Singly Linked Lists

Dr Yuhua Li

School of Computer Science & Informatics

(Content adapted from slides by Dr Bailin Deng)

Data structures

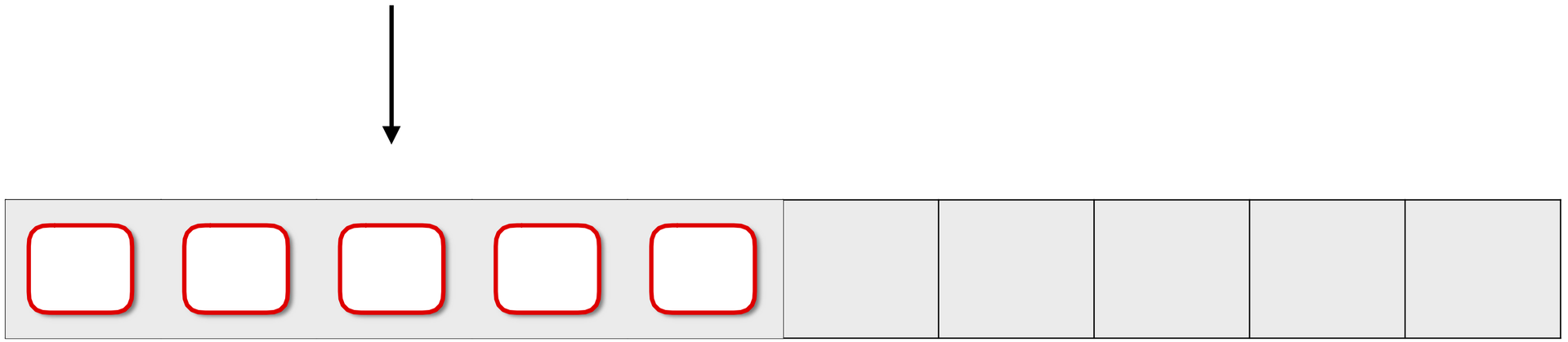
- Singly Linked List
- Queues
- Stacks

This Week

- Singly Linked List
  - In a singly Linked List we can perform the following operations
    - Traversing
    - Inserting
    - Deleting

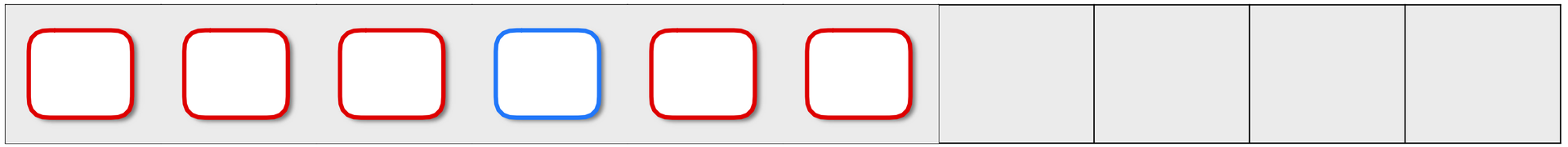
Array ADT

- O(1) time complexity for random access because of contiguous storage in memory



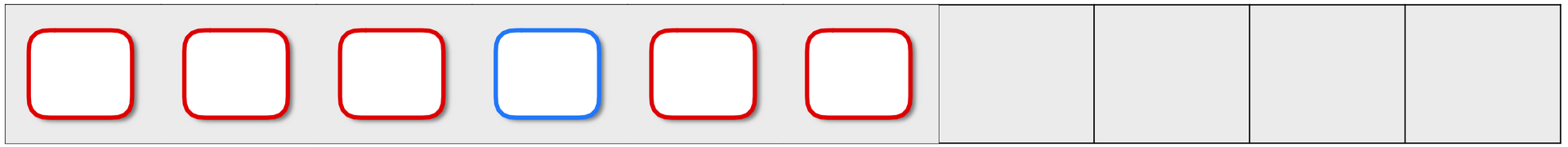
Array ADT

- Average O(n) complexity for inserting/deleting an element



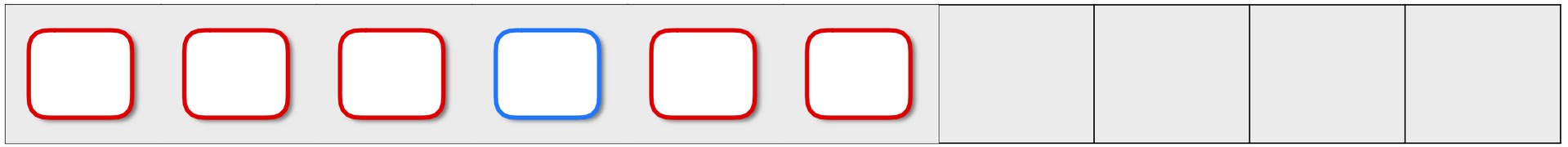
Array ADT

- Average O(n) complexity for inserting/deleting an element
  - Need to move O(n) elements to keep the storage contiguous



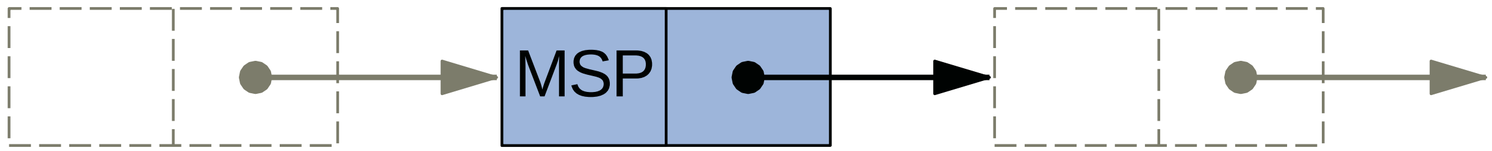
Array ADT

- Average O(n) complexity for inserting/deleting an element
  - Need to move O(n) elements to keep the storage contiguous
  - To make insertion/deletion more efficient, we need to give up the requirement of contiguous storage



