香港中文大學（深圳）
The Chinese University of Hong Kong

CSC3100 Data Structures
Lecture 6: List

Yixiang Fang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

List

- A general list of the form: $A_1, A_2, A_3, \ldots, A_N$
  - We say that the size of this list is N
  - For any list except the empty list, we say:
    - A_{i+1} follows (succeeds) A_i ($i < N$)
    - A_{i-1} precedes A_i ($i > 1$)
    - The first element of the list is A_1
    - The last element is A_N
    - We will not define the predecessor of A_1, or the successor of A_N.
    - The position of element A_i is i

List

- Some popular operations on List are:
  - printList
  - makeEmpty
  - find
    - return the position of the first occurrence of a key, e.g., given the list: 34, 12, 52, 16, 12, Find(52) returns 3.
  - insert
    - insert some key at some position, e.g., Insert(X, 3).
  - Delete
    - delete some key from some position, e.g., Delete(52).
  - next & previous (optional)

Array is a list

- PrintList and Find implemented in O(N)
- Require to estimate of the maximum size for memory allocation (often high overestimate, wasting space!)
- Insertion and Deletion are however expensive
  - Why? What is the major cost?
    - Answer: shift items

Linked List

- Consists of a series of nodes (Node class)
- A singly linked list, each node is composed of data and a reference
- Not necessarily adjacent in memory
- Flexible on element insertion and deletion

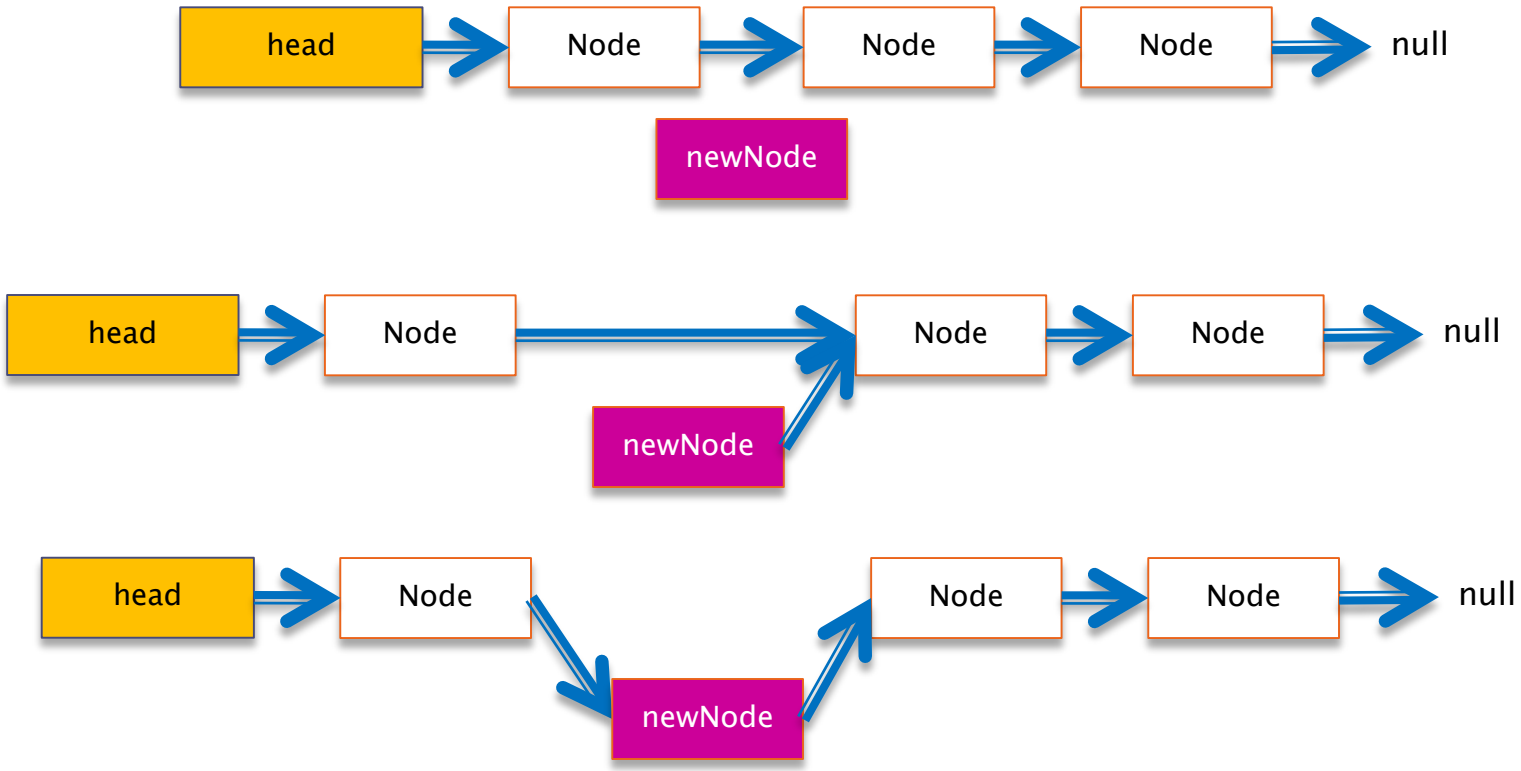
```
class Node {
        int element;
        Node next;
}
```

```
// constructor
public Node(int x) {
        element = x;
        next = null;
}
```

Note:
"*head*" here is a node that does NOT store data

Linked List :: Insert

- Add a new element to the list



Linked List :: Insert

```
void insert(int x, int p) {
        Node tmpNode = new Node(x);
        Node prevNode = head;

        for (int i=0; i<p-1; i++) {
                if (prevNode.next == null)
                        break;
                prevNode = prevNode.next;
        }

        tmpNode.next = prevNode.next;
        prevNode.next = tmpNode;
}
```

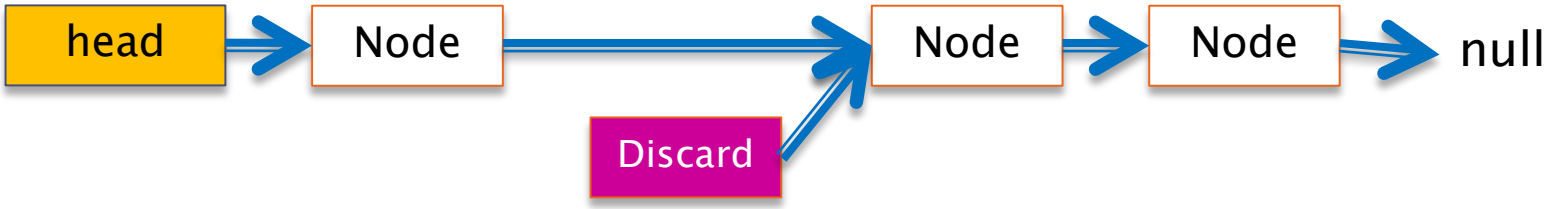
- *tmpNode* is a new node that contains x
- *prevNode* will be used for finding the previous node of *tmpNode*

Moves to the end of the list if p is larger than the size of list

Link the new node

Linked List :: Delete

- Delete a node from the list



Linked List :: Delete

```
void delete(int p) {
        Node prevNode = head;

        for(int i=0; i<p-1; i++)
                if (prevNode.next == null)
                        break;
                else
                        prevNode = prevNode.next;

        Node targetNode = prevNode.next;

        if (targetNode != null)
                prevNode.next = targetNode.next;

        free(targetNode)
}
```

