List (Part II)

A linear collection of items

Assumptions

- The list is contiguous (“no gaps”).
- Items are identified by their position in the list.
- The first item is at position 1.

Primary methods (IList)

```
// Adds an item at the end of the list
public void Add(T item)

// Inserts an item at position p in the list
public void Insert(T item, int p)

// Removes the item at position p in the list
public void RemoveAt(int p)
```

Primary methods (IList)

```
// Returns true if item is removed from the list;
// false otherwise
public bool Remove(T item)

// Retrieves the item at position p
public T Retrieve(int p)

// Returns true if the item is found in the list;
// false otherwise
public bool Contains(T item)
```

Supporting methods (IContainer)

```
// Resets the list to empty
public void MakeEmpty()

// Returns true if the list is empty; false otherwise
public bool Empty()

Returns the number of items in the list
public int Size()

// Outputs the list to console (not part of IContainer)
public void Print()
```

Generic List class

```
class List<T> : IList<T>, IContainer<T>
{
    ...
}
```

Note: In the code, IList inherits IContainer.

Data structures

- Linear array
- Singly linked list ←

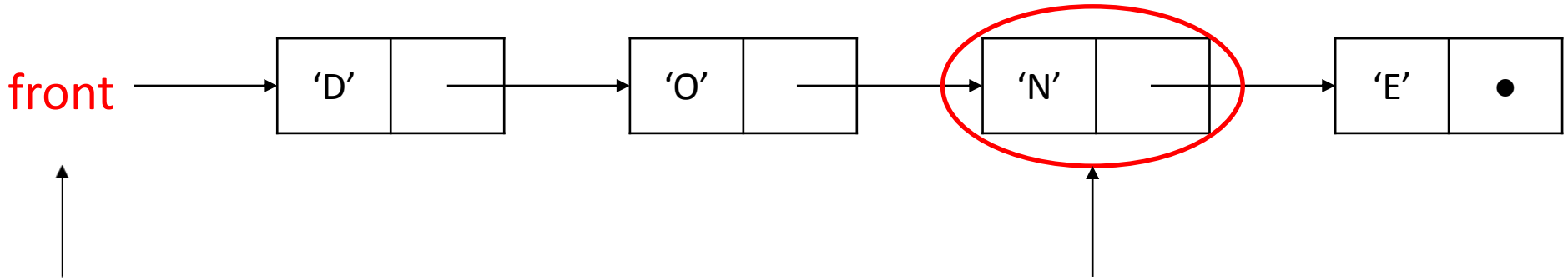
Singly linked list

A singly linked list differs from a linear array in the following ways:

1. Items are not stored in contiguous memory locations.

2. Instead, an instance of a Node class stores an item and a reference to another (next) instance of the Node class which stores an item and a reference to another (next) instance of the Node class, and so on. Hence, the term “linked”.

Diagrammatically



Reference to the first Node

One instance of the Node class

Public Node class

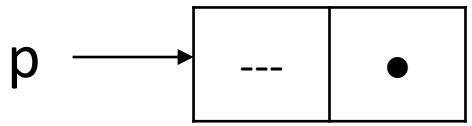
```
public class Node<T>
{
    public T Item       { get; set; }
    public Node<T> Next { get; set; }

    // Parameterless Constructor
    public Node()
    {
        Next = null;
    }
    ...
}
```

Read/Write Properties

Used as:

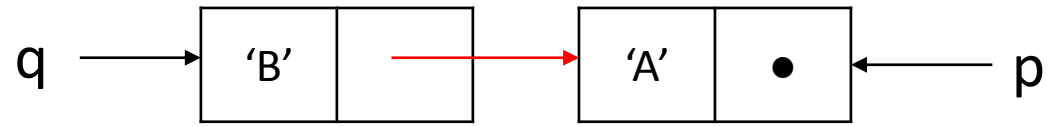
```
Node<char> p;
p = new Node<char>( );
```



```
public class Node<T>
{
    ...
    // Constructor
    public Node(T item, Node<T> next=null)
    {
        Item = item;
        Next = next;
    }
}
```

Used as:

```
Node<char> p, q;
p = new Node<char>('A');
q = new Node<char>('B', p);
```



Back to the singly linked list

Data Members

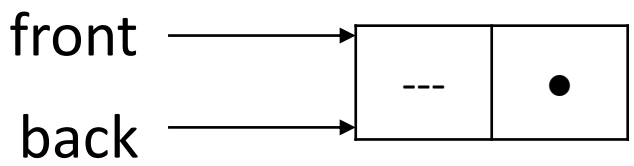
```
public class List<T> : IList<T>
{
    ...                         // Private Node class (See Stack)
    // Data members
    private Node front;         // Reference to the first node of the list
    private Node back;          // Reference to the last node of the list
    private int count;          // Number of items in the list
    ...
}
```

Constructor

- Basic strategy:
  - Create a “header” node and set the count to 0.

```
public List()
{
    MakeEmpty();
}

public void MakeEmpty()
{
    back = front = new Node();
    count = 0;
}
```



Add an item at the end of List L

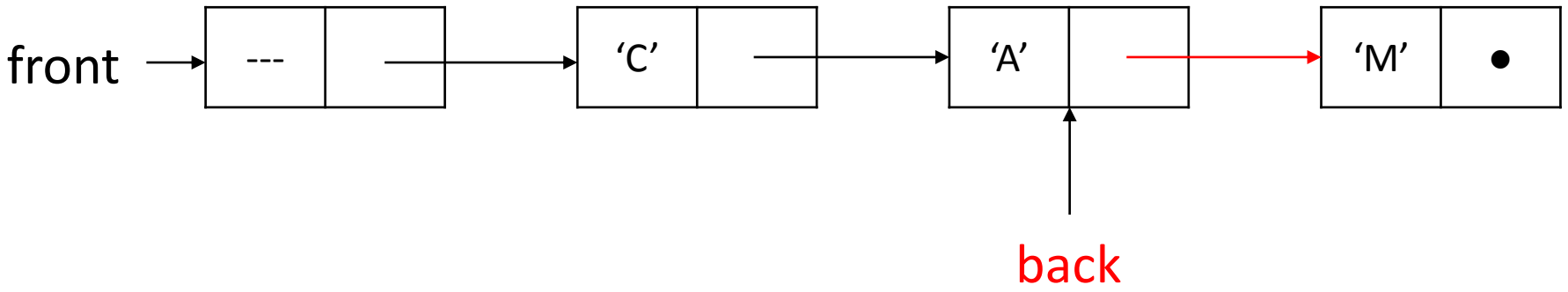
- Basic Strategy

  - Create a new node and assign it to back.Next.
  - Move back to the new node.
  - Increase count by 1.

```
public void Add(T item)
{
    back = back.Next = new Node(item);
    count++;
}
```