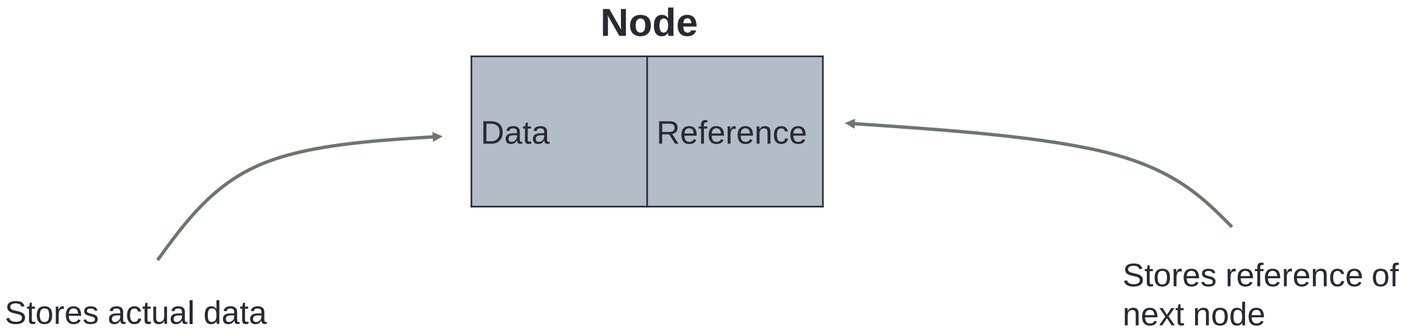
Singly Linked Lists

- A sequence of nodes, not necessarily stored on contiguous memory
- Each node stores an element, and a reference to the next node



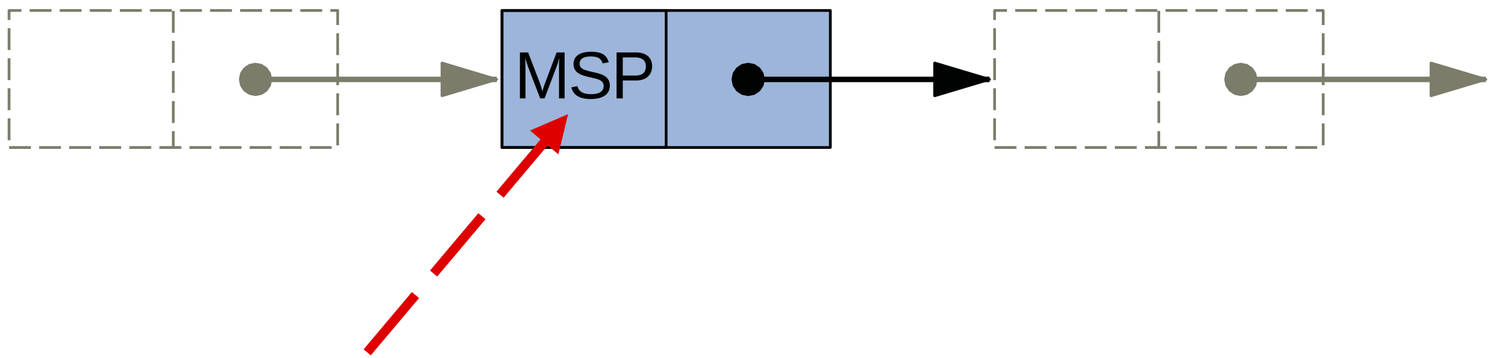
Singly Linked Lists

- Every Node contains two fields: data and reference.
  - The data field is used to store actual value of that node
  - The reference/next field is used to store the address/reference of the next node in the sequence.



Singly Linked Lists

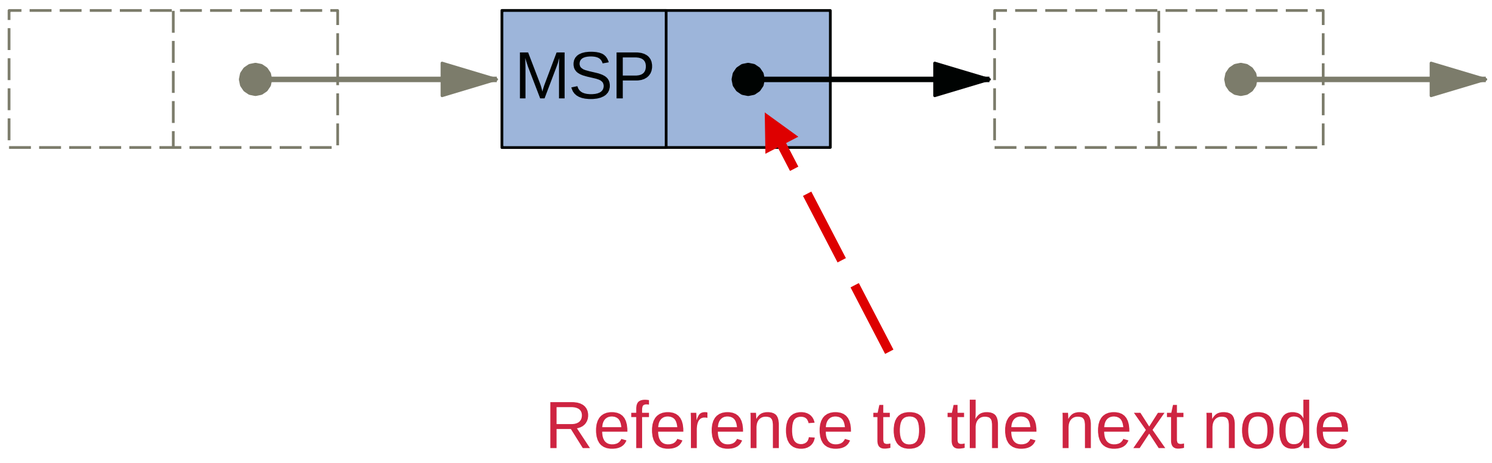
- A sequence of nodes, not necessarily stored on contiguous memory
- Each node stores an element, and a reference to the next node



