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

CSC3100 Data Structures
Lecture 8: List

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

Outline

- Review of List ADT
- Applications of lists
  - Operations on polynomials
  - Matrix representation
- Exercises
  - Intersection of two linked lists
  - Duplicate letter detection
  - Cycle detection

Application 1: operations on polynomials

- Single-variable polynomials

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

By array implementation

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

Application 1: operations on polynomials

- Initialize a polynomial

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

Application 1: operations on polynomials

- 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];
}
```

Application 1: operations on polynomials

- 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];
}
```

Application 1: operations on polynomials

- 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 on multiplying zeros

Application 1: operations on polynomials

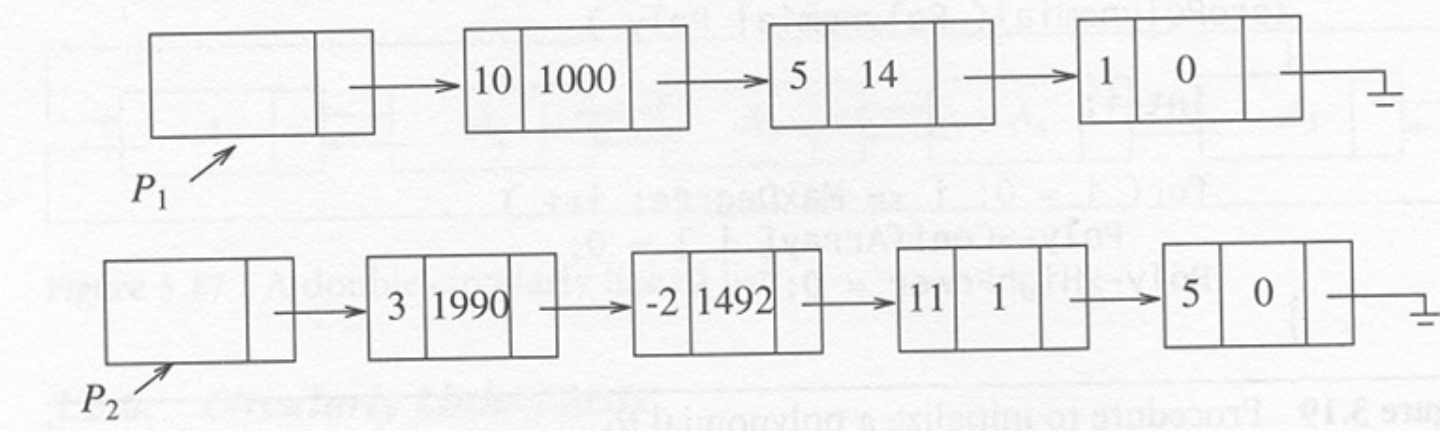
- Multiply two polynomials: better structure

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

Application 1: operations on polynomials

- Linked list representation of the previous structure

$$P_1(X) = 10\,X^{1000} + 5X^{14} + 1$$
$$P_2(X) = 3X^{1990} - 2X^{1492} + 11X + 5$$



Implementation exercises

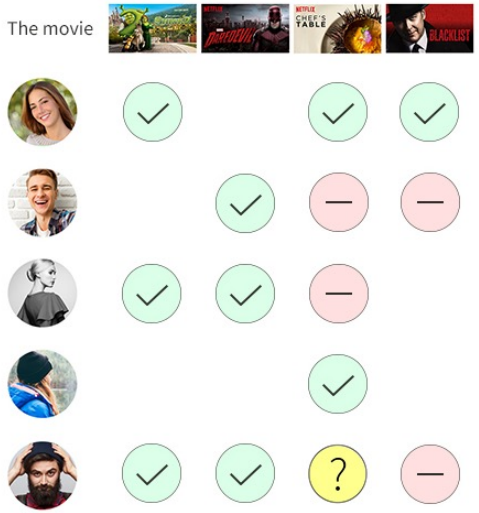
- Suppose we have two polynomials represented by linked lists, with m and n nodes, which are sorted according to their degrees
  - Write the pseudocodes of adding them
    - Hint: recall the merge function in MergeSort; create a new linked list and merge two linked lists where nodes with the same degrees are added together
  - Write the pseudocodes of multiplying them
    - Hint: create m linked lists and then merge them; create a new linked list and merge m linked lists

Application 2: matrix representation

- A university has 40,000 students and 2,500 subjects
  - How to represent the students' scores?

	Math	Art
Tom	80	90
Jack	68	88
...	...	...

- A company has 10 million users and 10,000 movies
  - How to represent users' ratings on movies?



Application 2: matrix representation

- Use 2D Array
  - Students and subjects
    - 40,000 students and 2,500 subjects
    - Total elements: 40K x 2.5K = 100M entries
    - If each student takes 3 subjects => only 120K entries (~0.1% of 100M) => waste of resources
  - Users and movies
    - Suppose we have 10 million users and 10,000 movies
    - Total elements: 10,000,000*10,000 = 100G items
    - How to save?

Use sparse matrix!

Application 2: matrix representation

- First solution: triplet representation (minimum space)

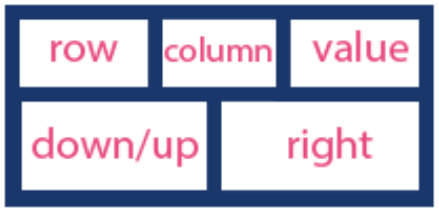
$$\begin{bmatrix} 0 & 0 & 0 & 0 & 9 & 0 \\ 0 & 8 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

