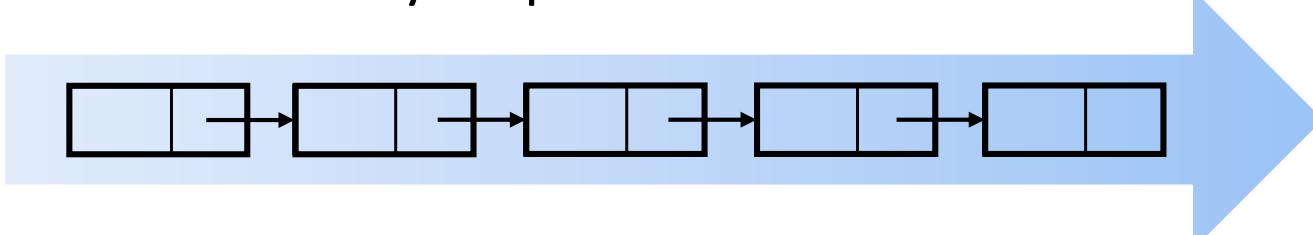
Circular, Doubly Linked Lists with Header

- Empty list

head

header

- Non-empty list

head

header
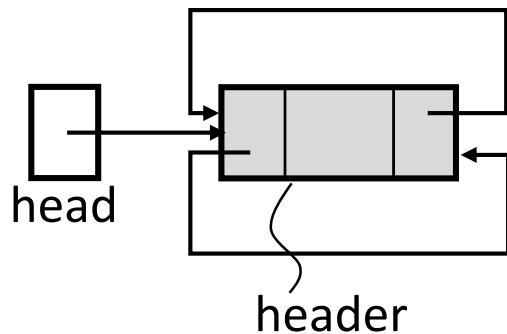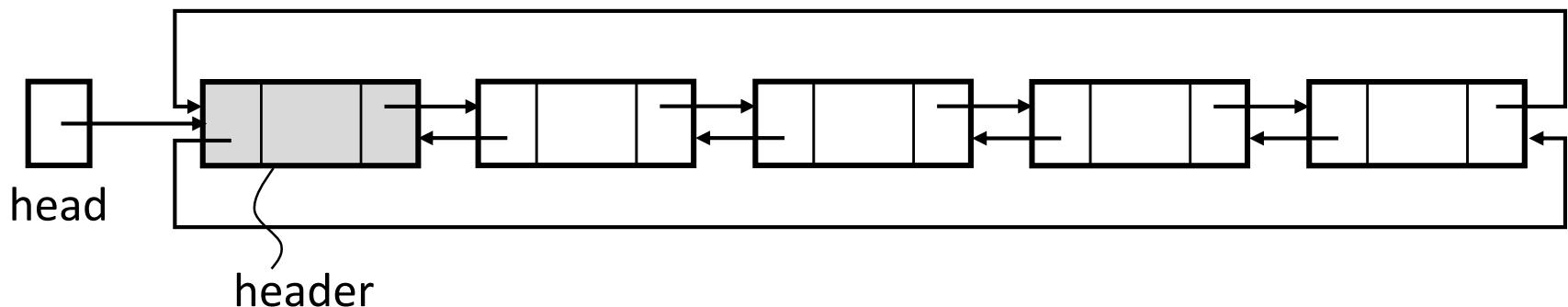
Doubly Linked List Definition

```cpp
class DblList;  //forward declaration

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

class DblList {
public:
  // List manipuation operations
  .
  .
private:
    DblListNode *head;  // points to header node
};
```