An element (the actual content of the list, here an airport code)

Singly Linked Lists

- A sequence of nodes, not necessarily stored on contiguous memory
- Each node stores an element, and a reference to the next node



Singly Linked Lists

- Example code for the Node class

```
public class Node
{
   private String airportCode;
   private Node next;

   …
}
```

Singly Linked Lists

- Example code for the Node class

```
public class Node
{
   private String airportCode;
   private Node next;


   …
}
```

Element inside the node

Singly Linked Lists

- Example code for the Node class

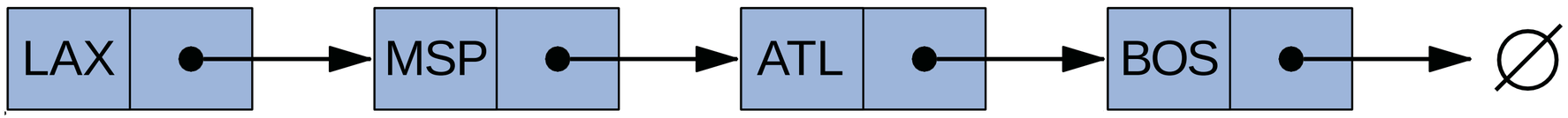
```
public class Node
{
   private String airportCode;
   private Node next;

   …
}
```

Reference to the next node

Singly Linked List

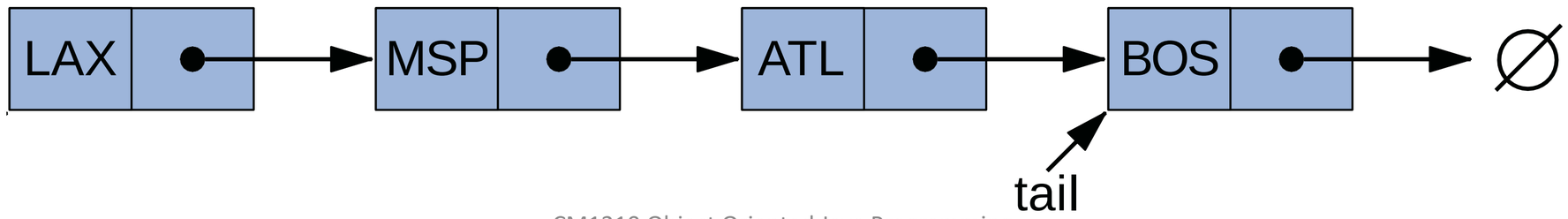
- A linked list consists of a sequence of nodes

Example: a singly linked list of airport codes



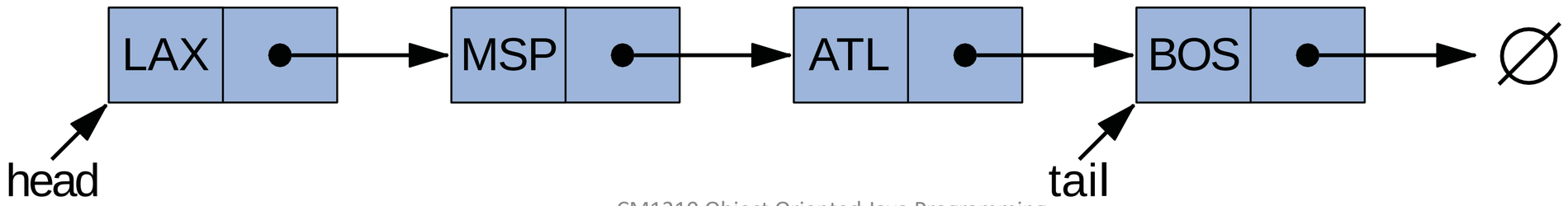
Singly Linked List

- A linked list consists of a sequence of nodes
  - For the last node (“tail”), its reference to the next node is `null`



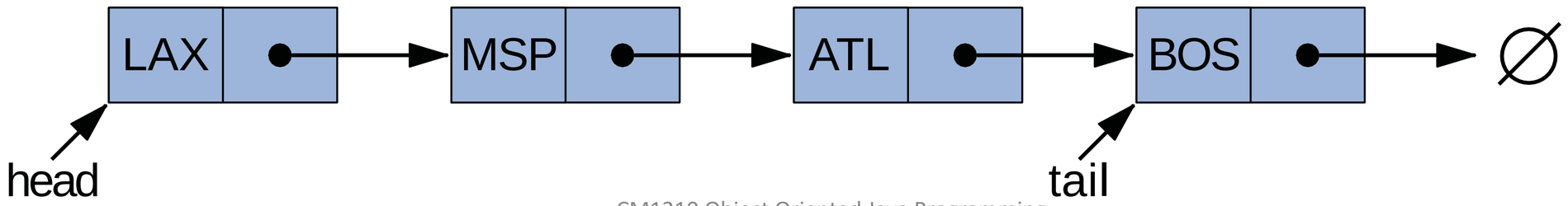
Singly Linked List

- A linked list consists of a sequence of nodes
  - For the last node (“tail”), its reference to the next node is `null`
  - Must store reference to the first node (“head”)