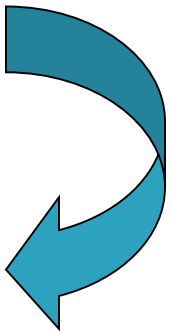
Go to the specific position

Bypass the target node

List: Example1

- Single-variable Polynomials

$$F(X) = \sum_{i=0}^{N} A_i X^i$$



By array implementation

```
class Polynomial {
        int coeffArray[MaxDegree + 1];
        int highPower;
}
```

Example1

- Initialize a polynomial

```
void zeroPolynomial(Polynomial poly){
        int j;
        for (j = 0;  j <= MaxDegree; j++)
                poly.coeffArray[j] = 0;
        poly.highPower = 0;
}
```

Example1

- Add two polynomials

```
void addPolynomial(Polynomial poly1, Polynomial poly2,
    Polynomial polySum) {
    zeroPolynomial(polySum);
    polySum.highPower = Math.max(poly1.highPower, poly2.highPower);
    for (int i = polySum.highPower; i>=0; i--)
          polySum.coeffArray[i] = poly1.coeffArray [i] +
                poly2.coeffArray[i];
}
```

Example1

- Multiply two polynomials

```
void multPolynomial(Polynomial poly1, Polynomial poly2, Polynomial
    polyProd) {
    zeroPolynomial(polyProd);
    polyProd.highPower = poly1.highPower+ poly2.highPower;
    If (polyProd.highPower > MaxDegree)
          Error("Exceed array size");
    else
          for (int i =0; i<=poly1.highPower; i++)
            for (int j =0; j<=poly2.highPower; j++)
              polyProd.coeffArray[i+j] += poly1.coeffArray [i] *
                                                  poly2.coeffArray[j];
}
```

Example1

- Good or bad?
- Consider the following situation
  $P_1(X) = 10\,X^{1000} + 5X^{14} + 1$
  $P_2(X) = 3X^{1990} - 2X^{1492} + 11X + 5$
- Most of the time is spent multiplying zeros

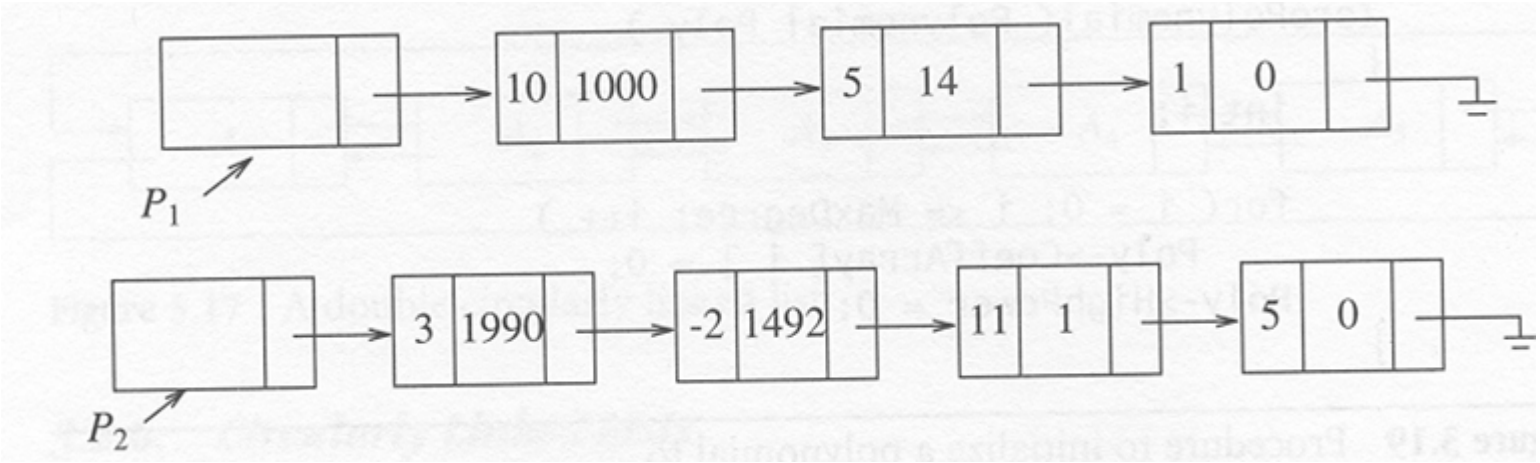
Example1

- Multiply two polynomials: better structure

```
class Node {
        int     coefficient;
        int     exponent;
        Node next;
}
```

Example1

- Linked list representation of the previous structure



Example2

- A university with 40,000 students and 2,500 subjects needs to generate 2 reports:
  1. Lists of students for each class
  2. Lists of classes that each student registered

  Implementation: construct 2D array (40Kx2.5K matrix) = 100M entries

  if each student takes 3 subjects => only 120K entries (~0.1% of 100M) => waste of resources

Example2

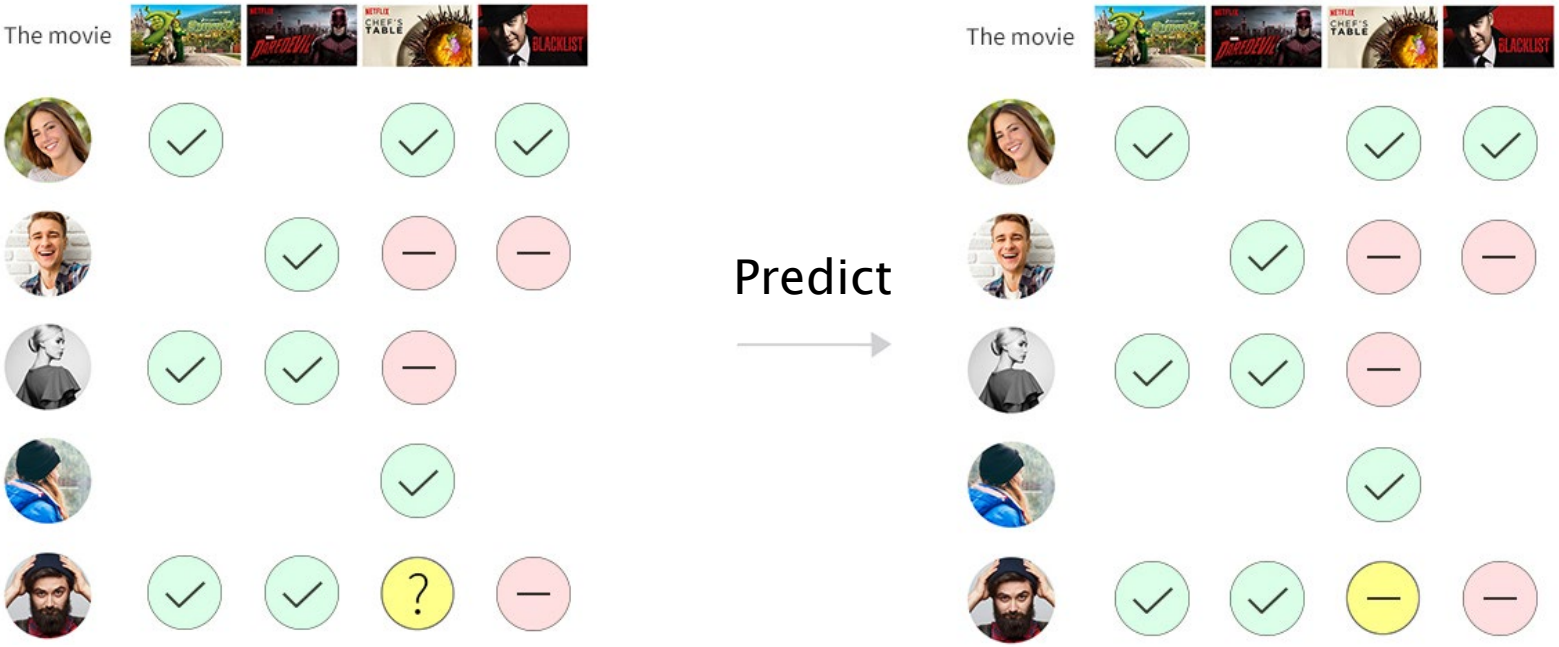
- First solution: triplet representation (minimum space)