> Please note the syntax for declaring two pointers a line

Insertion Operation

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


head

x   p
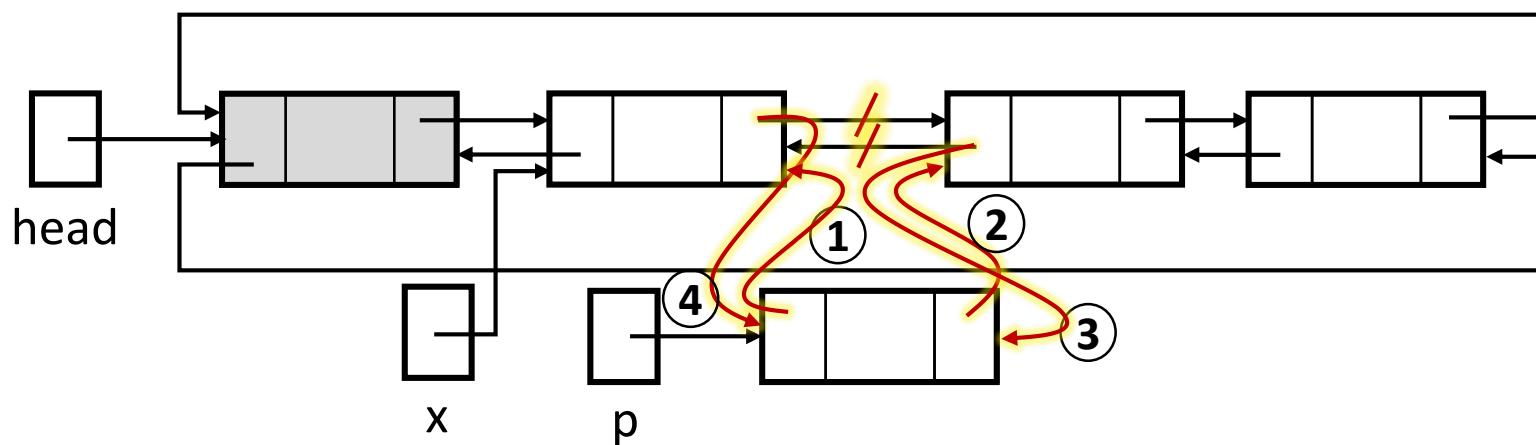
Deletion Operation

```
void DblList :: Delete(DblListNode *x)
{
  if (x == head)
    throw "Deletion of header node not permitted";
  else {
    x->left->right = x->right;  ①
    x->right->left = x->left;   ②
    delete x;
  }
}
```
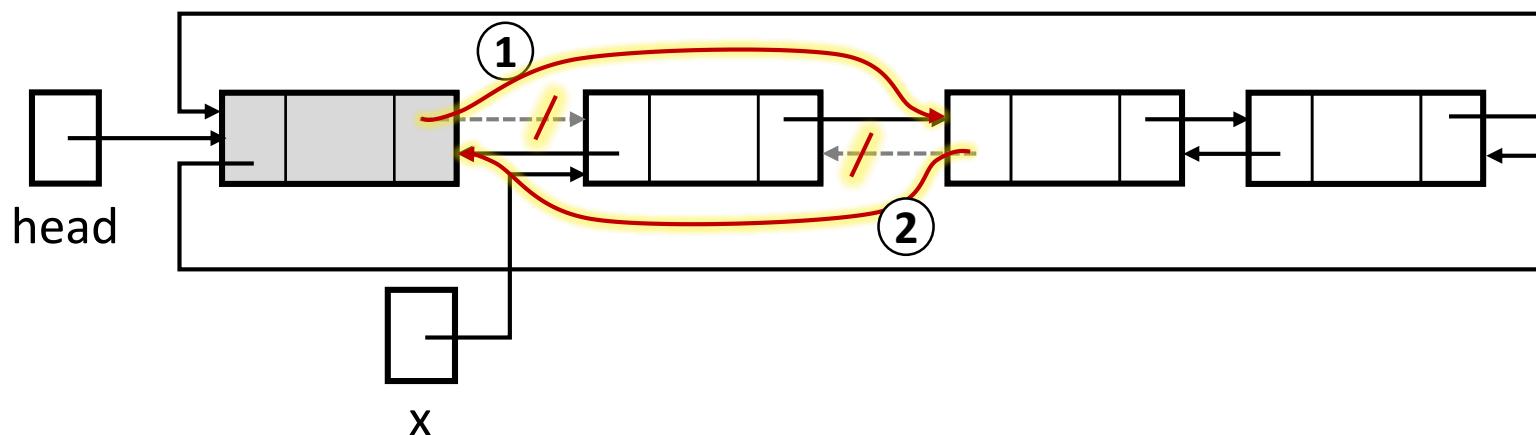
Outline