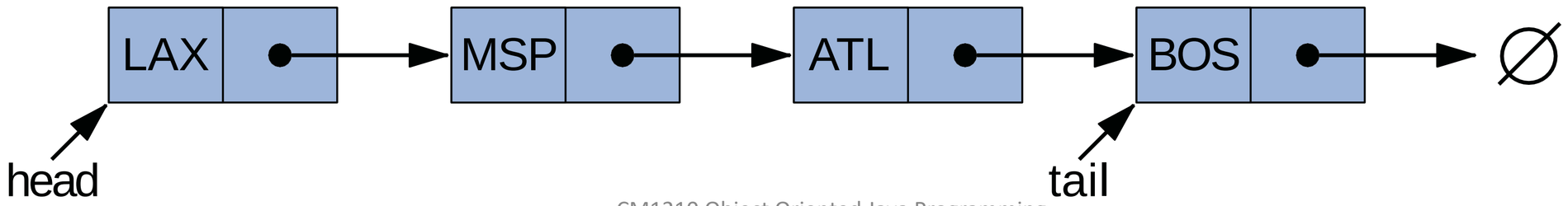
Singly Linked List

- A linked list consists of a sequence of nodes
  - For the last node (“tail”), its reference to the next node is `null`
  - Must store reference to the first node (“head”)
  - Also store reference to the tail for convenience



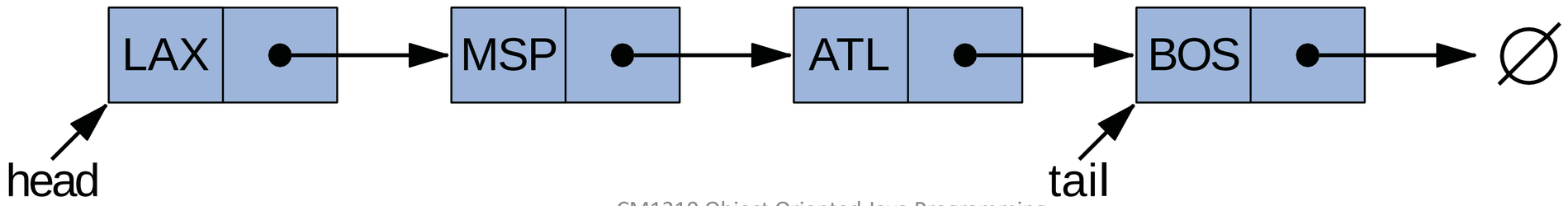
Singly Linked List

- A linked list consists of a sequence of nodes
  - For the last node (“tail”), its reference to the next node is `null`
  - Must store reference to the first node (“head”)
  - Also store reference to the tail for convenience
  - Store the number of nodes (“size”)



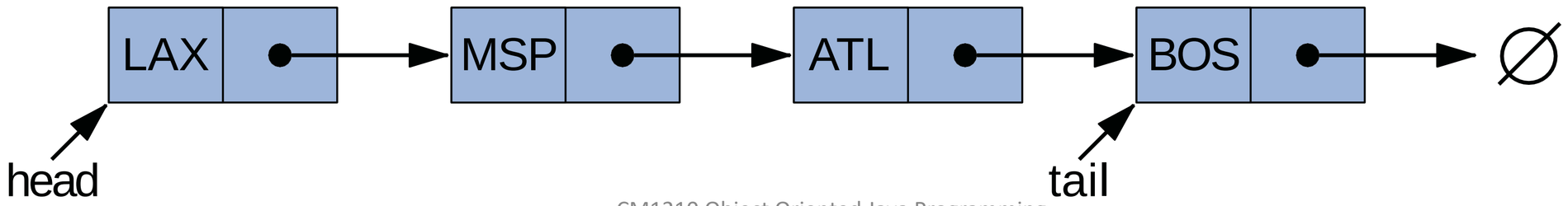
Singly Linked List

- A linked list consists of a sequence of nodes
  - For the last node (“tail”), its reference to the next node is `null`
  - Must store reference to the first node (“head”)
  - Also store reference to the tail for convenience
  - Store the number of nodes (“size”)
  - Empty list indicated by `head == null` or `size == 0`



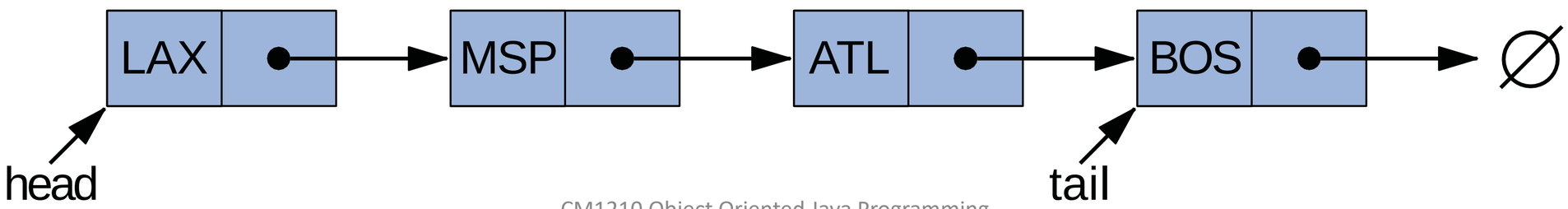
Singly Linked List

```
public class SinglyLinkedList
{
   private Node head;
   private Node next;
   private int size;

   …
}
```