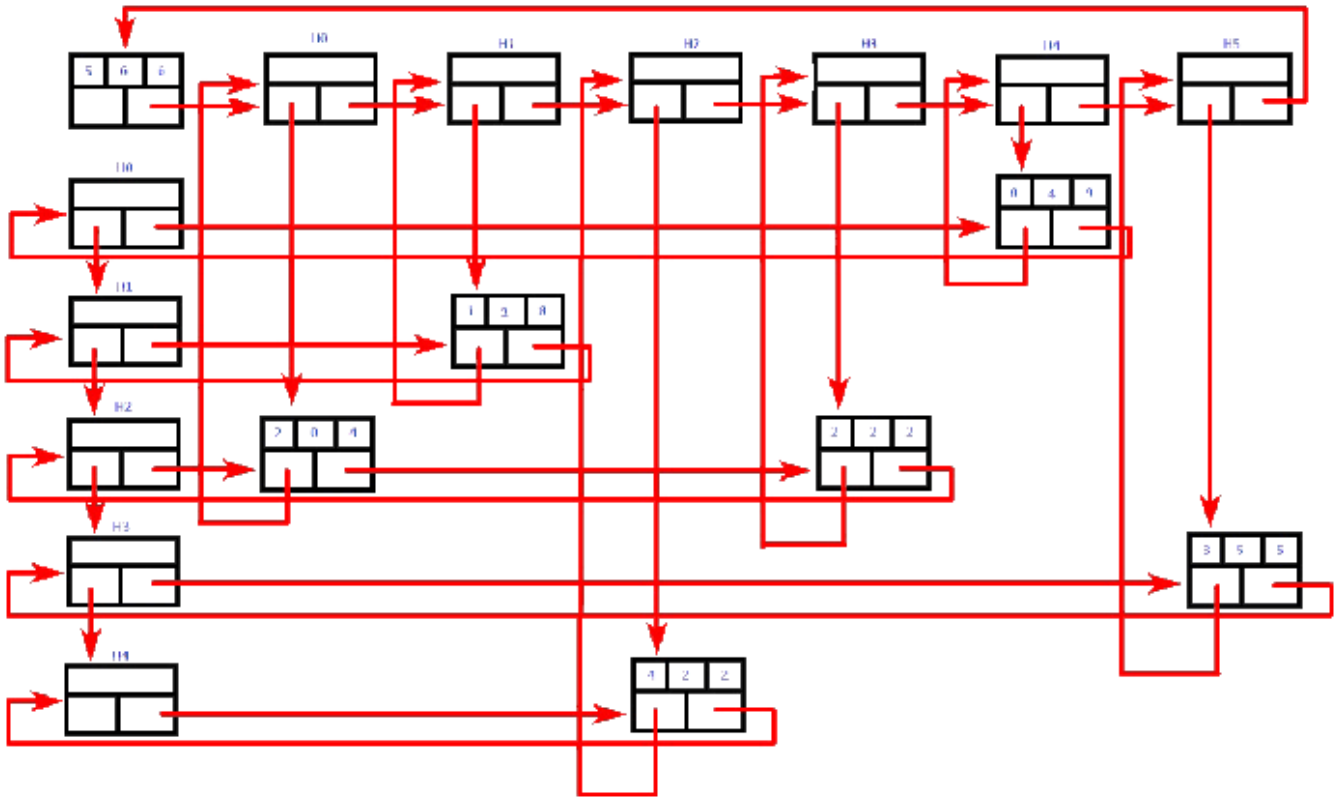
Rows	Columns	Values
0	4	9
1	1	8
2	0	4
2	2	2
3	5	5
4	2	2

Application 2: matrix representation

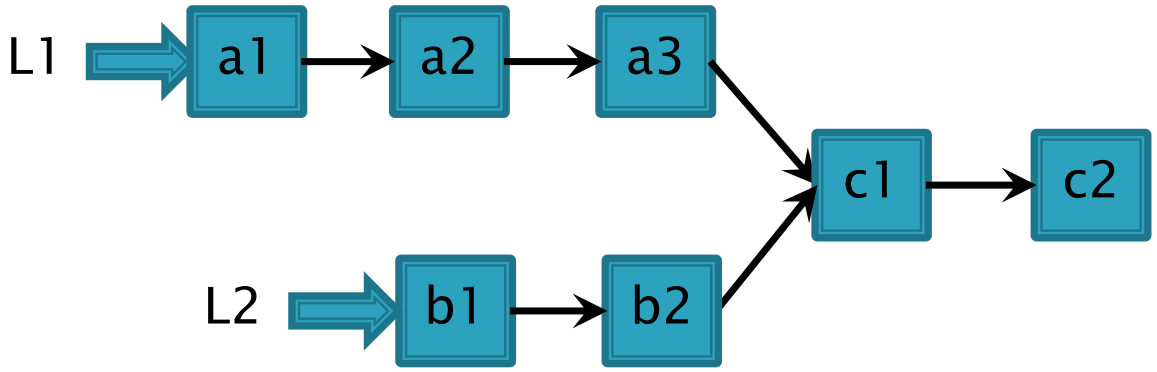
- Second solution: linked list representation

Element Node





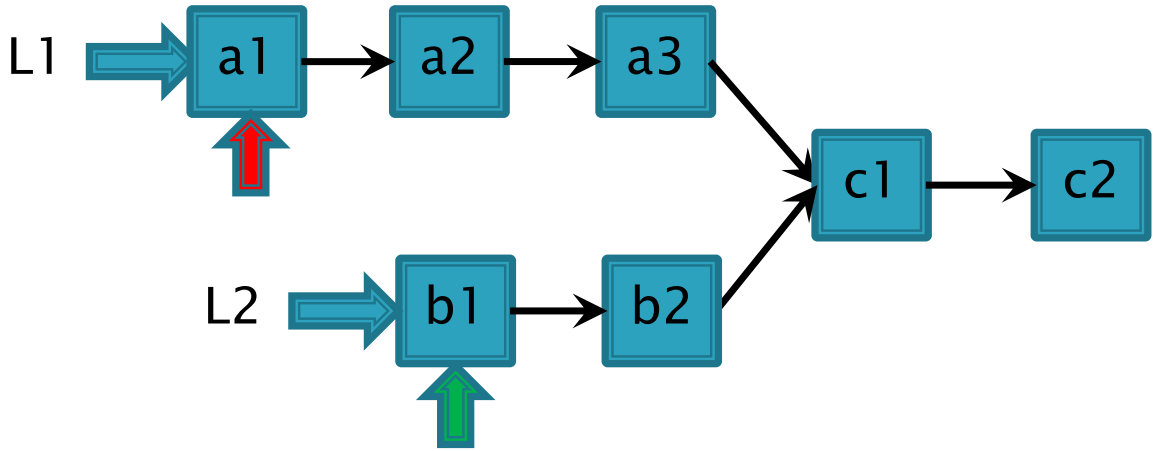
Exercise 1: intersection of two linked lists