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

Rows	Columns	Values
0	4	9
1	1	8
2	0	4
2	2	2
3	5	5
4	2	2

Example3

- Recommendation problem



- In a lot of applications, such as music app, taobao, meituan, seems everywhere in social network

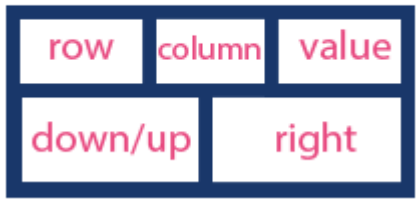
Example3

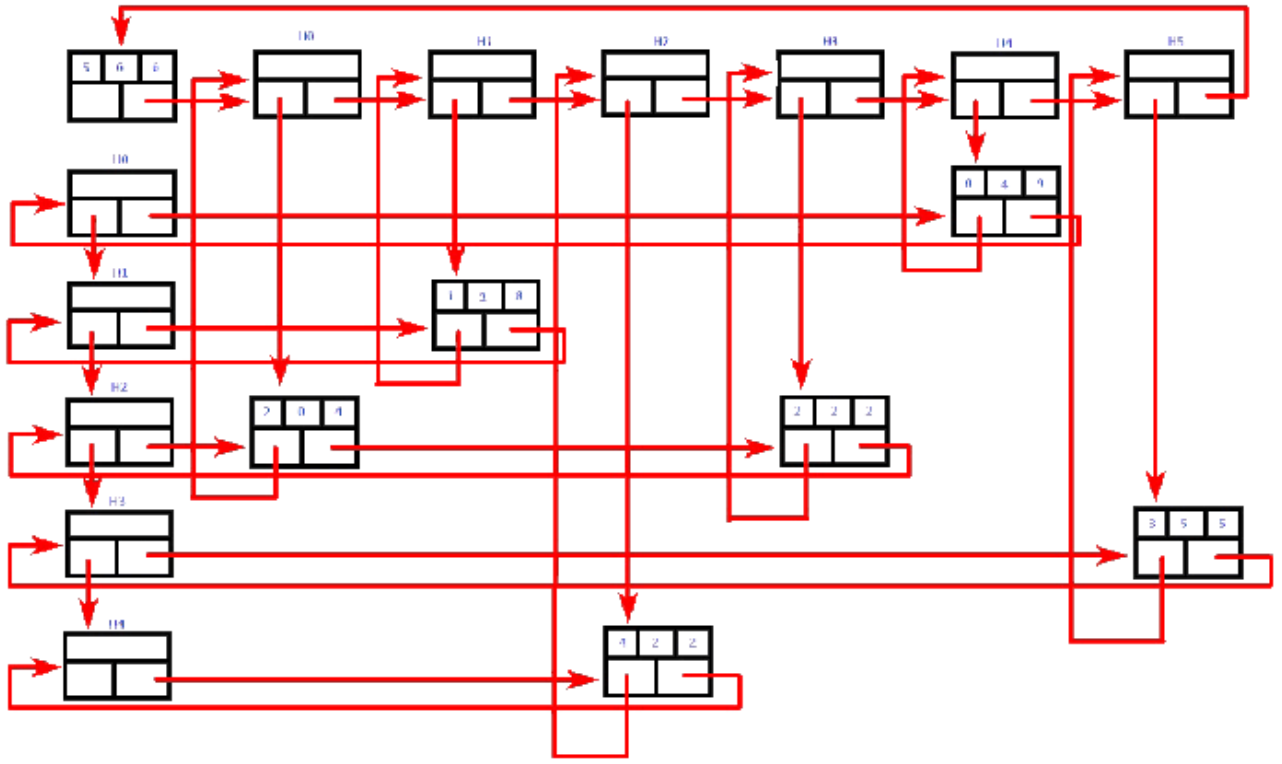
- Suppose we have 10 million people and tens of movies or products
  - 10,000,000*10,000 = 100G items, how to save?
- The huge matrix may involve in many computations, such as multiplications, eigen decomposition, etc.
  - Solution: spare data structure based on basic type

Example3

- Second solution: Linked Representation

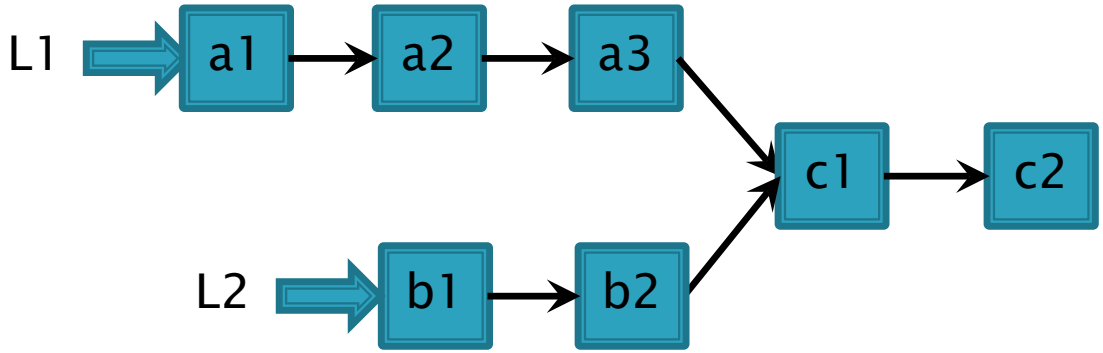
Element Node





Application 1: Intersection of Two Linked List

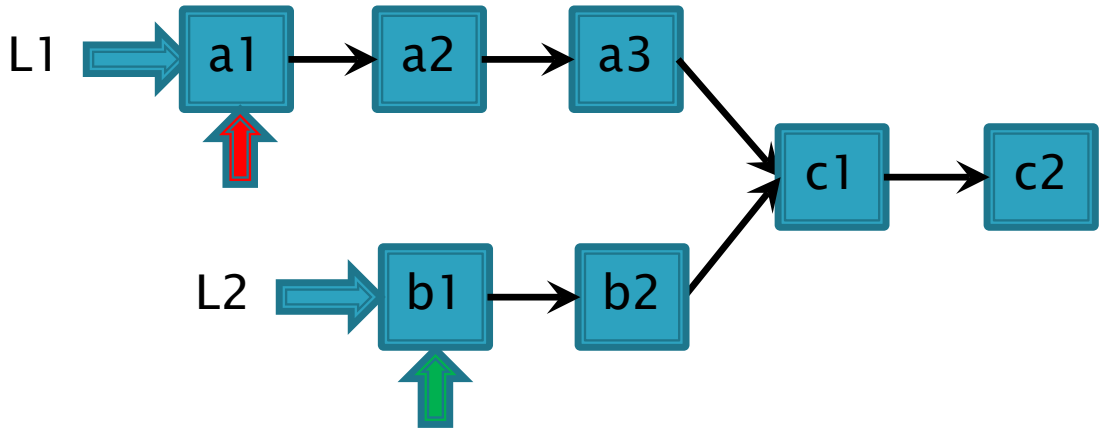
Problem Statement



L1 has M elements while L2 has N elements.