- 4.1-4.3 Basic singly linked lists and chains

- 4.4-4.5 Circular lists

- 4.6-4.9 Linked stacks, queues, polynomials, equivalence classes, and sparse matrices

- 4.10 Doubly linked lists

- **4.11 Generalized lists**

Generalized Lists

- Generalized list
  - Finite sequence of n≥0 elements, $(a_0, \ldots, a_n)$, where a_i is either an atom or a list

- Head
  - a_0

- Tail
  - $(a_1, \ldots, a_n)$

Exampling Generalized Lists

- **A** = ()
  - Empty (also referred to as null) list
  - Length is zero
- **B** = (a, (b, c))
  - Length is two
  - First element is the atom a
  - Second element is the list (b, c)
- **C** = (**B**, **B**, ())
  - Length is three
  - First two elements are the list B
  - Third element is the empty list
- **D** = (a, **D**)
  - Recursive definition
  - Corresponds to the infinite list (a, (a, (a, …)))

Exampling Generalized Lists

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

- head(**B**) = a
- tail(**B**) = ((b, c))
- head(tail(**B**)) = (b, c)
- tail(tail(**B**)) = ()

- head(C) = **B**
- tail(C) = (**B**, ())

Generalized-List Polynomials

- Example
  - $x^{10}y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz$

- Factorizing out variables
  - $x^{10}y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz$

  $= (x^{10}y^3 + 2x^8y^3 + 3x^8y^2)z^2 + (x^4y^4 + 6x3y^4 + 2y)z$

  $= \underline{((x^{10} + 2x^8)y^3 + 3x^8y^2)z^2} + \underline{((x^4 + 6x^3)y^4 + 2y)z}$

  $=$

  $\underline{C(x, y)}z^2 + \underline{D(x, y)}z$

  $\underline{E(x)y^3 + F(x)y^2} \qquad \underline{G(x)y^4 + G(x)y}$
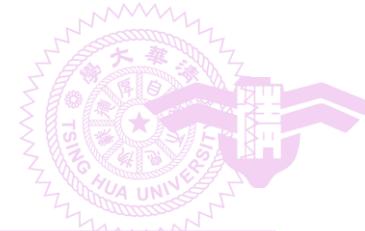
Generalized-List Polynomials

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

- Three types of information
  - Factored-out variable   e.g., 'z'
  - Exponent of the factored variable and a link to the associated sub-polynomial   e.g., 2
                                                                $(x^{10} + 2x^8)y^3 + 3x^8y^2$
  - Coefficient of each term   e.g., 1

- Node structure for storing the information

type of the node
variable name / link to sub-polynomial / coefficient
exponent (optional)
link to the next node

Generalized-List Polynomials

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

$3x^8$

| x | 0 | → | 3 | 8 | 0 |

$x^{10}+2x^8$

| x | 0 | → | 1 | 10 | → | 2 | 8 | 0 |

Generalized-List Polynomials
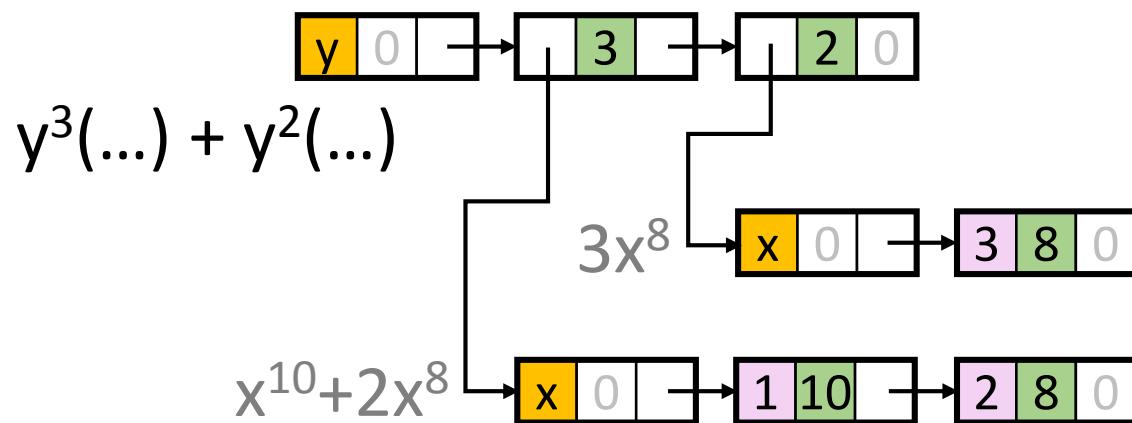
- $((x^{10} + 2x^8)y^3 + 3x^8y^2)z^2 + ((x^4 + 6x^3)y^4 + 2y)z$