Setting: L1 has M elements while L2 has N elements

Goal: find the first node where L1 and L2 intersect

Exercise 1: intersection of two linked lists



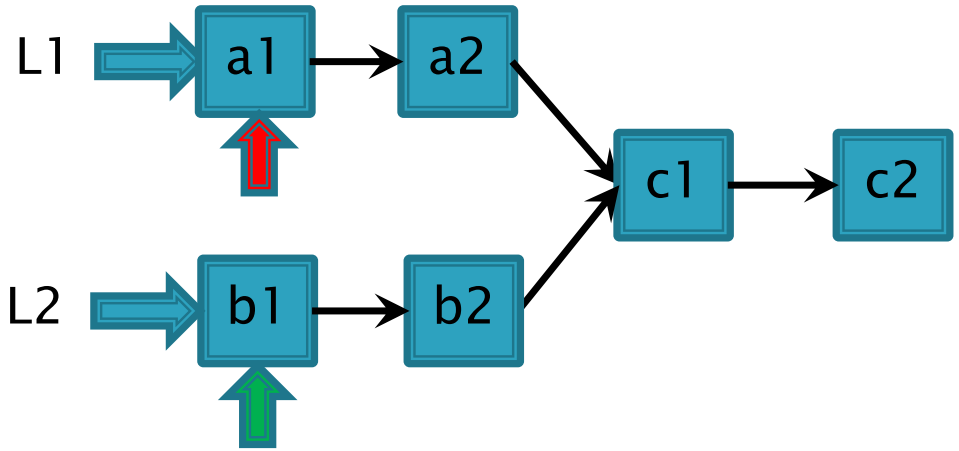
- Use pointer *A* to traverse L1, and use pointer B to traverse L2
- Compare every possible pair of *A* and B