Find the first node where L1 and L2 intersect.

Problem Statement



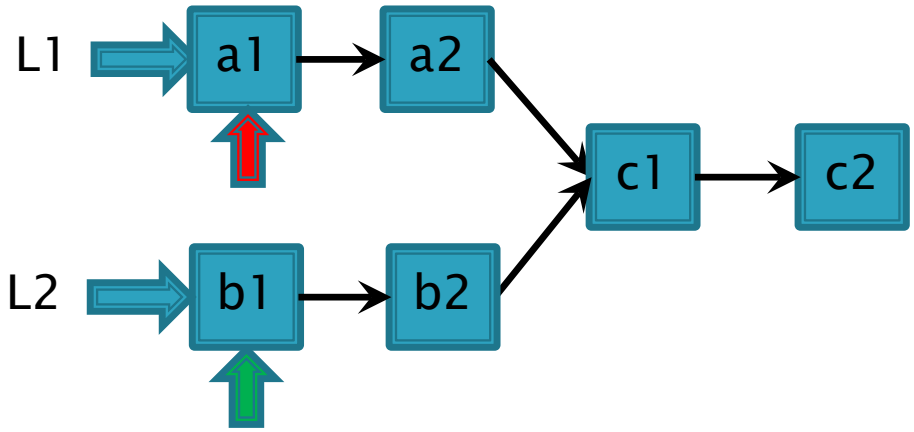
- Use pointer A to traverse L1, and use pointer B to traverse L1
- Compare every possible pair of A and B

Method #1

```
A = L1.head
while A != NULL
        B = L2.head
        while B != NULL
                if A == B
                        return A
                B = B.next
        A = A.next
```

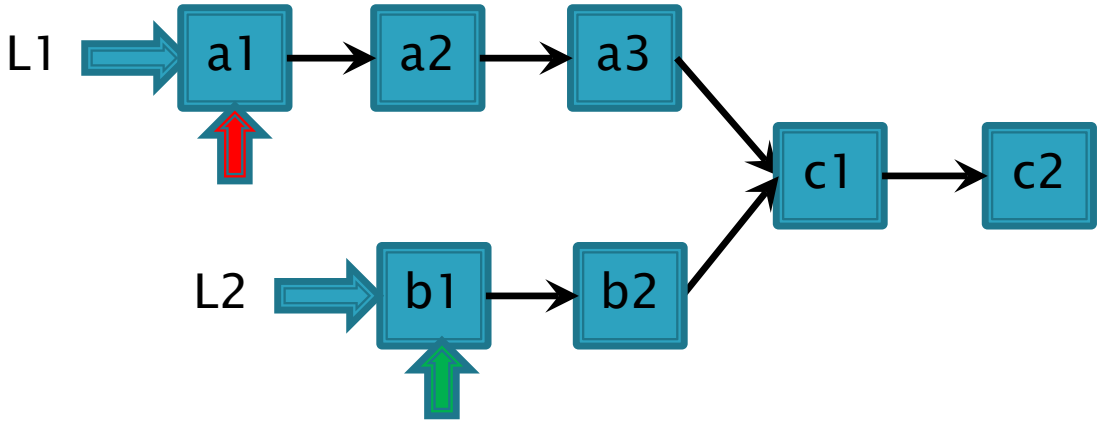
O(MN)

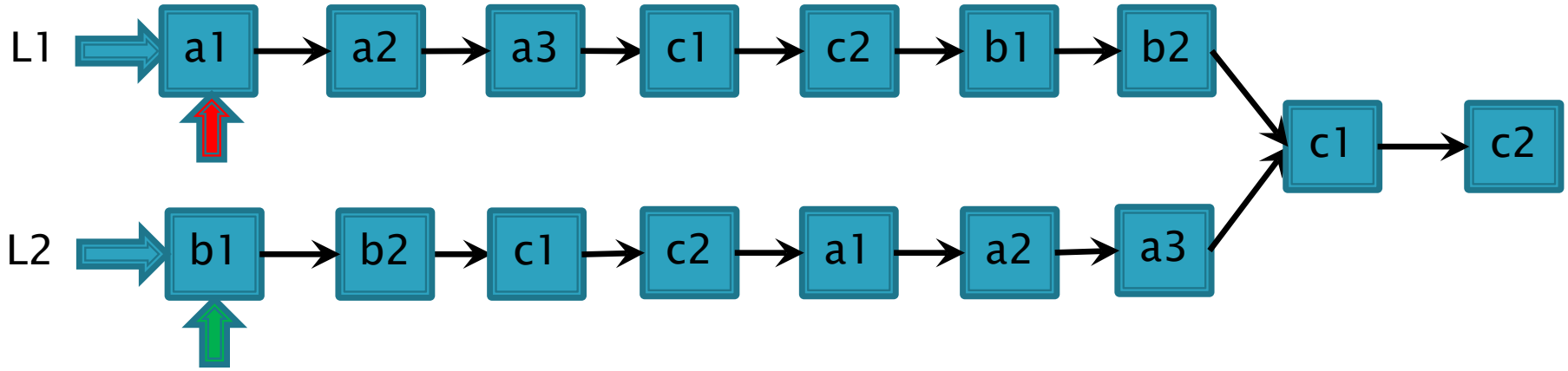
Problem Statement



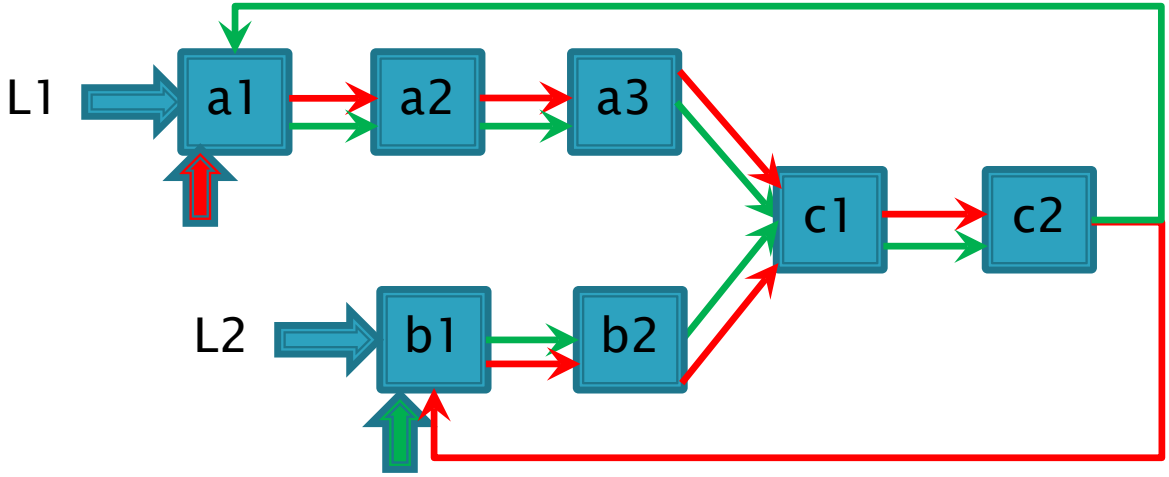
If L1 and L2 have the same length, the problem is easy to solve