Add ‘M’ to list L

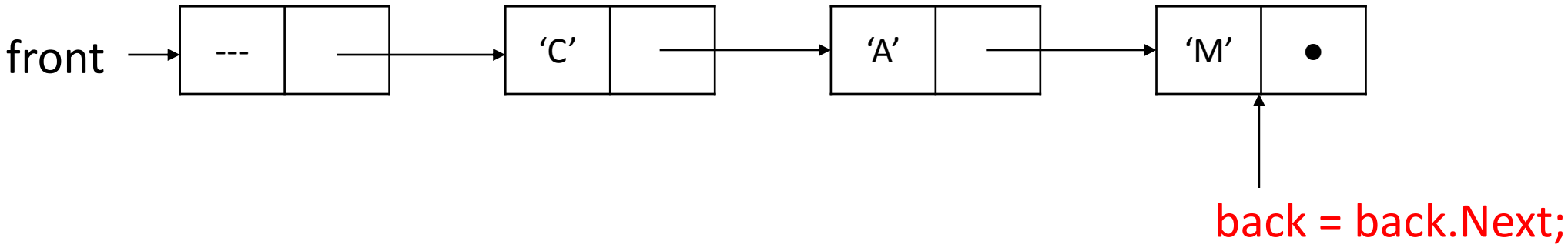
- Create a new node to hold ‘M’ and assign it to back.Next.



back.Next = new Node(item);

Add ‘M’ to list L

- Move back to the new node.

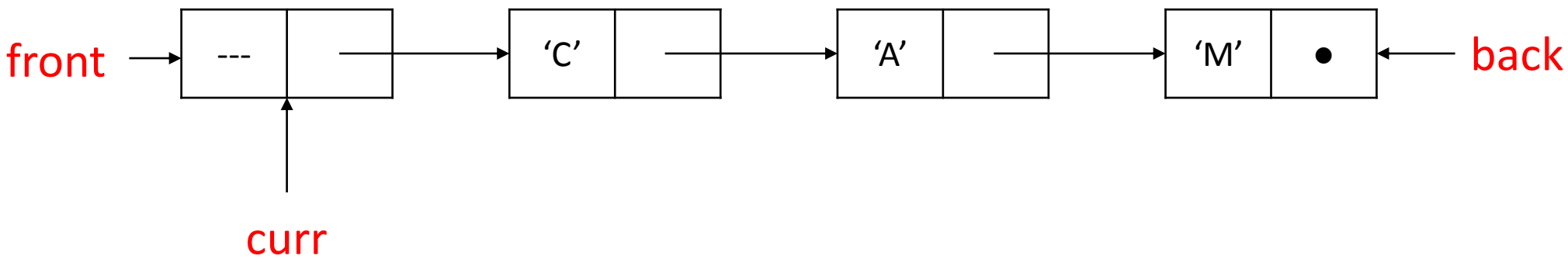


Insert an item at position p in list L

- Basic Strategy
  - If position p is out-of-range, then throw an exception.
  - Else
    - Set a node reference (say curr) to the front of the list.
    - Move curr to position p - 1 (one behind position p).
      - Note: The header node is at position 0.
    - Create a new node and place it between curr and curr.Next.
    - Increase count by 1.
    - Special Case: Invoke Add if p == count + 1 (end of the list).

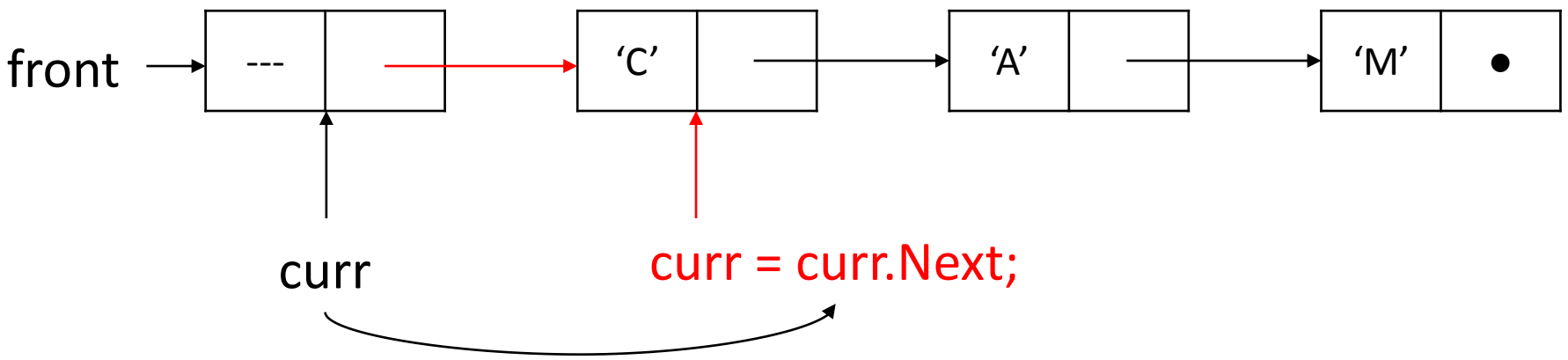
Insert ‘R’ at position 2 in list L

- Set curr to the front of the list.