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
```

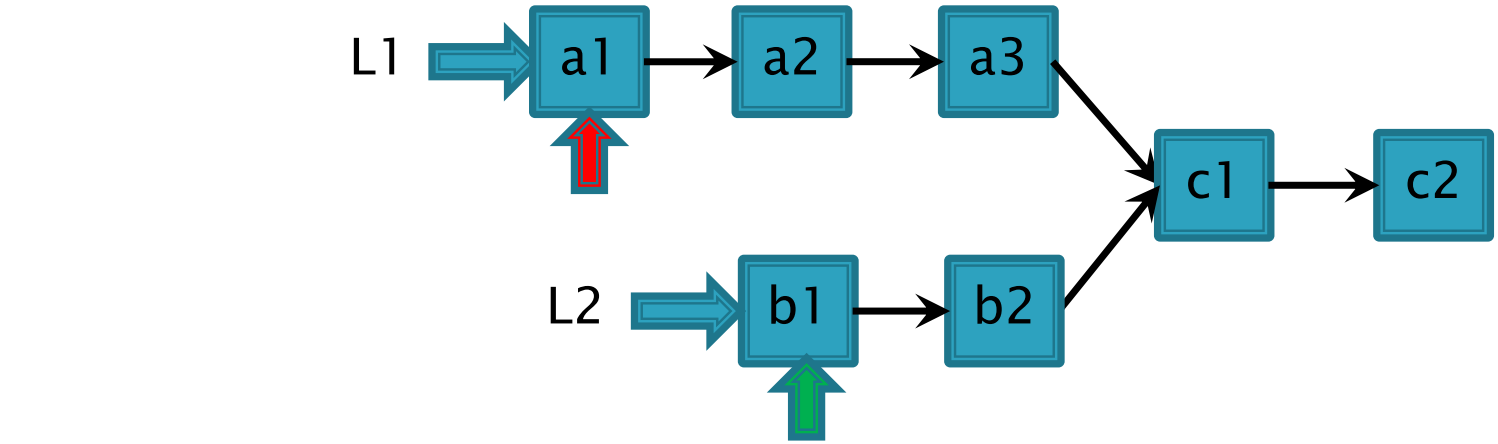
O(MN)

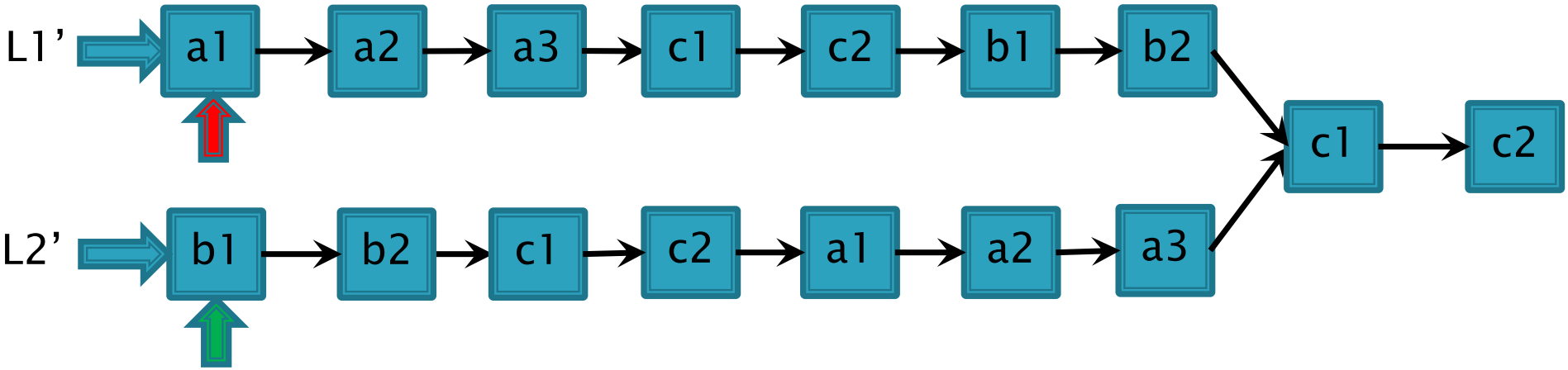
Consider a special case



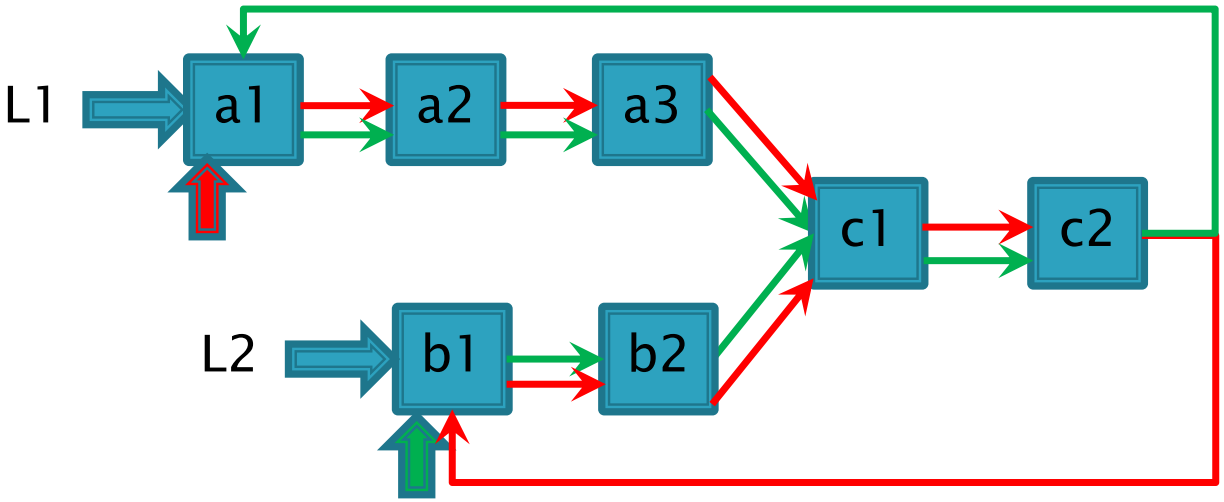
If L1 and L2 have the same length, the problem is easy to solve

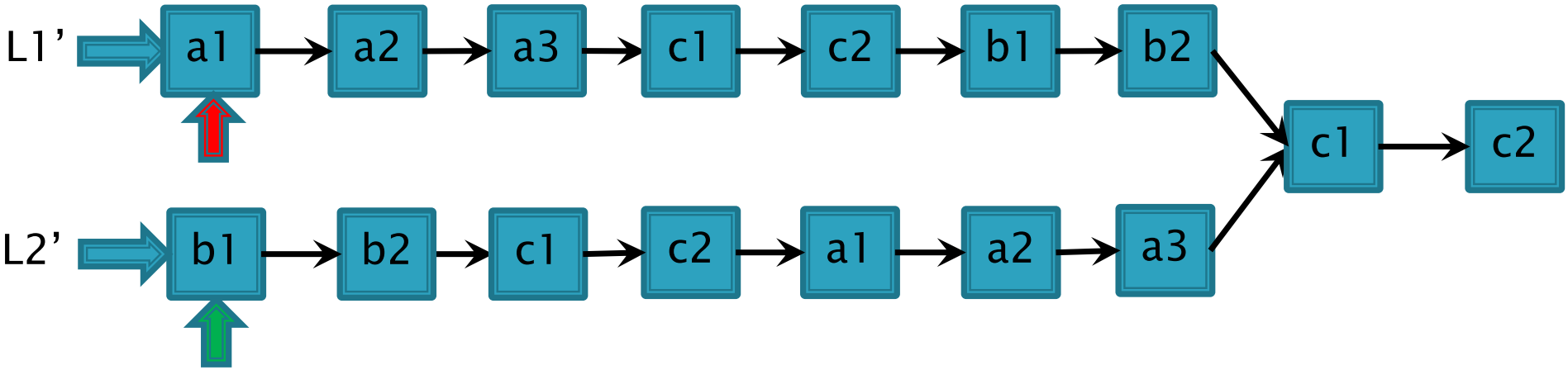
Method #2


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
c1
c2

Method #2


L1
a1
a2
a3
c1
c2
L2
b1
b2


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

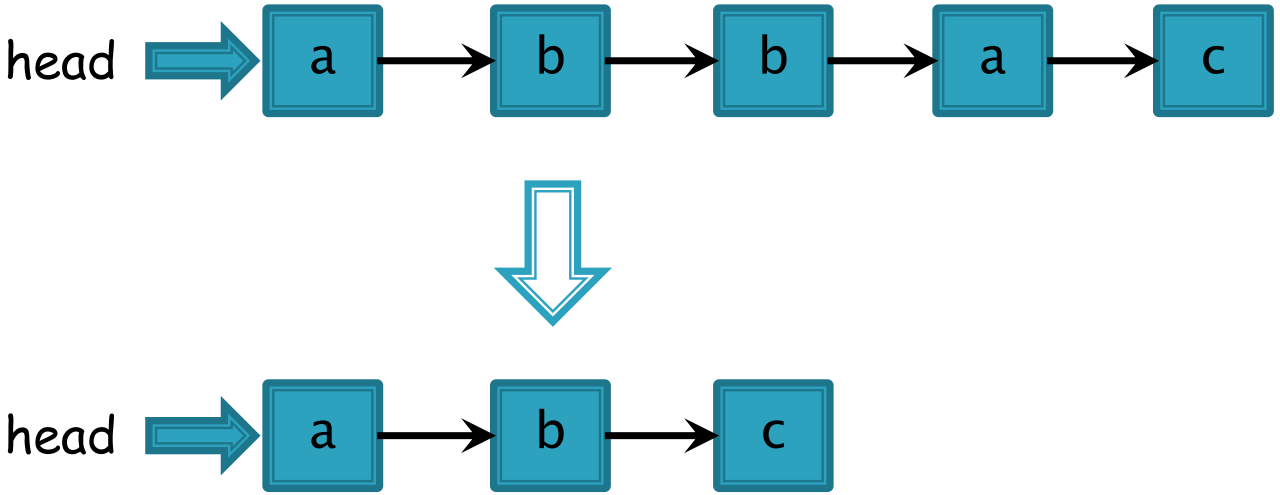
Method #2

```
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)

Exercise 2: duplicate letter detection

Given the head of a singly linked list L, in which each node's data is a lowercase letter, remove all the nodes with duplicate lowercase letters



Method #1

```
If L.head = null return
A = L.head.next
while A != null
    dataA = A.data
    B = A.next, pre = A
    while B.next != null
        dataB = B.data
        if dataB == dataA
            pre.next = B.next
        pre = B
        B = B.next
    A=A.next
```