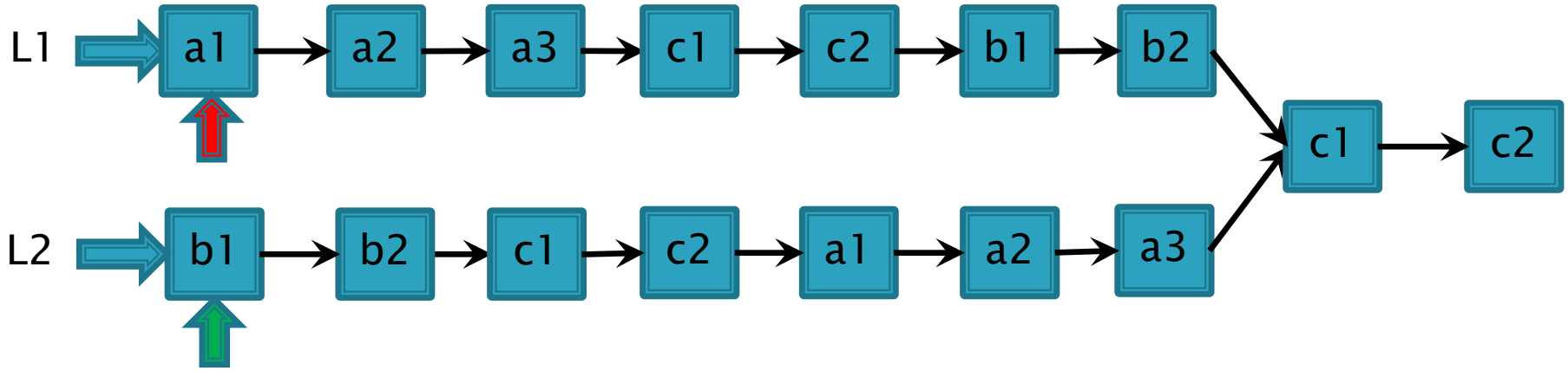
Method #2


L1
a1
a2
a3
c1
c2
L2
b1
b2


L1
a1
a2
a3
c1
c2
b1
b2
L2
b1
b2
c1
c2
a1
a2
a3

Method #2


L1
a1
a2
a3
c1
c2
L2
b1
b2


L1
a1
a2
a3
c1
c2
b1
b2
L2
b1
b2
c1
c2
a1
a2
a3
c1
c2

Method #2

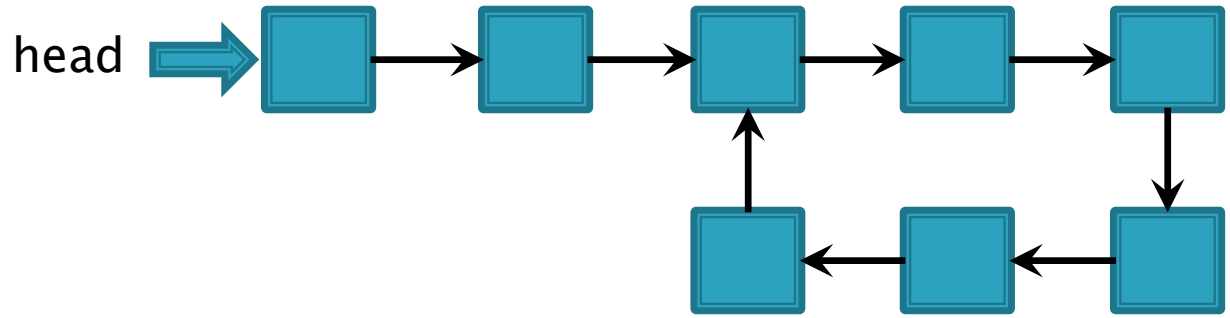
```
A = L1.head
B = L2.head
while TRUE
        if A == B
                return A
        if A.next == NULL
                A = L2.head
        else A = A.next
        if B.next == NULL
                B = L1.head
        else B = B.next
```

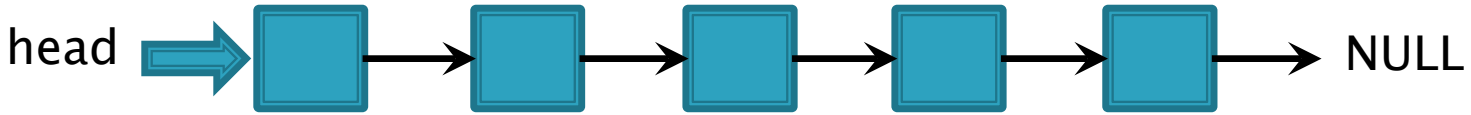
O(M+N)

Application 2: Cyclic or Not?

Problem Statement

Given the head of a singly linked list *L*, decide if *L* has a cycle.





Method #1

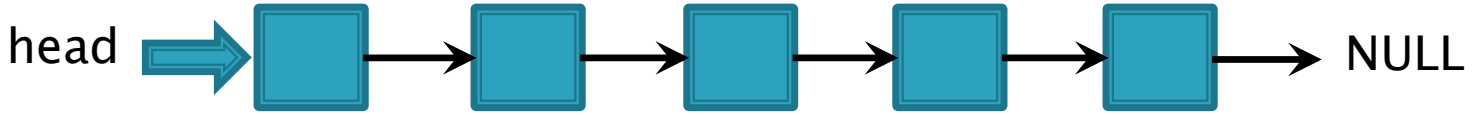
- If *L* is acyclic, we ultimately arrive at NULL by continuously following the next pointer:

```
p = L.head
for i = 1 upto M
        if p == NULL
                return "acyclic"
        else   p = p.next
return "cyclic"
```

M is some big number

- M must be sufficiently large to guarantee correctness, but it is hard to decide M.