$y^3(\ldots) + y^2(\ldots)$

$3x^8$

$x^{10}+2x^8$

Generalized-List Polynomials

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

$y^3(\ldots) + y^2(\ldots)$

$3x^8$

$x^{10} + 2x^8$

Generalized-List Polynomials

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

$z^2(\ldots) +$

$z(\ldots)$

$y^3(\ldots) + y^2(\ldots)$

$3x^8$

$x^{10} + 2x^8$

Node Definition

```cpp
enum NodeType{var, ptr, no}; // var=0, ptr=1, no=2
class PolyNode
{
    NodeType type;

        char name;
        PolyNode* sub;
        int coef;

    int exp;
    PolyNode* next; // link to next node
}
```

type of the node
variable name
link to sub-polynomial
coefficient
exponent (optional)
link to the next node
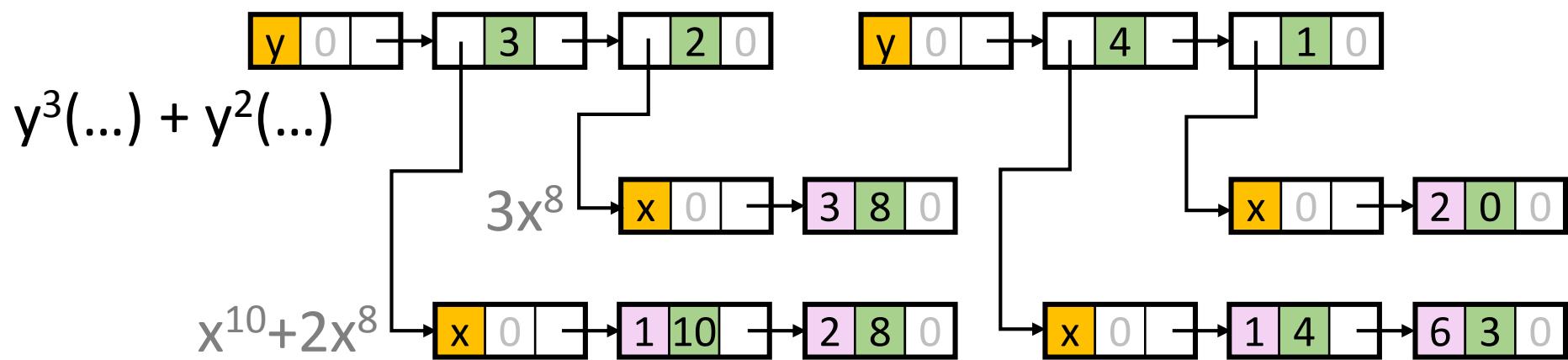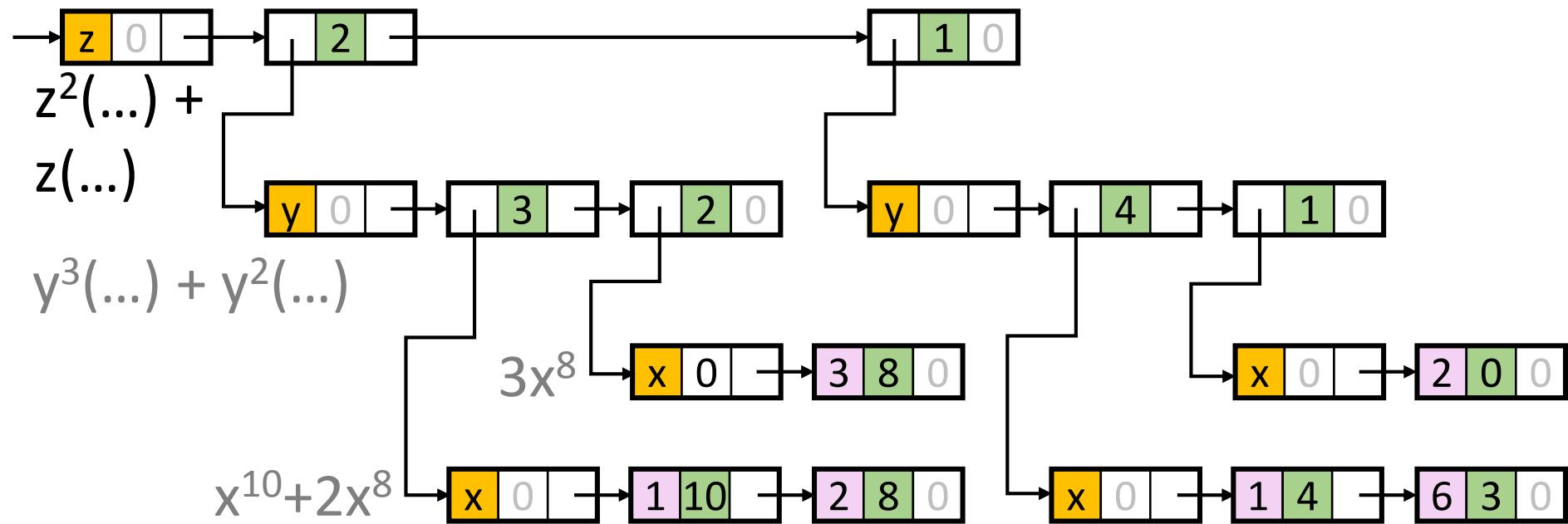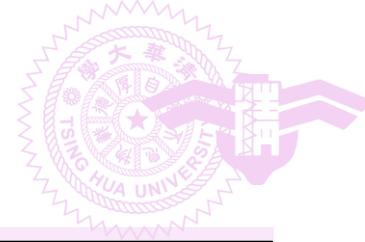
Node Definition

```
enum NodeType{var, ptr, no}; // var=0, ptr=1, no=2
class PolyNode
{
    NodeType type;
    union{
        char name;
        PolyNode* sub;                        // three variables
        int coef;                             overlap in memory
    }
    int exp;
    PolyNode* next; // link to next node
}
```