$O(M^2)$

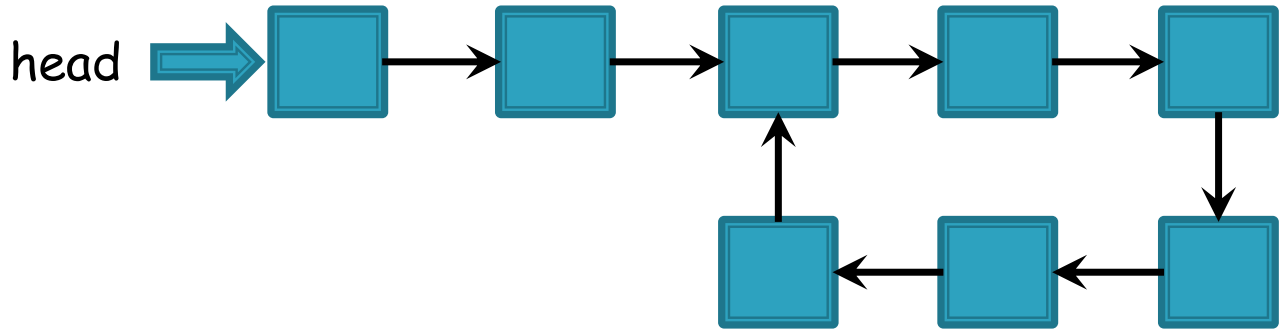
Method #2

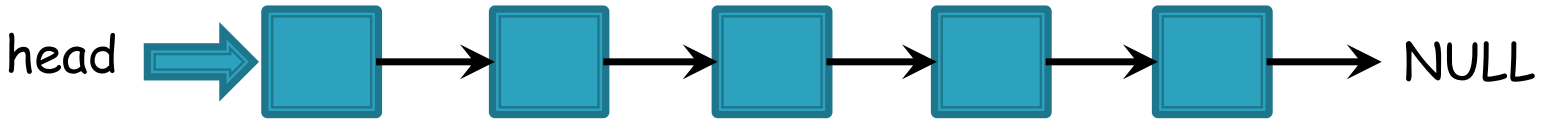
```
Boolean b[ ] = new Boolean[26]
A = L.head, pre = L.head
while A.next != null
    dataA = A.data
    index = dataA – 'a'
    if b[index] == false
        b[index] = true
        pre = A
    else
        pre.next = A.next
A = A.next
```

O(M)