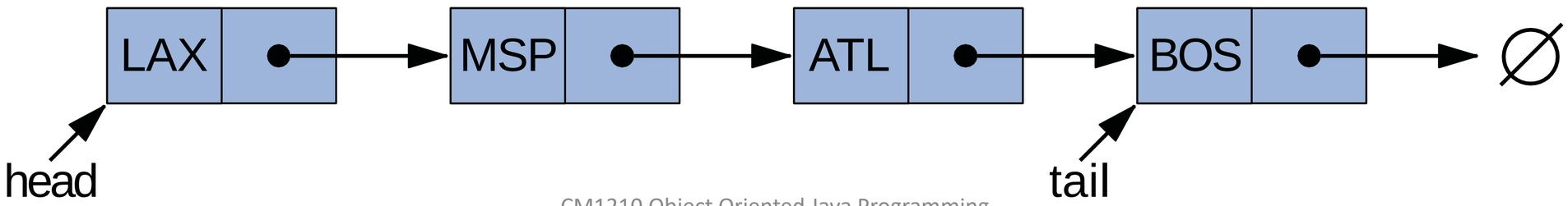
Traversing a Singly Linked List

- We can go through all nodes of the list starting from its head



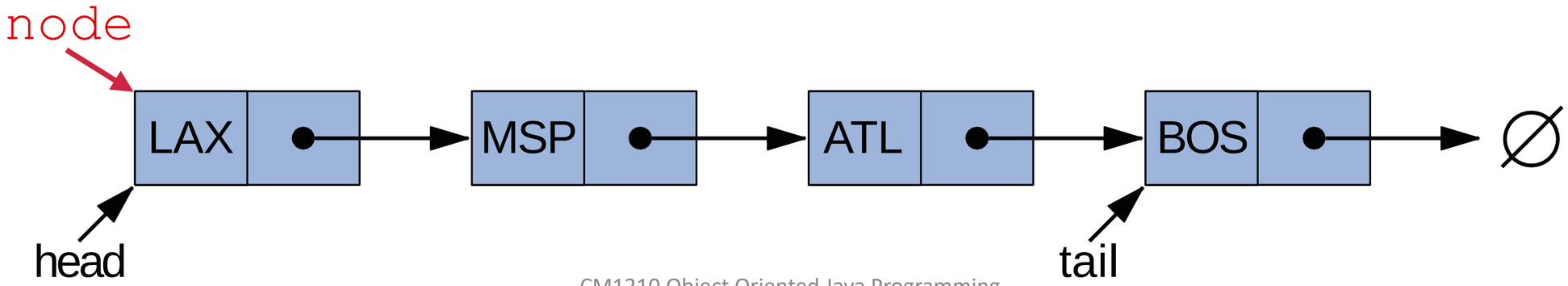
Traversing a Singly Linked List

```
Node node = head;
while(node != null)
{
  … //do something here

  node = node.next;
}
```



Traversing a Singly Linked List

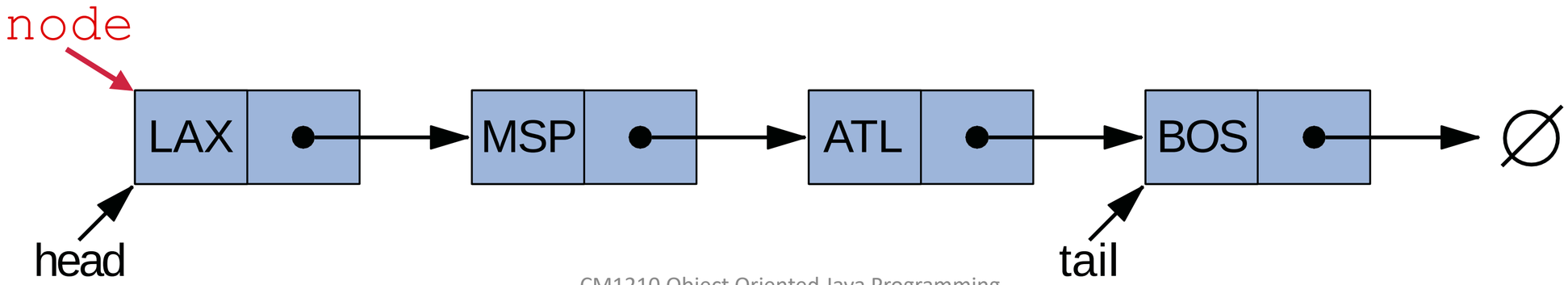
```
Node node = head;
while(node != null)
{
   … //do something here

   node = node.next;
}
```



Traversing a Singly Linked List

```
Node node = head;
while(node != null)
{
   … //do something here

   node = node.next;
}
```

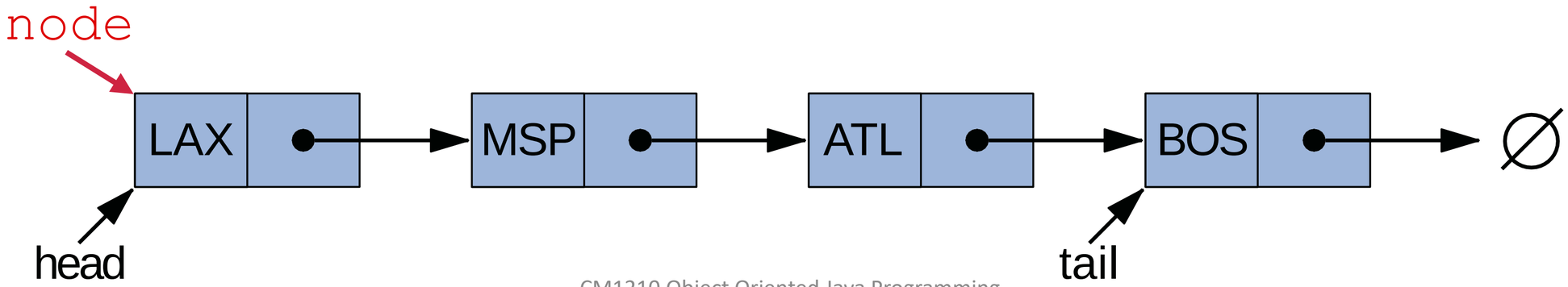
Exit if list is empty



Traversing a Singly Linked List

```
Node node = head;
while(node != null)
{
  … //do something here

  node = node.next;
}
```