type of the node
name / sub / coef
exponent (optional)
link to the next node

reduced node size

Generalized Lists

- Node structure

| tag (F/T)	data / down	next

- **A** = ()
- **A'** = (())
- **B** = (a, (b, c))
- **C** = (**B**, **B**, ())
- **D** = (a, **D**)
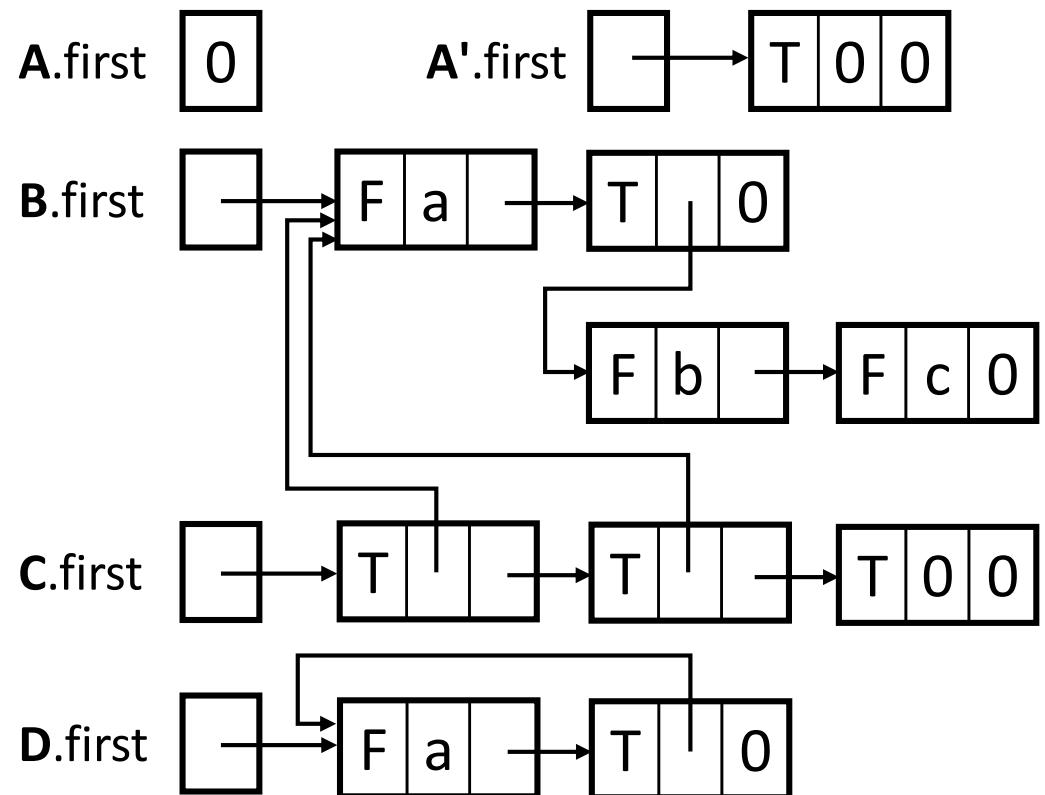
Generalized List

```
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;
}
```

Copying a List

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

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

Testing Equality for Two Lists

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

// Workhorse
bool Equal(GenListNode<T>*s, GenListNode<T>*t)
{
  if ((!s) && (!t)) return true;
  if (s && t && (s->tag == t->tag))
    if (s->tag)
      return Equal(s->down, t->down)&&Equal(s->next, t->next);
    else