Exercise 3: cycle detection

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





Method #1

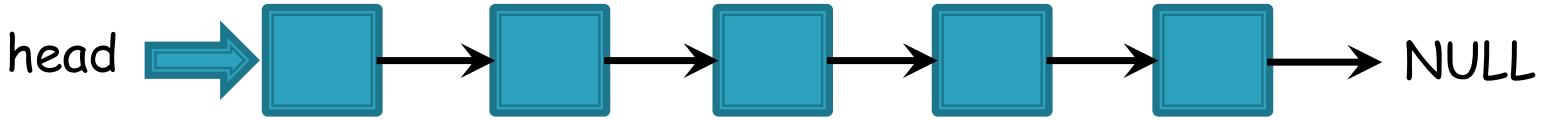
- If *L* is acyclic, we ultimately arrive at NULL by continuously following the next pointer:

```
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

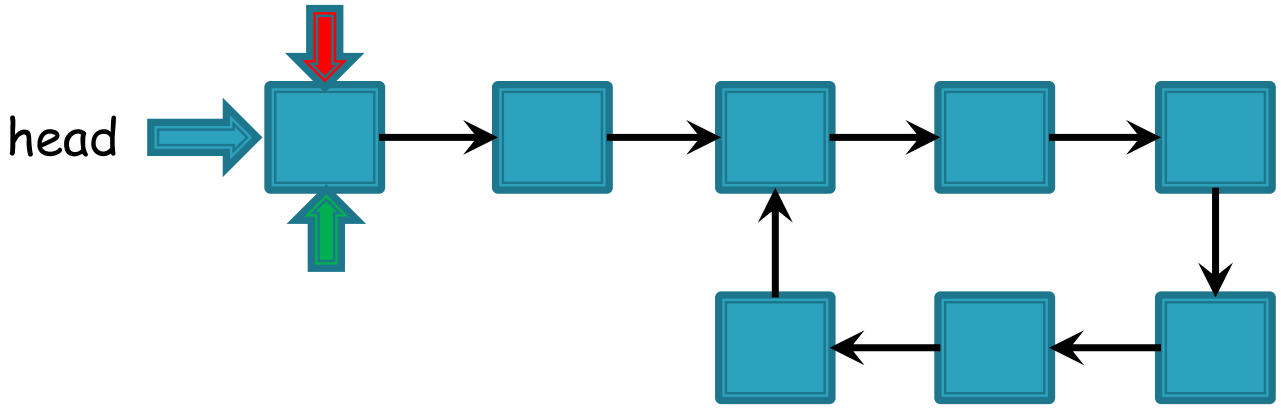


- Store all revisited noded in a new list L':

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

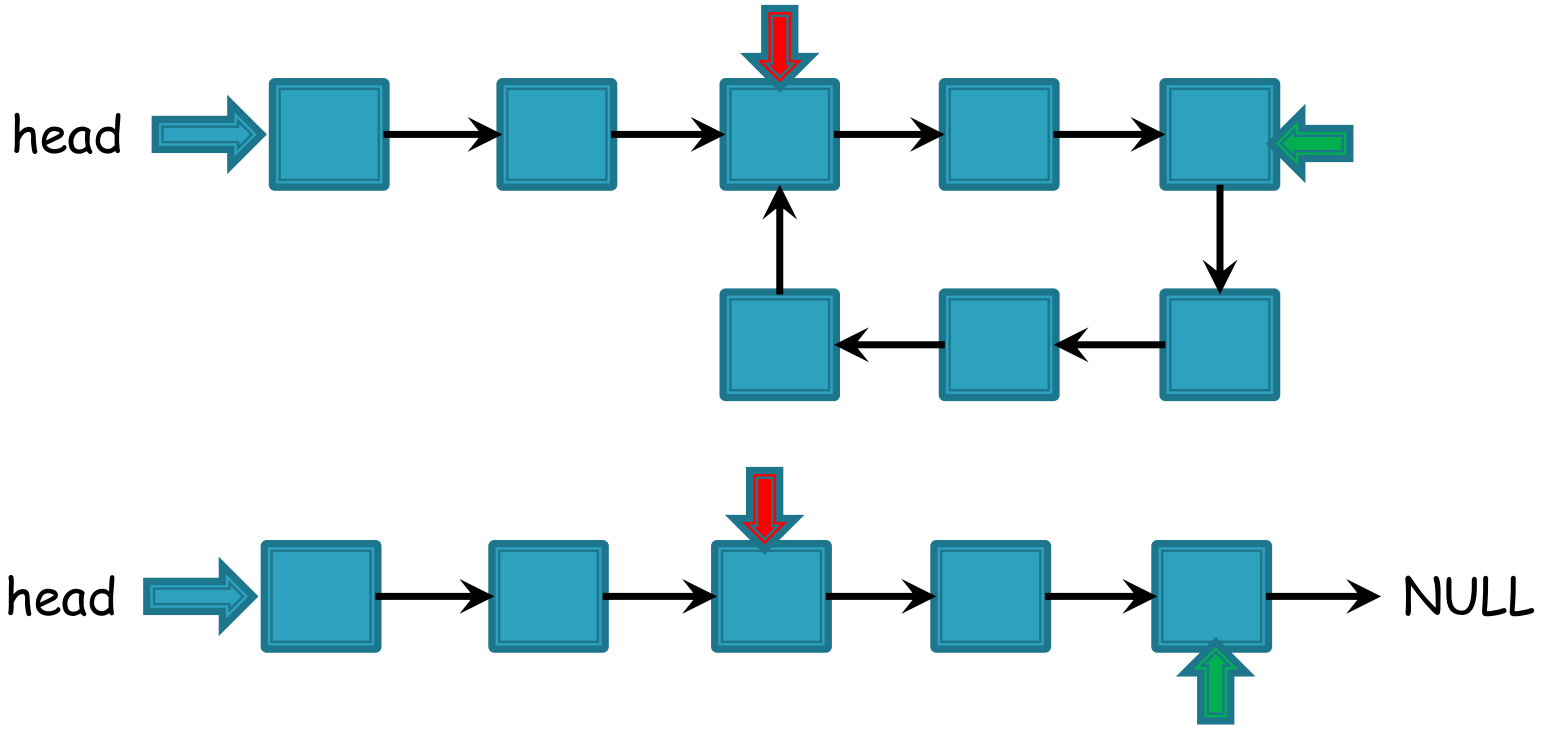
- Search on L' is expensive; use a Hash table

Method #3



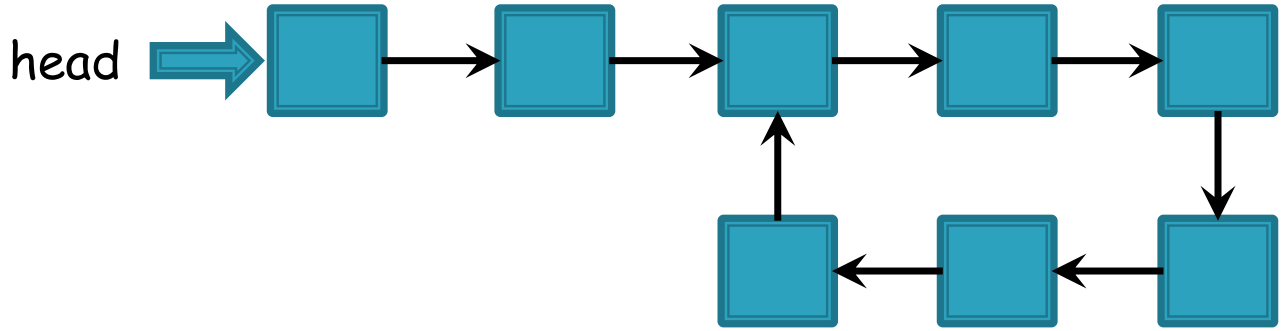
- Use two pointers A and B, both initialized to head