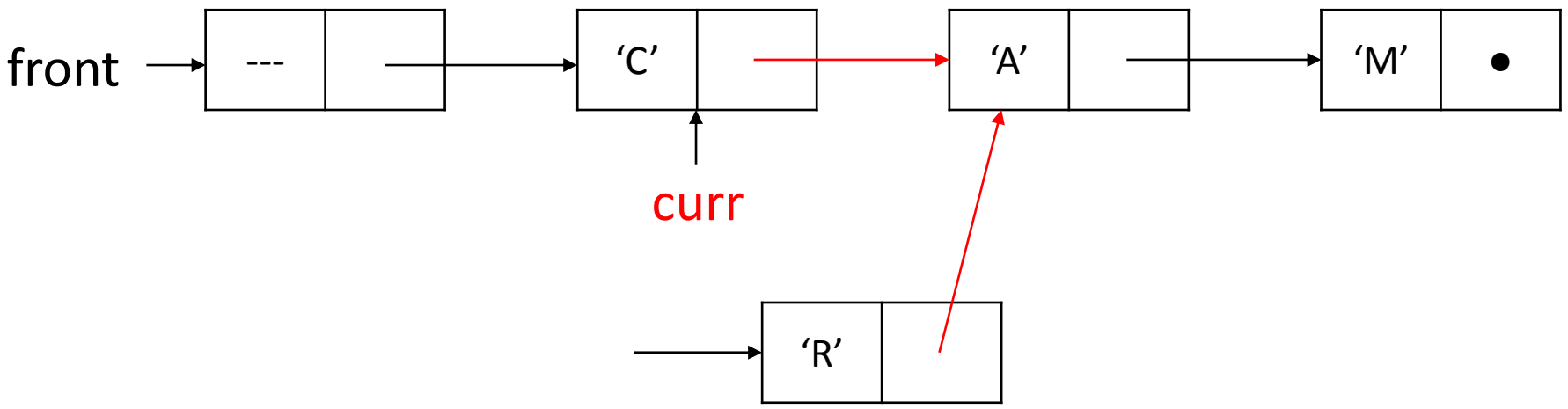
Insert ‘R’ at position 2 in list L

- Move curr to position 1 (i.e., p-1).



Insert ‘R’ at position 2 in list L

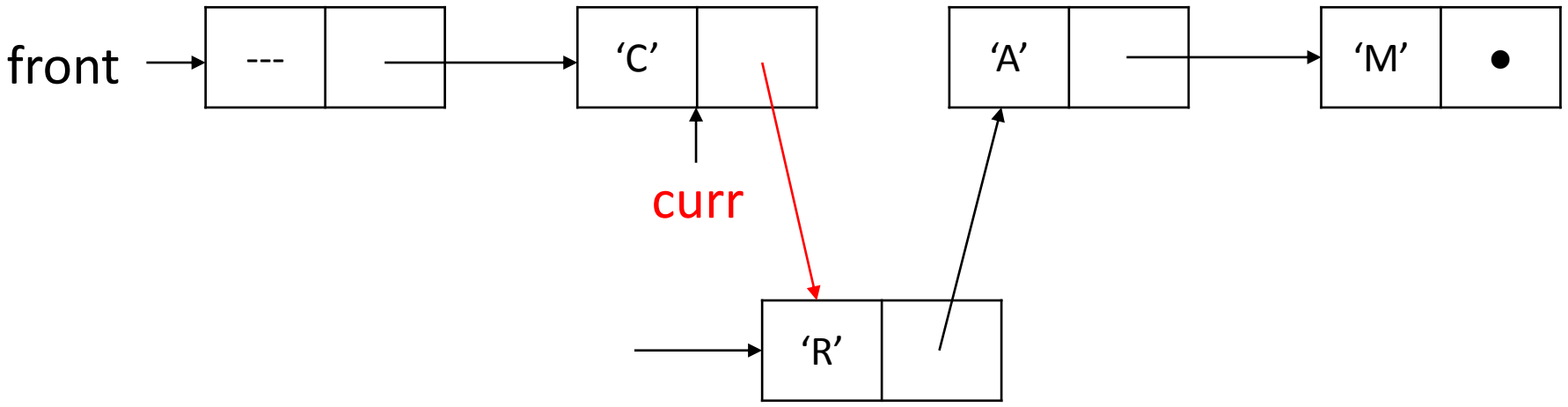
- Create a new node and place it between curr and curr.Next.



curr.Next = new Node (item, curr.Next);

Insert ‘R’ at position 2 in list L

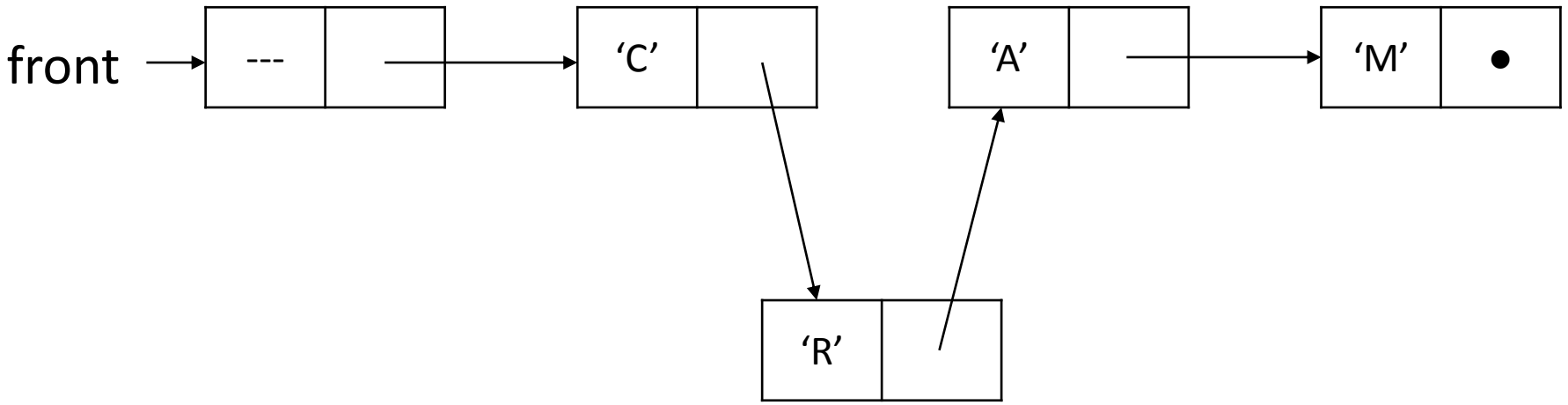
- Create a new node and place it between curr and curr.Next.



curr.Next = new Node (item, curr.Next);