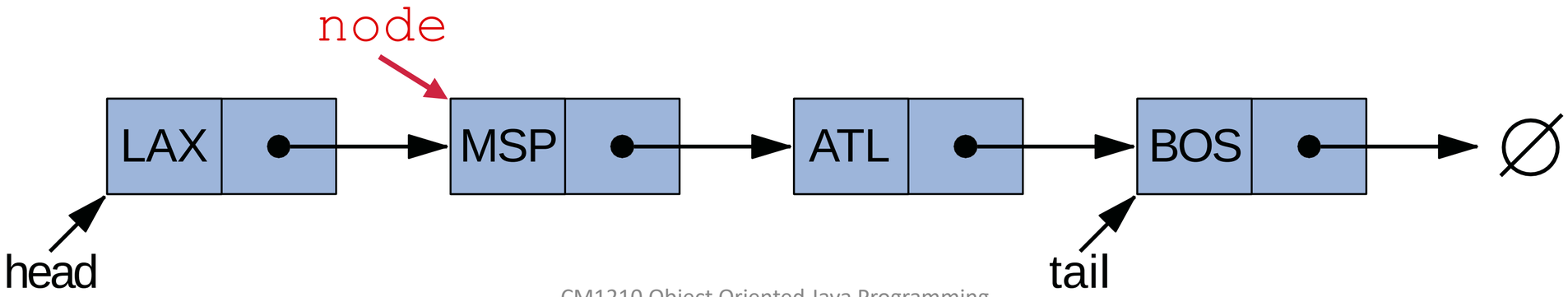
Application-specific processing of current node



Traversing a Singly Linked List

```
Node node = head;
while(node != null)
{
   … //do something here

   node = node.next;
}
```

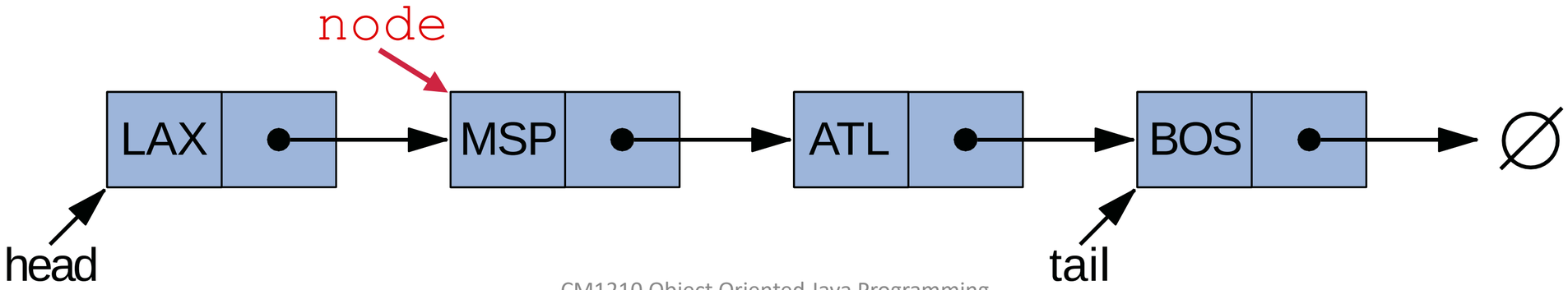
Hop to the next node by following the "`next`" reference



Traversing a Singly Linked List

```
Node node = head;
while(node != null)
{
   … //do something here

   node = node.next;
}
```

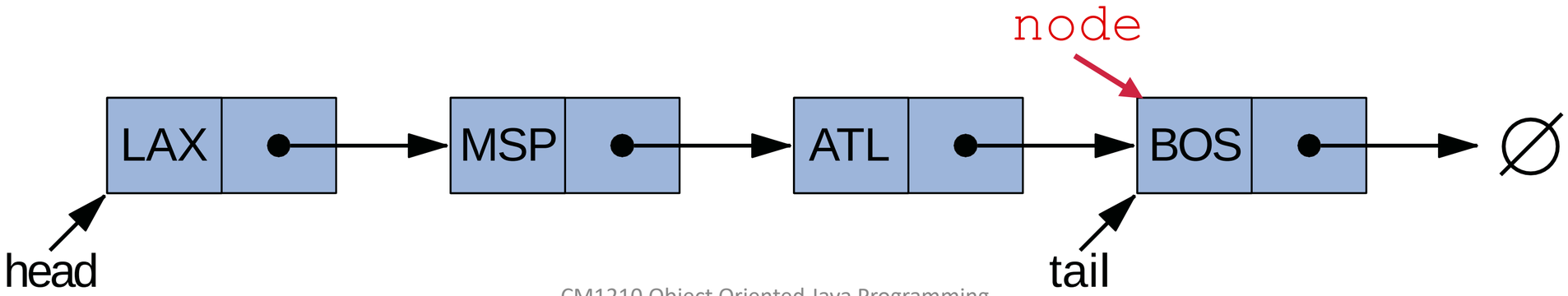
Continue to check existence of current node and process it



Traversing a Singly Linked List

```
Node node = head;
while(node != null)
{
   … //do something here

   node = node.next;
}
```