- Every time A=A.next while B=B.next.next

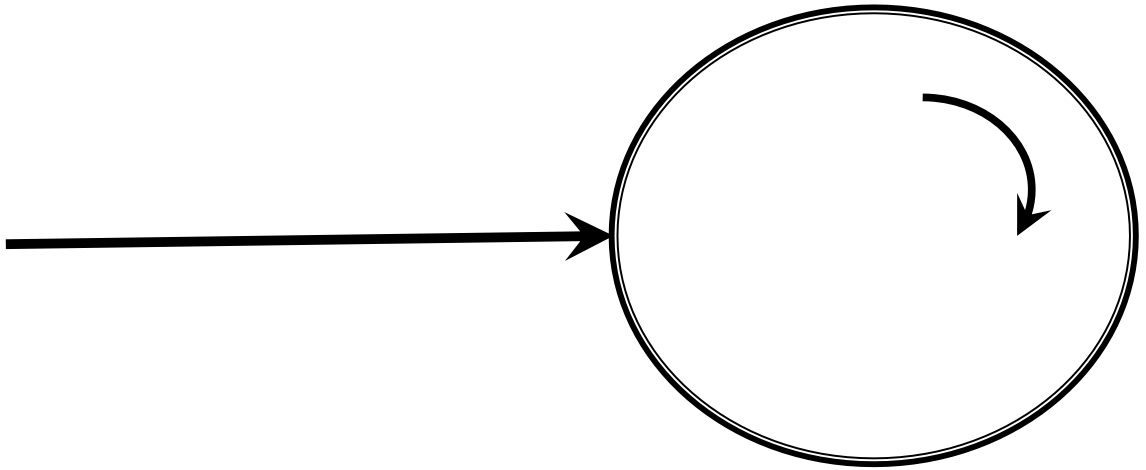
Method #3



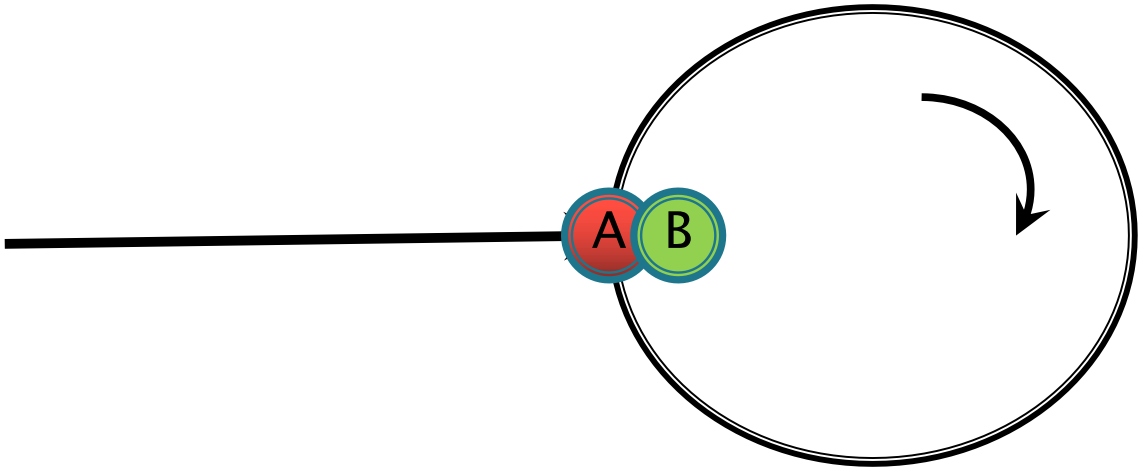
- If *L* is acyclic, either B or B.next be NULL
- If *L* is cyclic, B enters the cycle earlier than *A*

Method #3



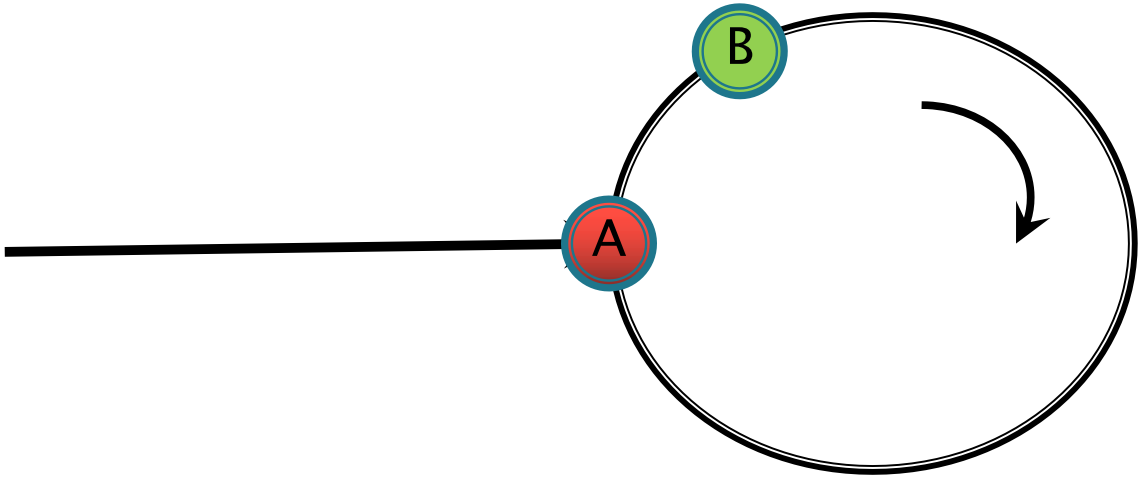


Method #3



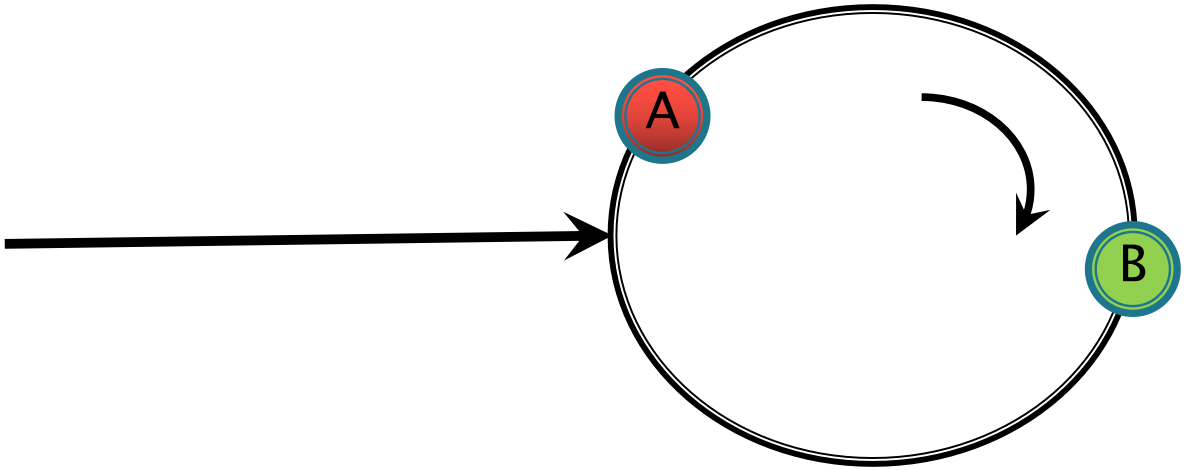
- Case I: B is exactly at entrance when A arrives at the cycle
- So A and B meet at entrance

Method #3



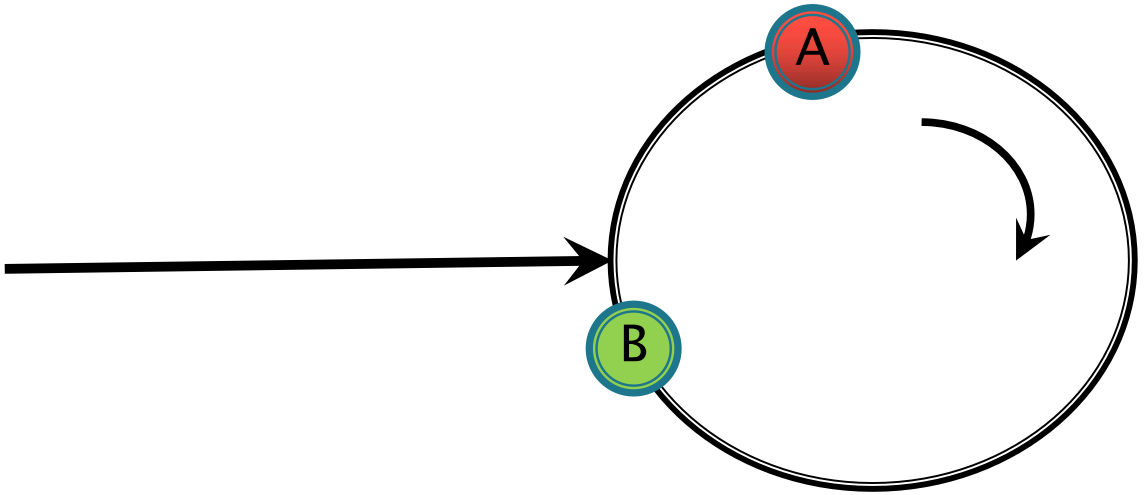
- Case II: B is somewhere else in the cycle when A arrives at entrance

Method #3



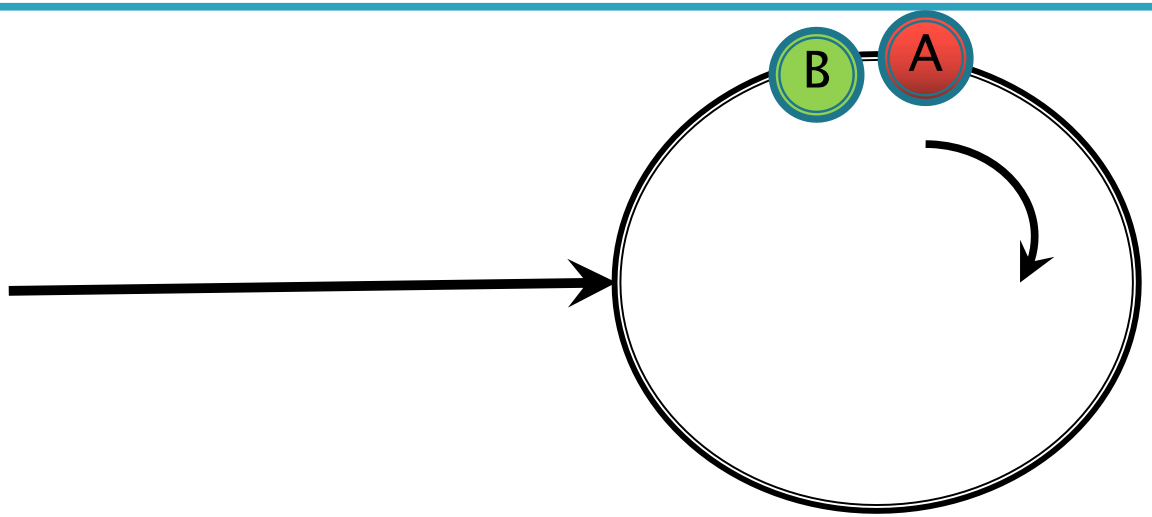