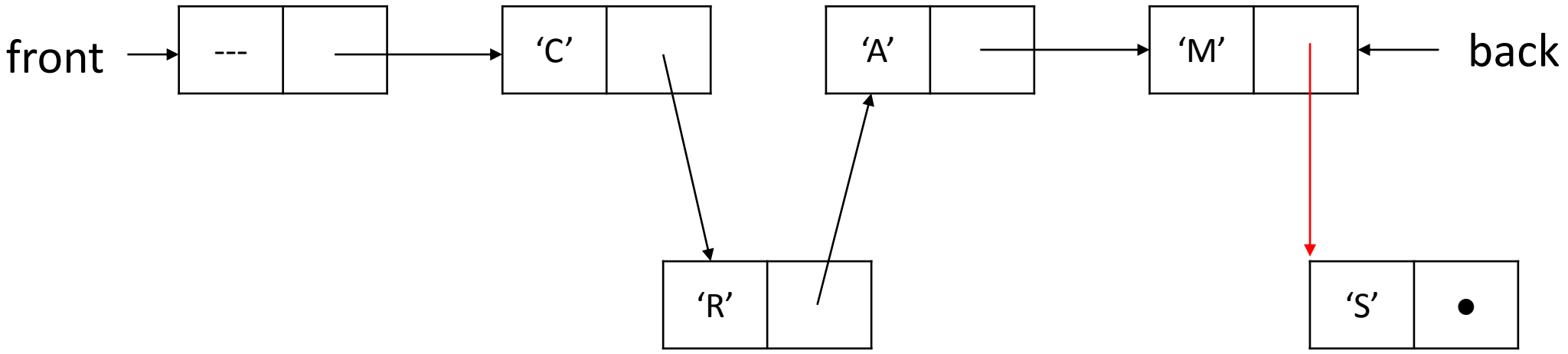
Insert ‘S’ at position 5 in list L (Special case)

- Invoke Add since ‘S’ is added to the end of the list.



Insert ‘S’ at position 5 in list L

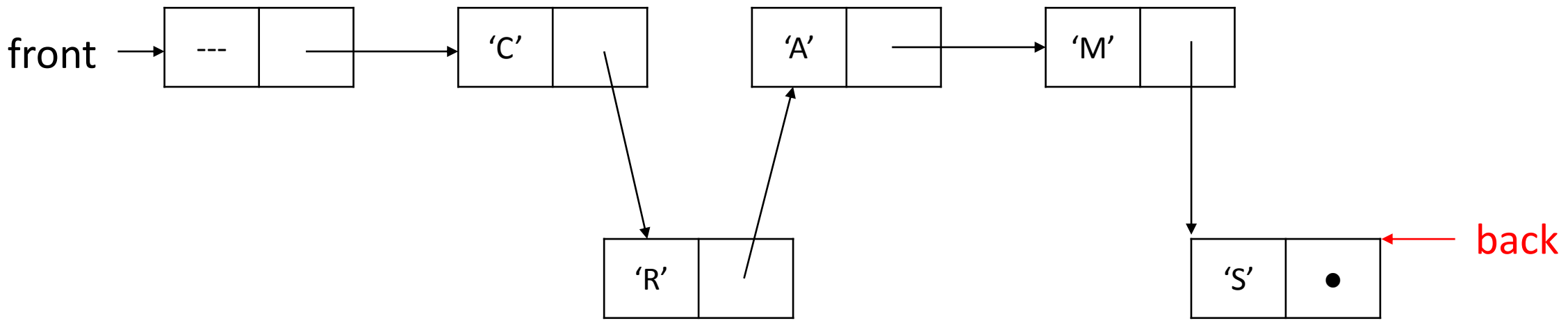
- Create a new node and assign it back.Next.



back = back.Next = new Node (item);

Insert ‘S’ at position 5 in list L

- Assign back to the new node.



back = back.Next = new Node (item);

Worst case time complexity of Insert

- To insert an item at position one before the end of a list with n items, curr must move along n-1 nodes. Therefore, the worst-case time complexity of Insert is:

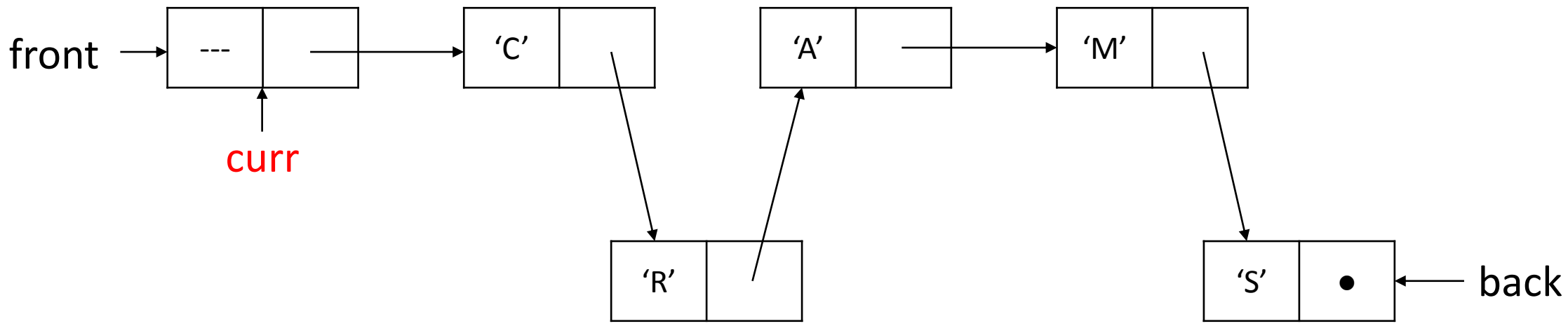
O(n)

Remove an item at position p in list L

- Basic Strategy
  - If the list is empty or the position is out-of-range, throw an exception.
  - Else
    - Set a reference (say curr) to the front of the list.
    - Move curr to position p - 1 (one behind position p).
      - Note: The header node is at position 0.
    - Set the Next field of curr to go around curr.Next (the item at position p).
    - Set back to curr if the last item (node) is removed.
    - Decrease count by 1.