      return (s->data == t->data)&&Equal(s->next, t->next);
  return false;
}
```

Computing the List Depth

```
// Driver                          // Depth ≡ max degree of nested sublists
template <class T>
int GenList<T>::Depth()
{ // Compute the depth of a non-recursive list
  return Depth(first);
}
// Workhorse
template <class T>
int GenList::Depth(GenListNode<T>* s)
{
  if (!s) return 0;   // empty list
  GenListNode<T>* current = s;
  int m=0;   // initial minimum value
  while (current) {
      if (current->tag) m = max(m, Depth(current->down));
      current = current->next;
  } // m is the max depth among sublists
  return m+1;
}
```

Computing the List Depth
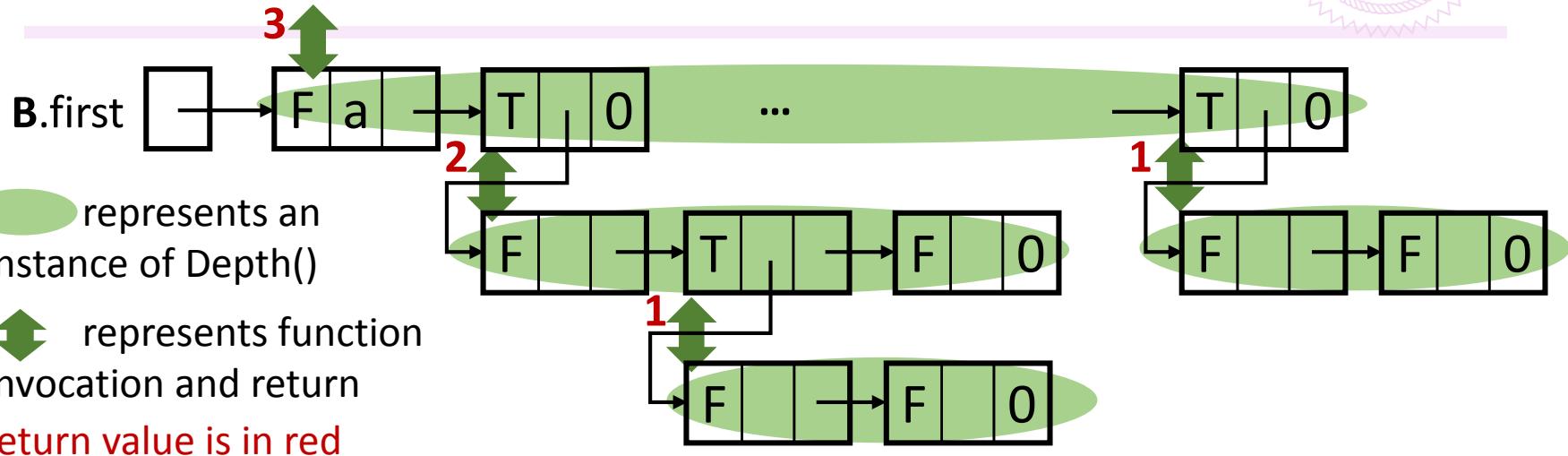
represents an instance of Depth()

represents function invocation and return

return value is in red