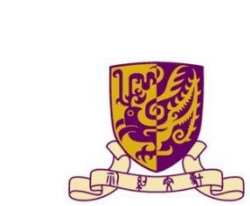
Method #2



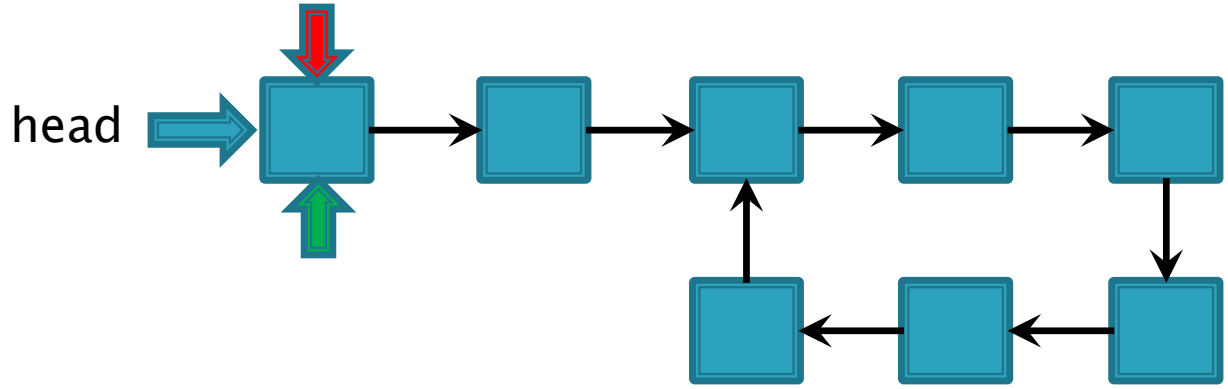
- Store all revisited noded in a new list L':

```
p = L.head
while p != NULL
        if search(L', p) == NULL
                insert(L',p)
                p = p.next
        else return "cyclic"
return "acyclic"
```

- L' and search are expensive; use a Hash table.