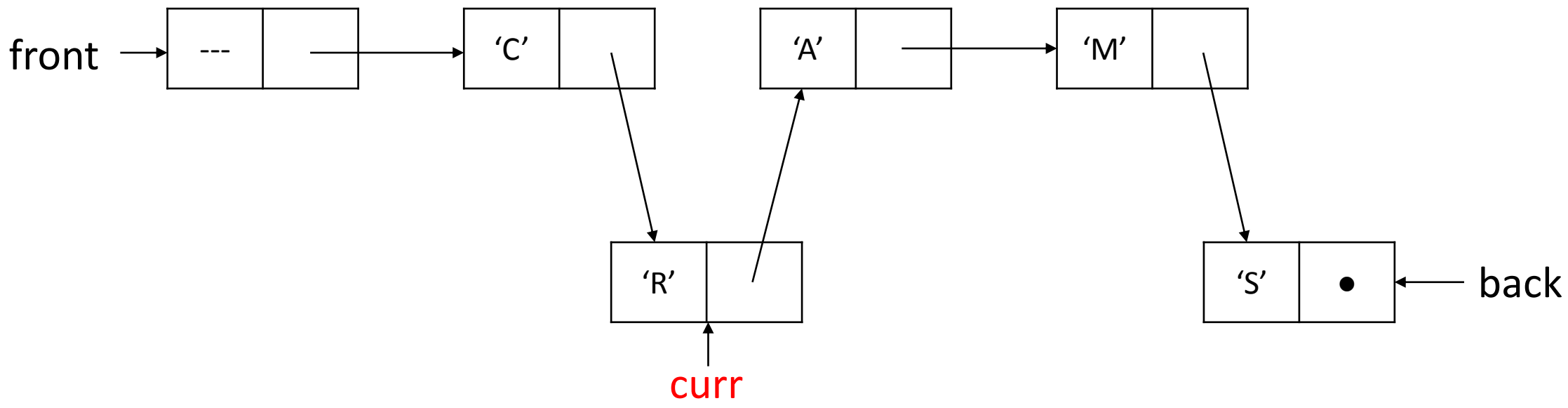
Remove item at position 3 in list L

- Set curr to the front of the list.



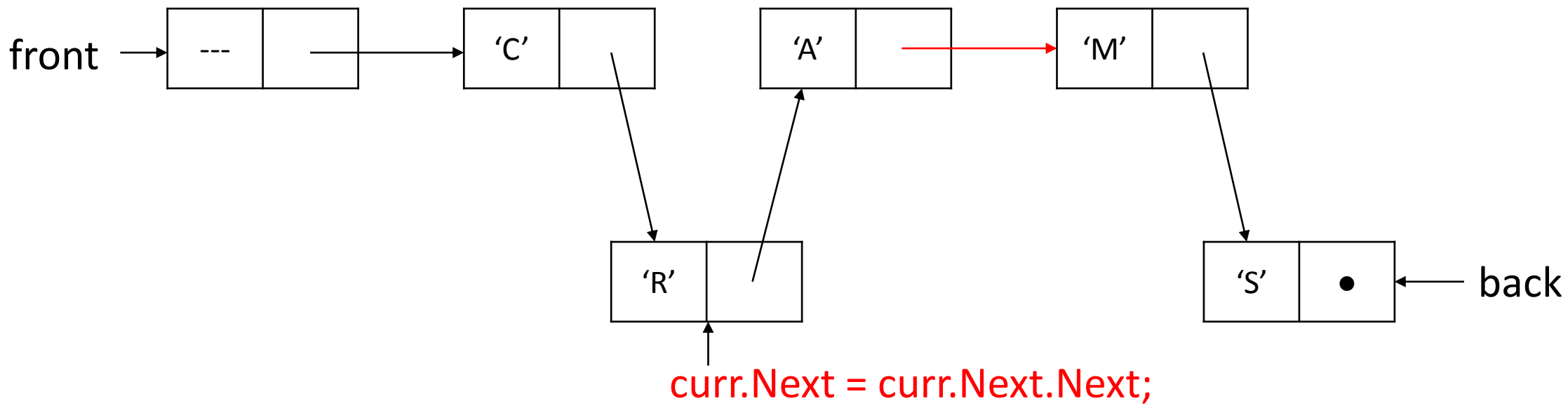
Remove item at position 3 in list L

- Move curr to position 2 (i.e., iterate curr = curr.Next; two times).



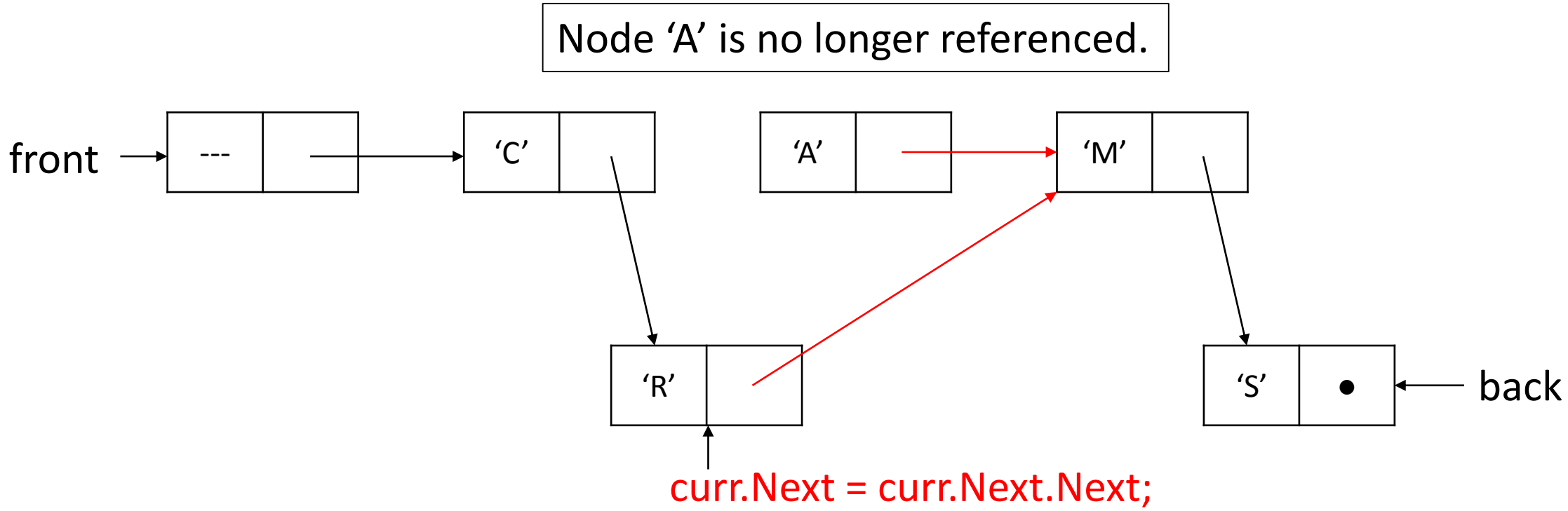
Remove item at position 3 in list L

- Set the Next field of curr to go around the node curr.Next.