```cpp
template <class T>
int GenList::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;
  } // m is the max depth among sublists (m=0 if no sublists exist)
  return m+1;  // increments m for the current list
}
```
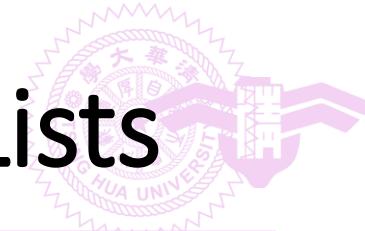
Adding a Node or Deleting a Lists

- Design issues
  - Adding or deleting node at the front of a list can affect the *down* fields of other lists
    - e.g., adding an 'x' in front of 'a' in **B**
    - We normally do not know these affected fields
  - Determining whether list nodes need to be freed
    - e.g., the node 'a' needs to be kept (if other lists require it) even if B.first no longer points to it
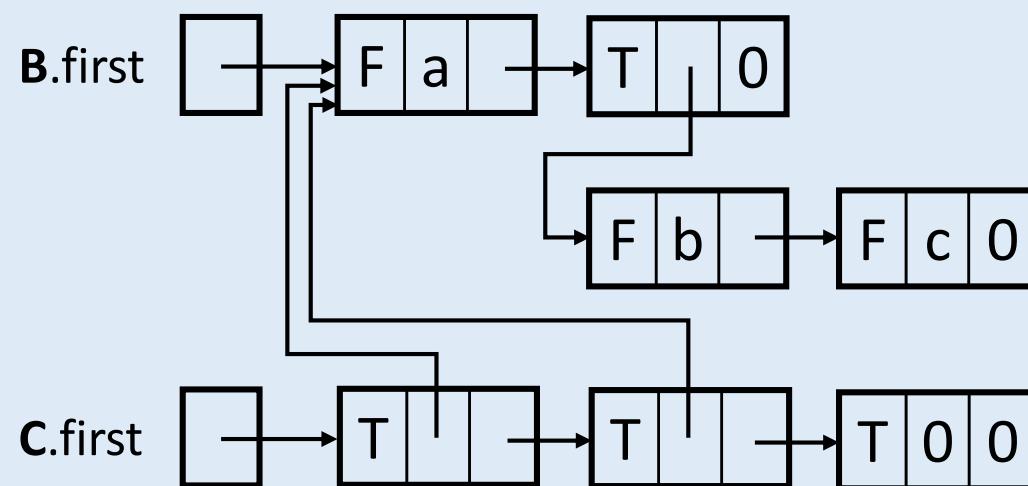
Headers and Reference Counts

- Each list/sublist equips a header