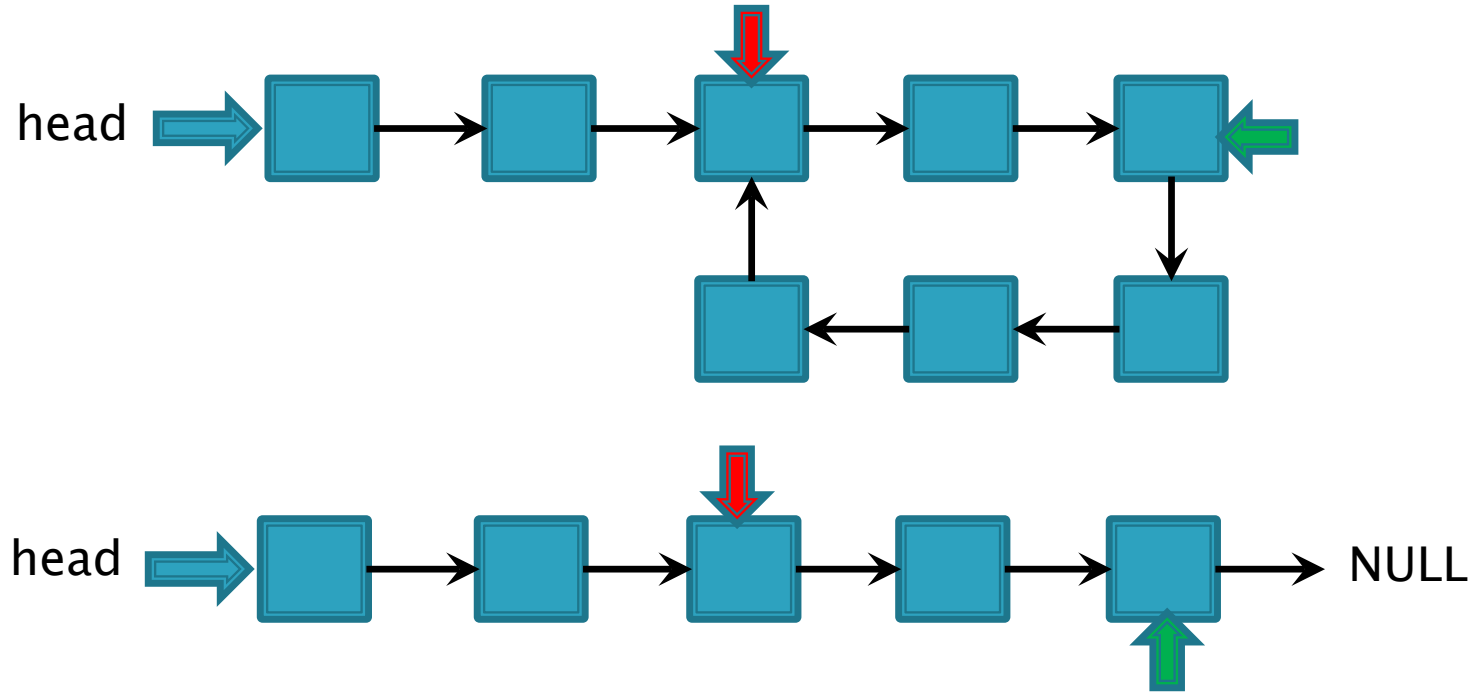


Method #3

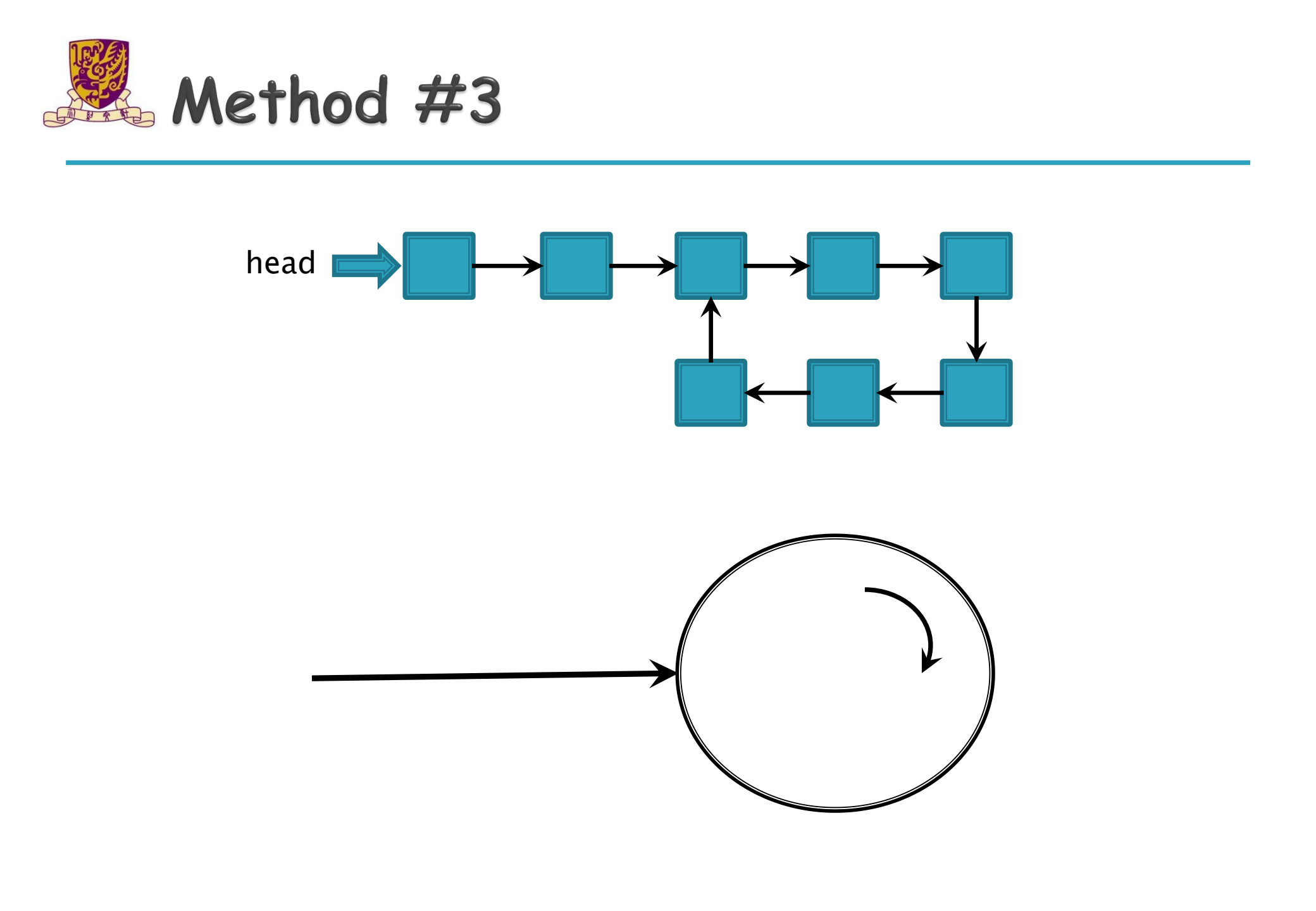


- Use two pointers A and B, both initialized to head
- Every time A=A.next while B=B.next.next

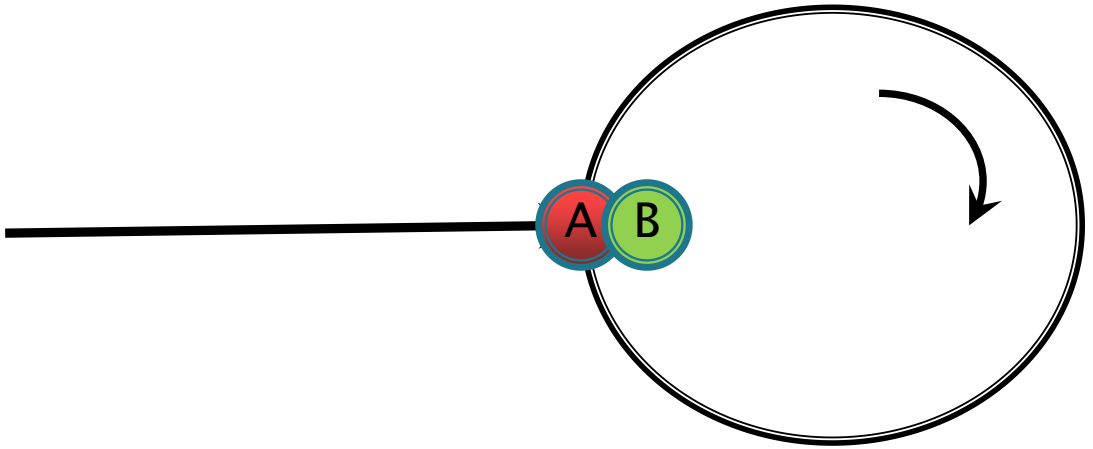
Method #3



- If *L* is acyclic, either B or B.next be NULL
- If *L* is cyclic, B enters the cycle earlier than *A*

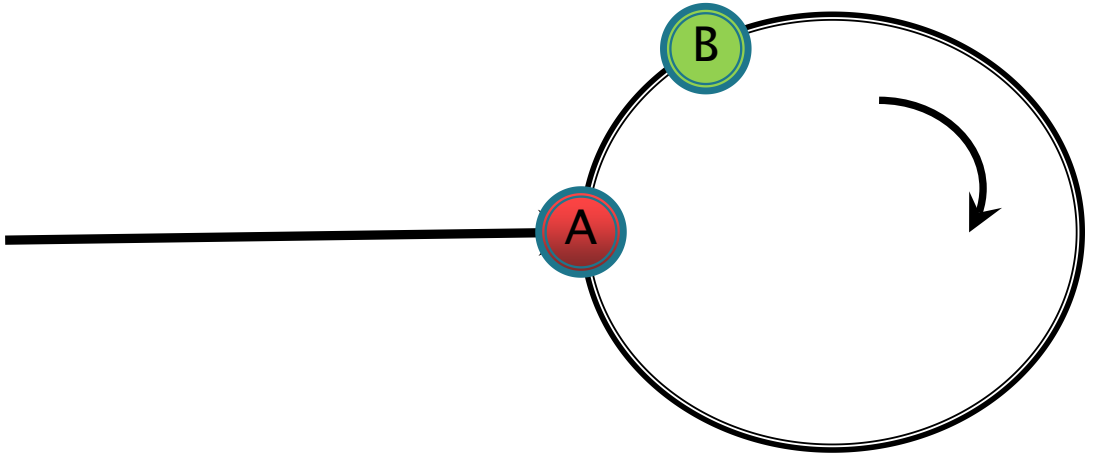

Method #3
head

Method #3



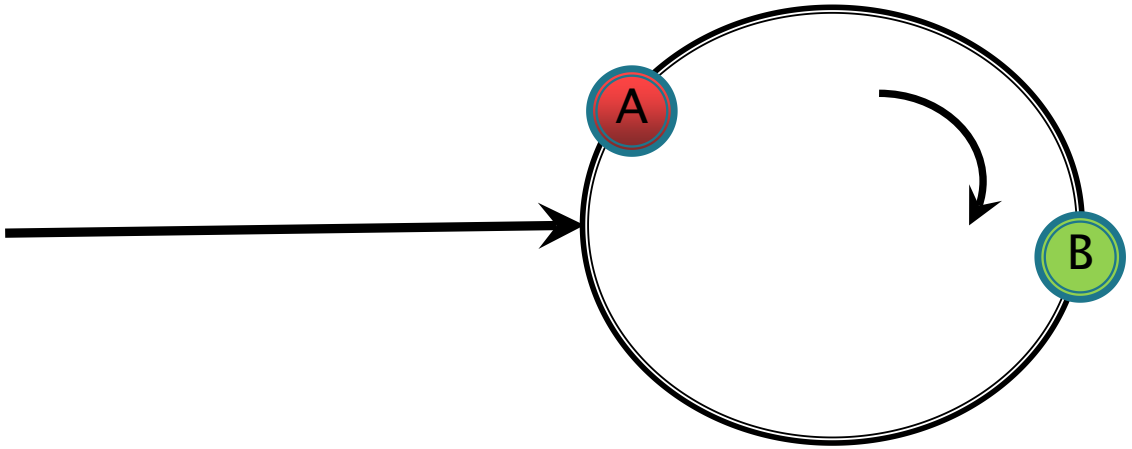
- Case I: B is exactly at entrance when A arrives at the cycle
- So A and B meet at entrance