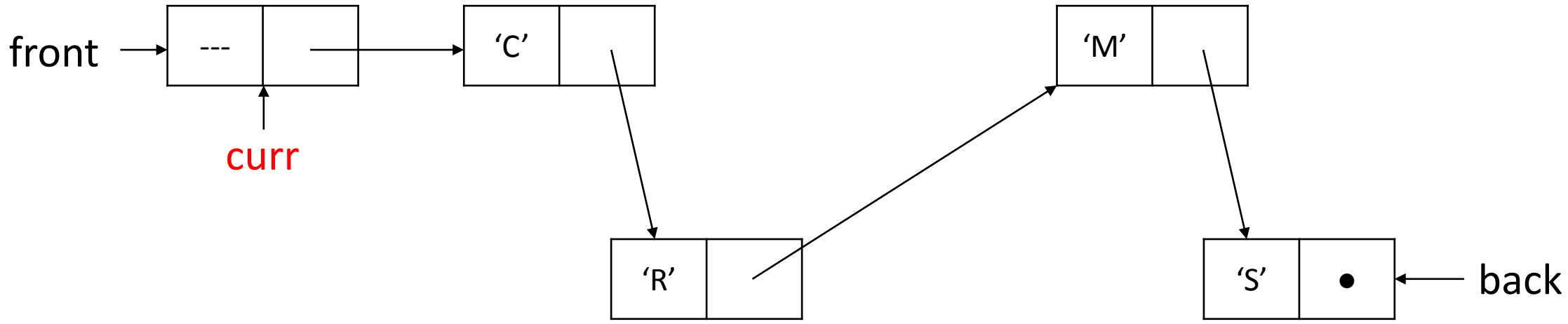
Remove item at position 3 in list L

- Set the Next field of curr to go around the node curr.Next.



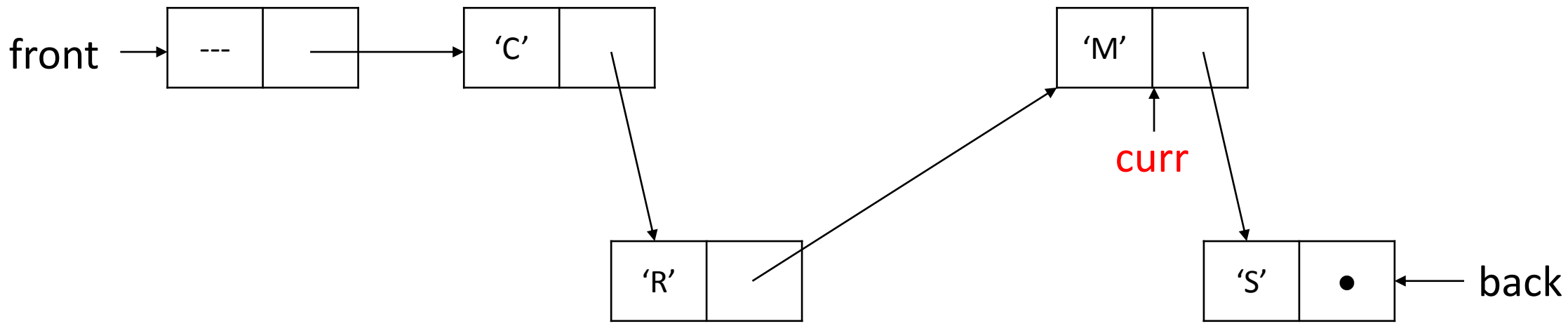
Remove item at position 4 in list L

- Set curr to the front of the list.



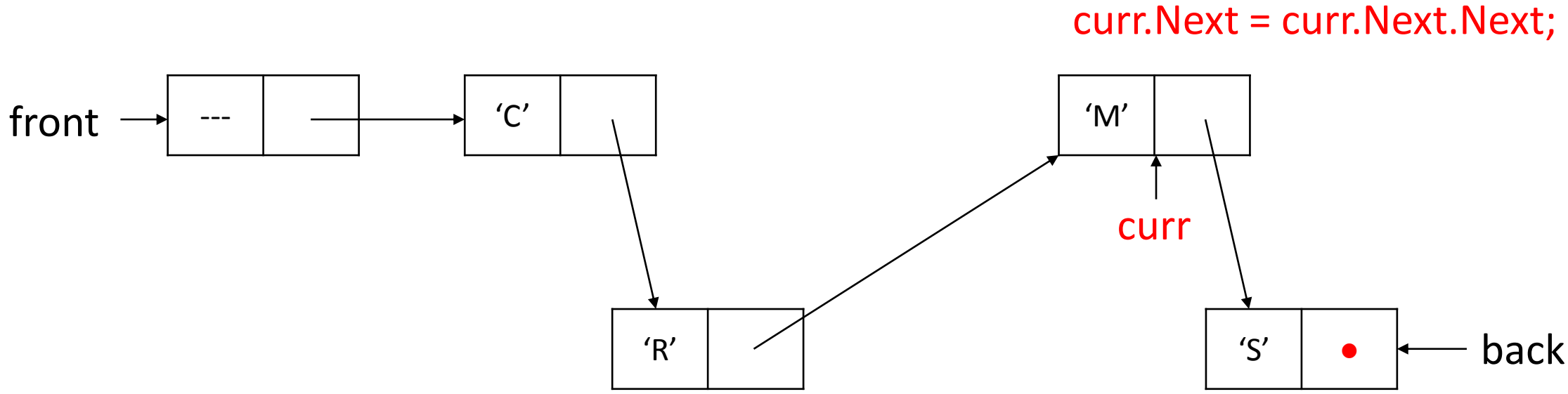
Remove item at position 4 in list L

- Move curr to position 3.