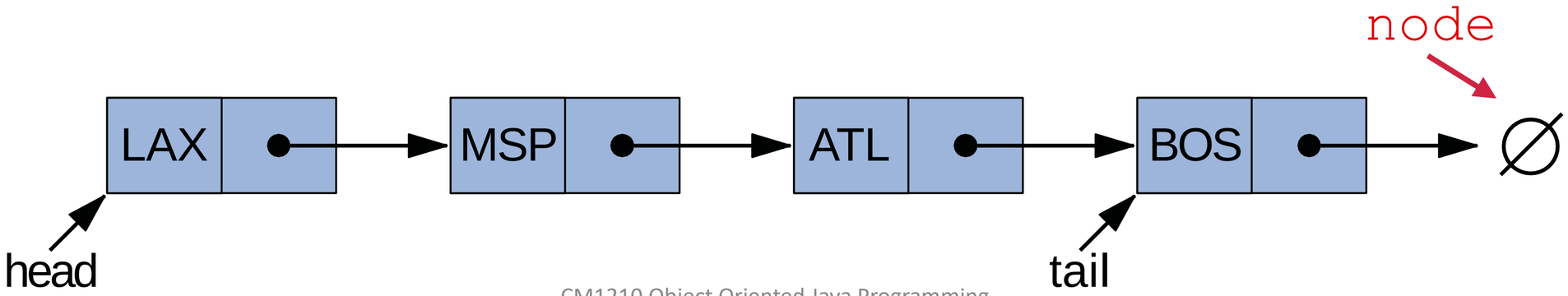
After processing the tail, node is set to null



Traversing a Singly Linked List

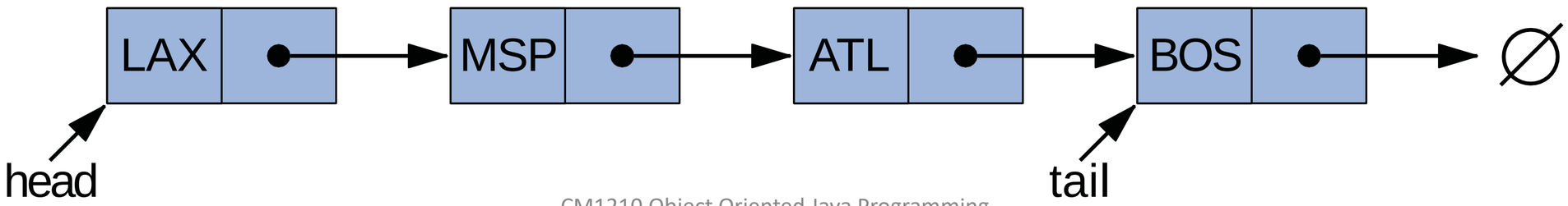
```
Node node = head;
while(node != null)
{
   … //do something here

   node = node.next;
}
```

No more node to process, exit the loop

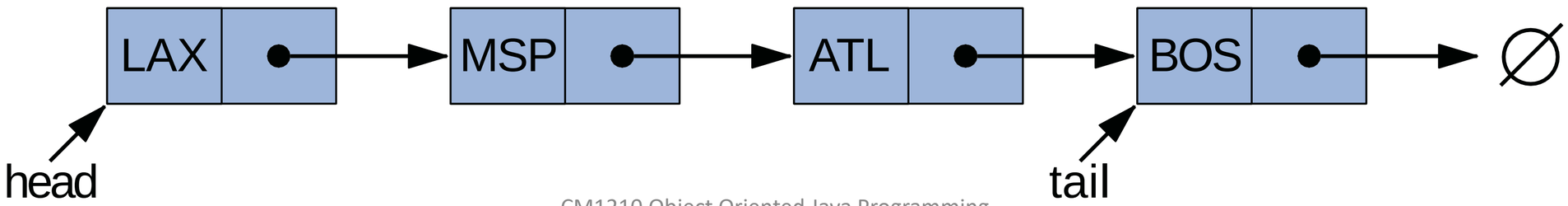


No Random Access

- How do we access the i-th node?
  - No contiguous memory storage: cannot directly compute its address
  - Have to traverse the list from the head

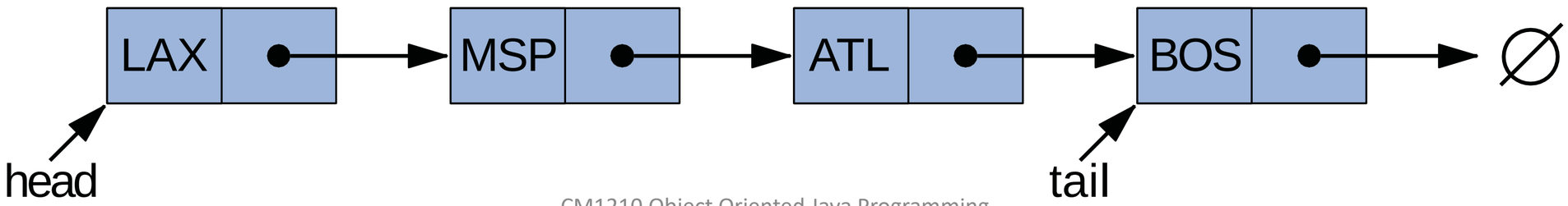


```
Node getNode(int i)
{
   if(i < 0 || i >= size)
      return null;

   Node node = head;
   int count = 0;
   while(count < i){
      node = node.next;
      count++;
   }

   return node;
}
```



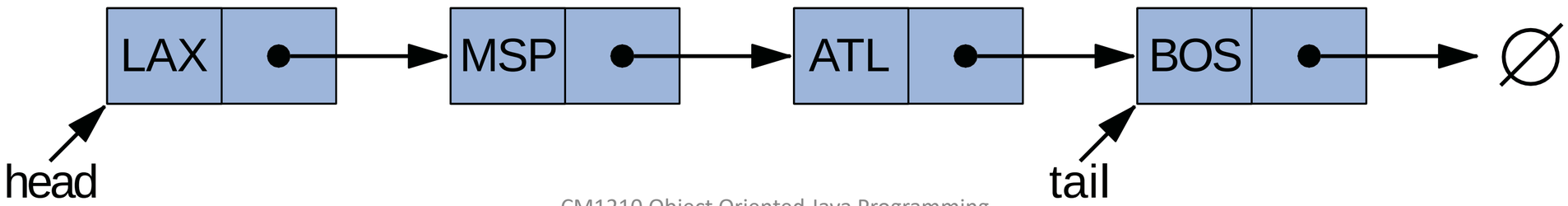
```
Node getNode(int i)
{
   if(i < 0 || i >= size)
      return null;

   Node node = head;
   int count = 0;
   while(count < i){
      node = node.next;
      count++;
   }

   return node;
}
```