  - Adding and deleting node at the front of the list does not affect other lists

- Headers serve the reference point of a list

- Header tracks the reference count, the number of pointers pointing to the lists
  - List nodes with no reference to them require being freed
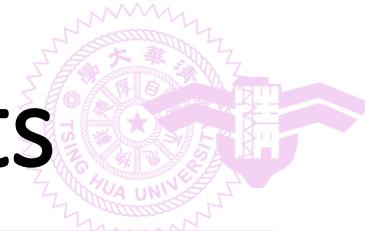
Headers and Reference Counts

- **A** = ()
- **B** = (a, (b, c))
- **C** = (**B**, **B**, ())
- **D** = (a, **D**)
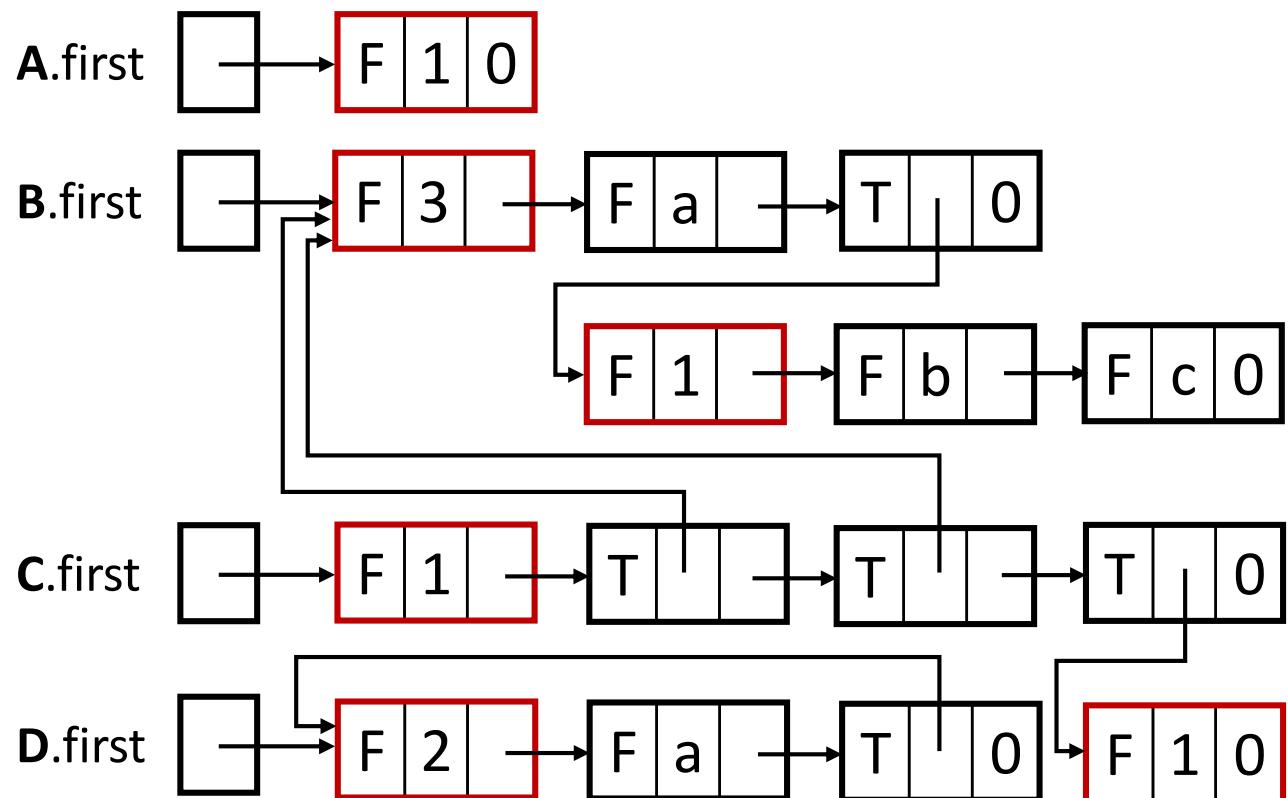
Deleting Generalized Lists

```
// Driver
template <class T>
GenList<T>::~GenList()  // Destructor
{ // 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;
        while (y->next) // y traverses top level of x
        { y = y->next; if (y has down link) Delete(y->down); }
        y->next = av; // attach top-level nodes to av list
        av = x;
    }
}
```