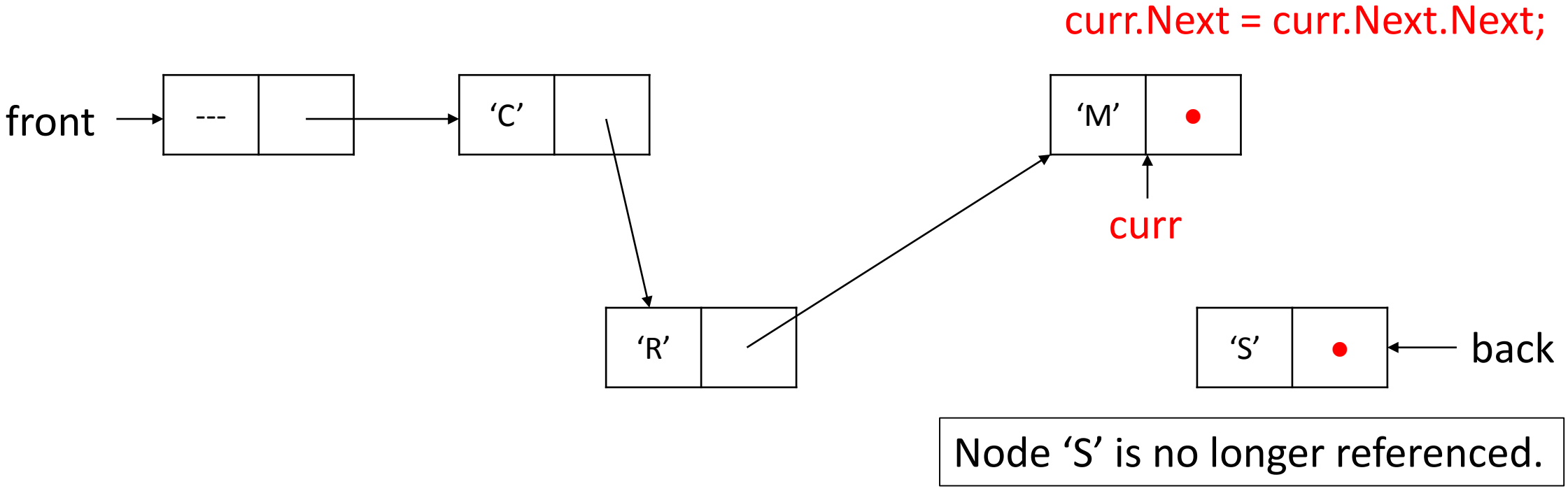
Remove item at position 4 in list L

- Set the Next field of curr to go around curr.Next.



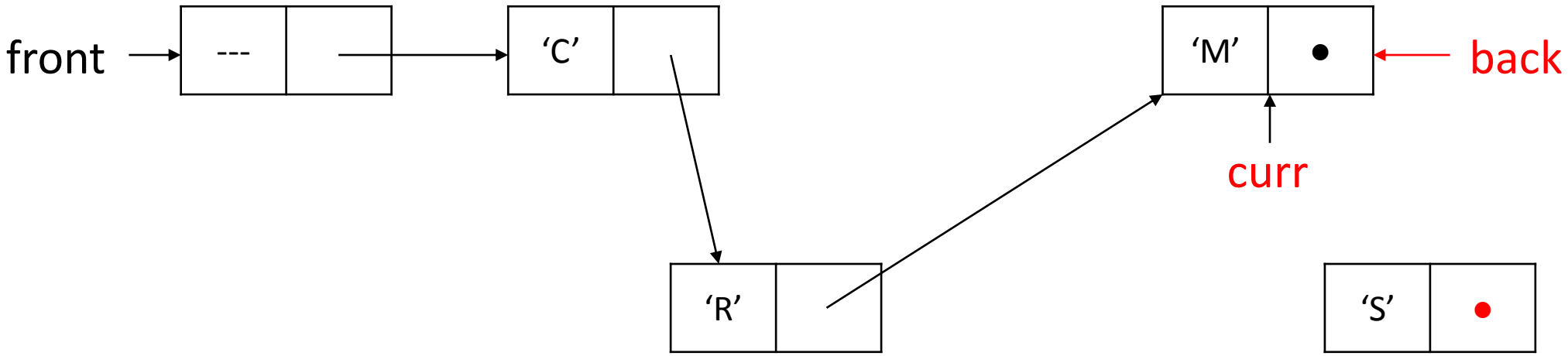
Remove item at position 4 in list L

- Set the Next field of curr to go around curr.Next.

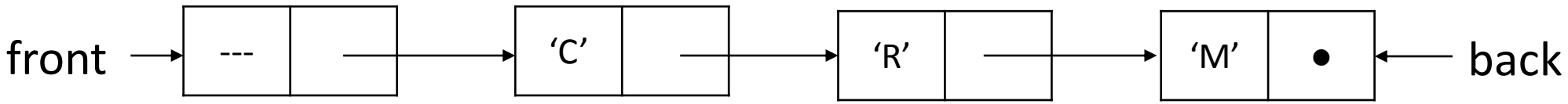


Remove item at position 4 in list L

- Move back to curr.



Final list L



Worst case time complexity of RemoveAt

- To remove the item at the end of a list with n items, curr must move along n - 1 nodes. Therefore, the worst-case time complexity of RemoveAt is:

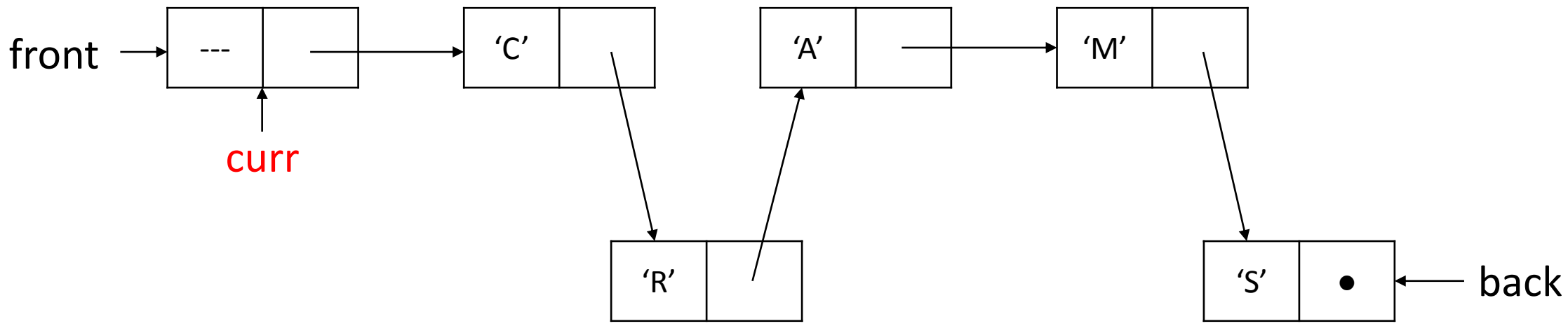
O(n)

Remove item from list L

- Basic Strategy
    - If the list is empty, throw an exception.
    - Else
        - Set a reference (say curr) to the front of the list.
        - Traverse the list until either the item is removed, or the end of the list reached
            - If the item at the Next node is found, to remove:
                - Set the Next field of curr to go around curr.Next.
                - Set back to curr if the last item (node) is removed.
                - Decrease count by 1.
            - Else
                - Move curr to the Next node.
        - If the item is found (removed), return true; otherwise return false.

Remove ‘A’ and ‘S’ as an exercise

- Note the similarity with RemoveAt(3) and RemoveAt(4).



Worst cast time complexity of Remove

- If an item is found and is the last item in the list, then all previous n-1 items (nodes) must be traversed. Hence, the time complexity is O(n).

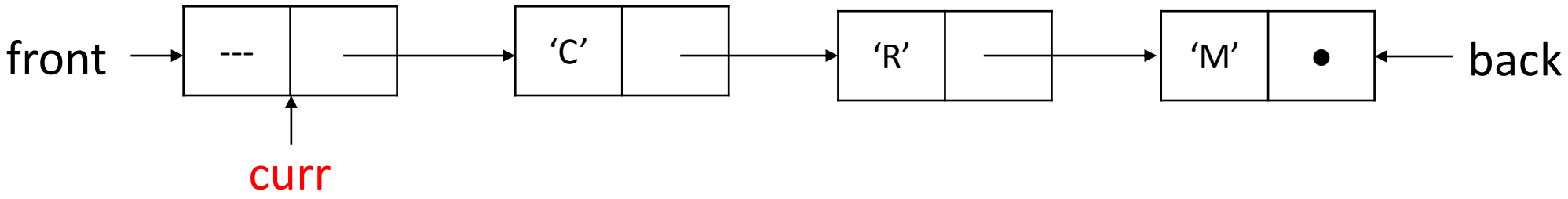
- If the item is not found, the entire list must be traversed. The time complexity is also O(n).

Retrieve item at position p

- Basic Strategy
    - Set curr to the front of the list.
    - Move curr to position p and return curr.Item.

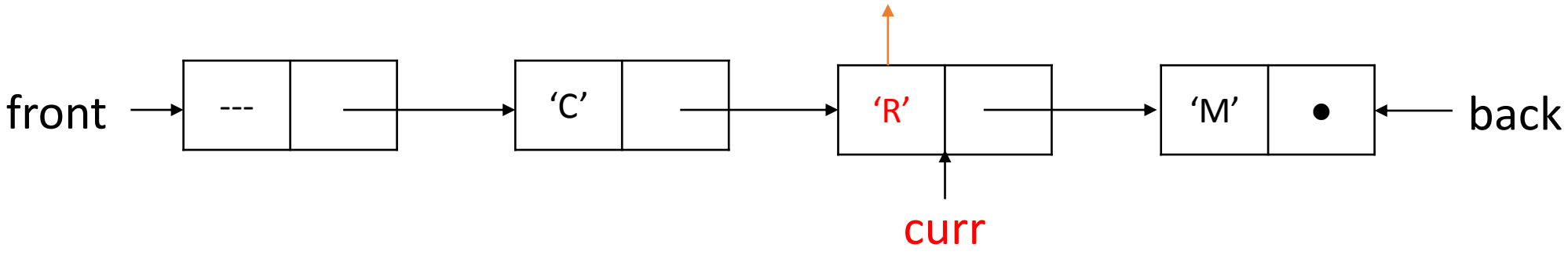
Retrieve item at position p=2 in list L

- Set curr to the front of the list.



Retrieve item at position p=2 in list L

- Move curr to position 2 and return curr.Item.



Worst-case time complexity of Retrieve

- To retrieve an item at the end of a list with n items, curr must move along n nodes. Therefore, the worst-case time complexity of Insert is:

O(n)

Note that the worst-case time complexity of Retrieve for the linear array implementation was only O(1).

Contains: Does list L contain item?

- Basic Strategy
  - Traverse A until either the list is fully examined, or the item is found.
  - Return true if the item is found; false otherwise.

Draw the comparison with the Contains methods of the linear array implementation.

```
public bool Contains(T item)
{
    Node curr = front.Next;                    // i = 0;
    bool found = false;

    while (curr != null && !found)             // i < count
        if (curr.Item.Equals(item))            // A[i].
            found = true;
        else
            curr = curr.Next;                  // i++;
    return found;
}
```

Supporting methods

- MakeEmpty
  - Sets the Next field of the header node to null
  - Sets count to 0
  - O(1)
- Empty
  - Returns true if count is 0; false otherwise
  - O(1)
- Size
  - Returns count
  - O(1)

Exercises

1. Why use a header node i.e., why not set the head of an empty linked list to null?

2. What happens when the following Insert statement:

```
curr.Next = new Node(item, curr.Next);
```

is replaced by:

```
curr.Next = new Node(item);
curr.Next.Next = curr.Next;
```

Exercises (con't)

3. Show the result when the item at position 1 is removed from a list. Can you think of another way to remove the first item in a list?

4. Why must MakeEmpty set the Next field of the header node to null?

5. Using only the RemoveAt method, what is the minimum and maximum number of total steps to remove all items from an initial list with n items? State the worst-case time complexity to remove all items using the Big-Oh notation.