- Since B is moving faster, it must overtake A at a certain point in time
- After t timestamps, the distance gap is (2-1)t = t
- The distance of a circle is x, which is a constant
- So the distances between A and B are 0,1,2,…, t-1

Method #3



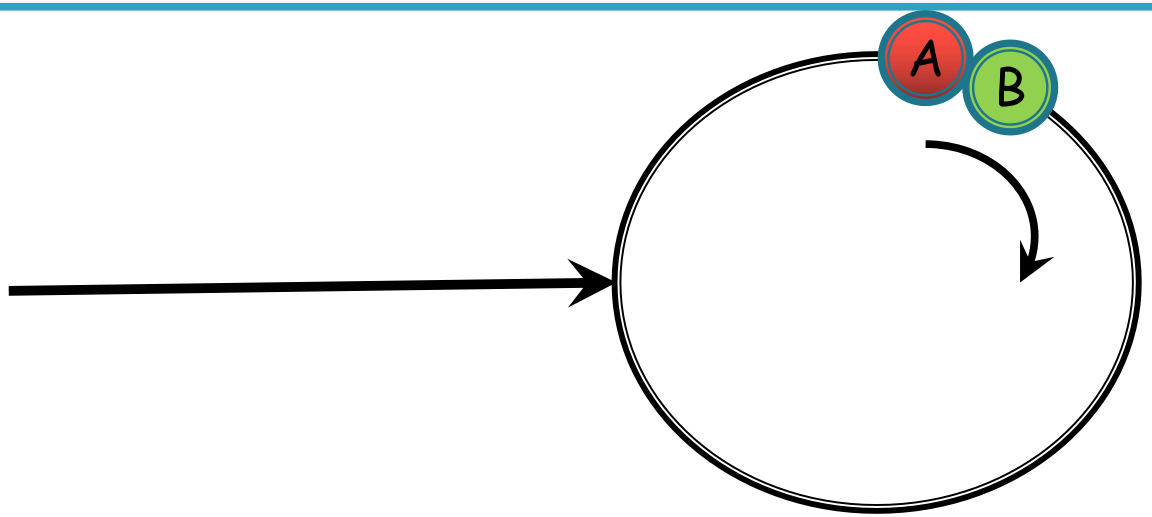
- Since B is moving faster, it must overtake A at a certain point in time

Method #3



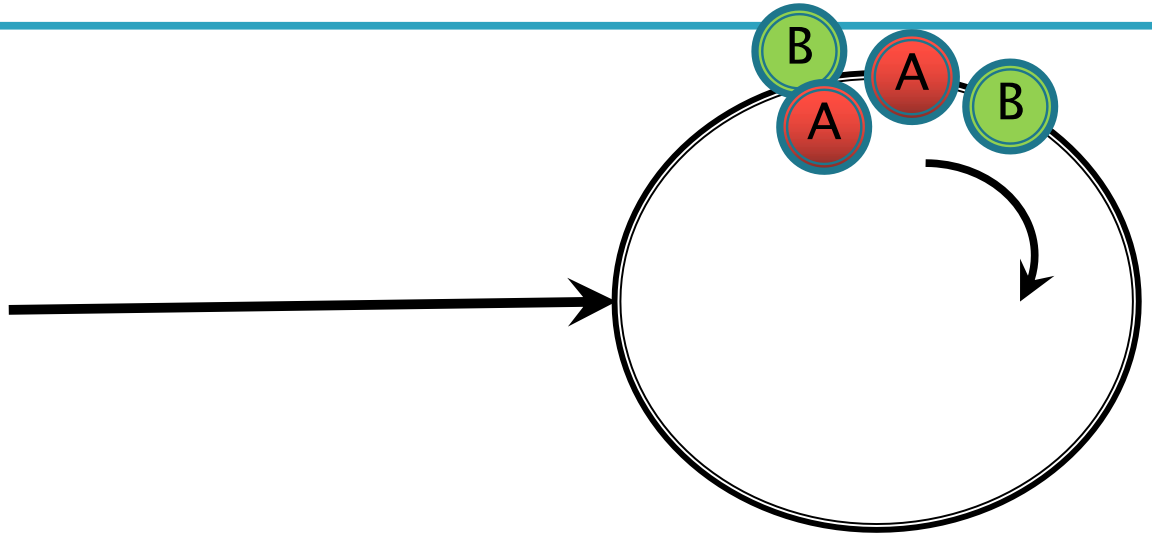
- Since B is moving faster, it must overtake A at a certain point in time

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 list 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 lecture
  - Stack and queue