Method #3



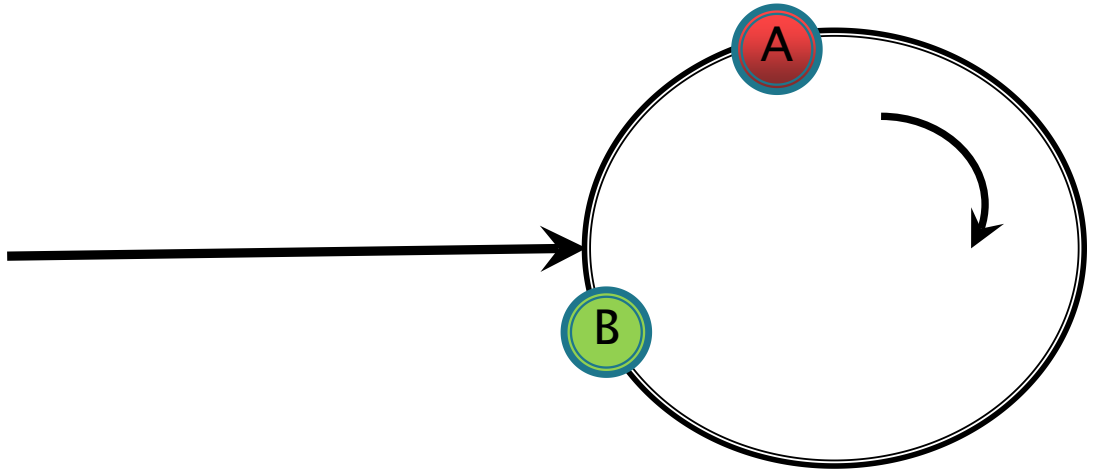
- Case II: B is somewhere else in the cycle when A arrives at entrance

Method #3



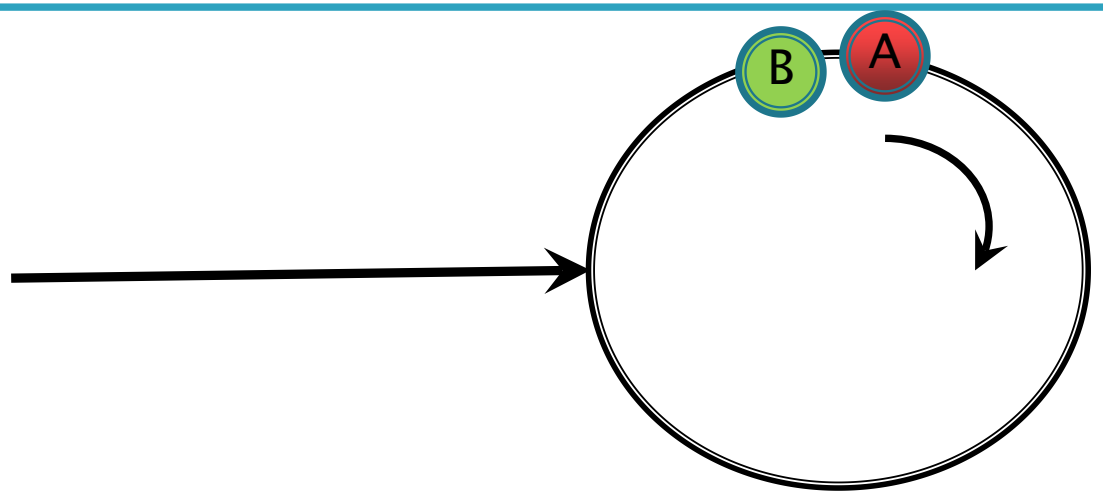
- Since B is moving faster, it must overtake A at a certain point in time

Method #3



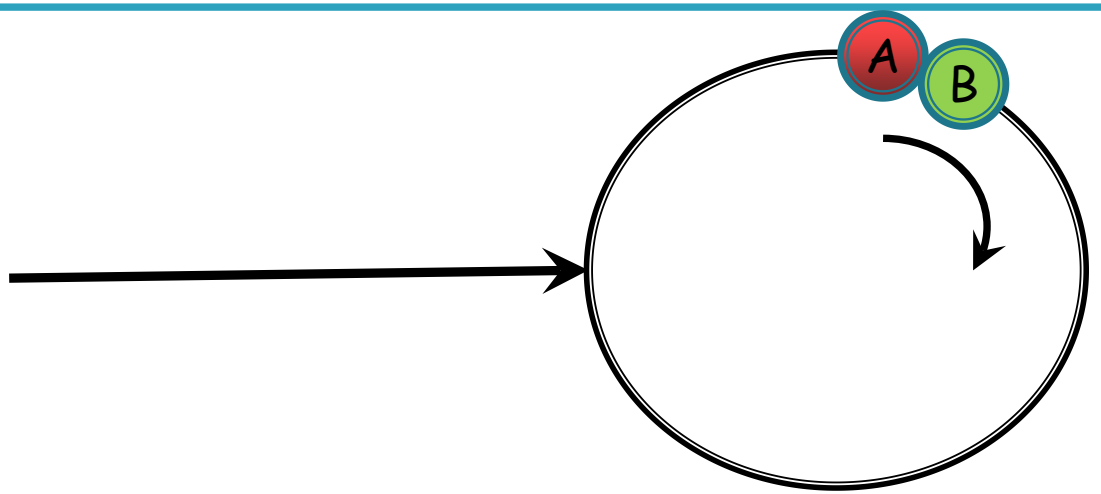
- Since B is moving faster, it must overtake A at a certain point in time

Method #3



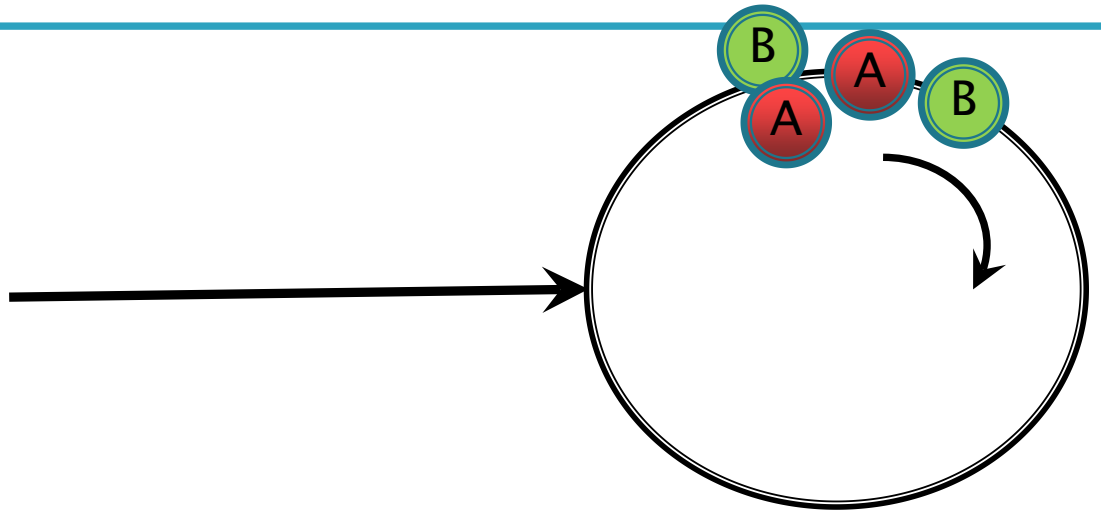
- Since B is moving faster, it must overtake A at a certain point in time

Method #3



- Since B is moving faster, it must overtake A at a certain point in time

Method #3



- And right before B overtakes A, the two nodes meet

Method #3

- Thus, A and B are guaranteed to meet if link is cyclic.

```
A = L.head; B = L.head
while B != NULL and B.next != NULL
        if A == B
                return "cyclic"
        A = A.next
        B = B.next.next
return "acyclic"
```

Recommended reading

- Reading
  - Chapter 10, textbook
- Next lectures
  - Stack and queue