0-based “index" of element to access



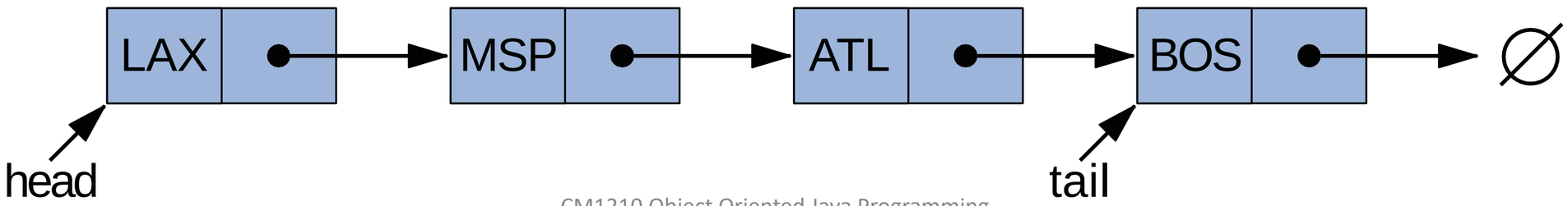
```
Node getNode(int i)
{
   if(i < 0 || i >= size)
     return null;

   Node node = head;
   int count = 0;
   while(count < i){
      node = node.next;
      count++;
   }

   return node;
}
```

Sanity check of index



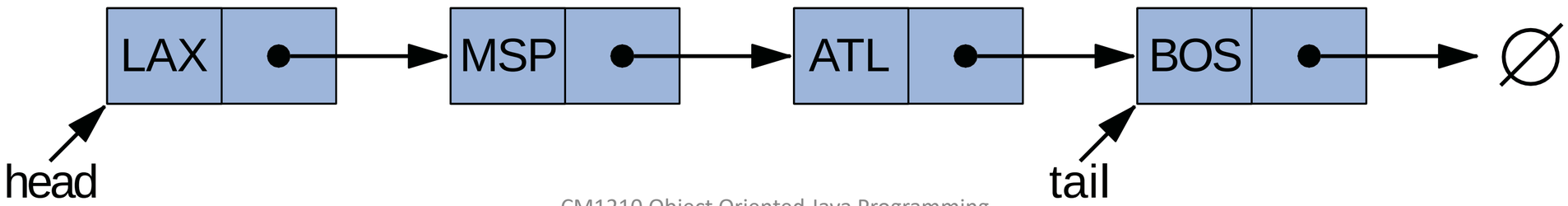
```
Node getNode(int i)
{
  if(i < 0 || i >= size)
    return null;

  Node node = head;
  int count = 0;
  while(count < i){
     node = node.next;
     count++;
  }

  return node;
}
```

Start from the head, initialise index counter



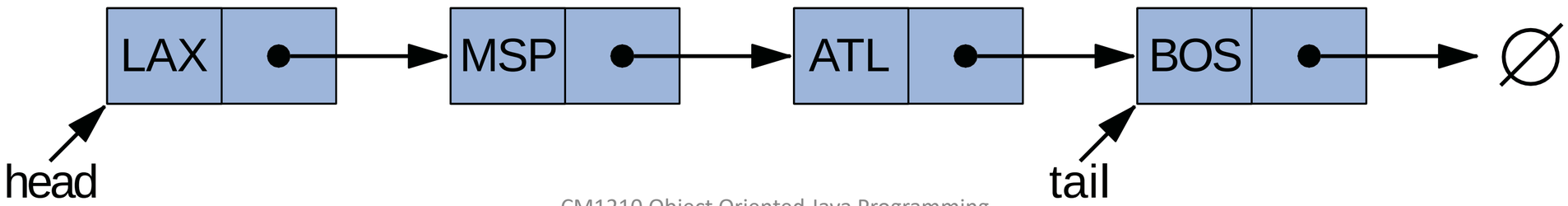
```
Node getNode(int i)
{
  if(i < 0 || i >= size)
    return null;

  Node node = head;
  int count = 0;
  while(count < i){
     node = node.next;
     count++;
  }

  return node;
}
```

Hop to the next node and increase the index counter, unless we reach node *i*



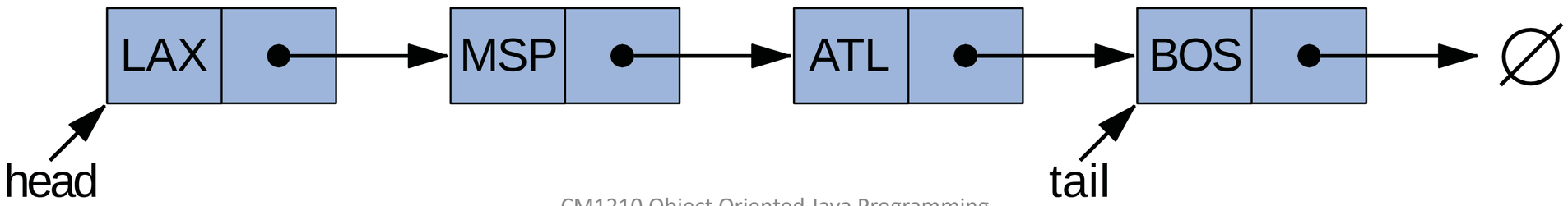
```
Node getNode(int i)
{
   if(i < 0 || i >= size)
      return null;

   Node node = head;
   int count = 0;
   while(count < i){
      node = node.next;
      count++;
   }

   return node;
}
```

Return node *i*



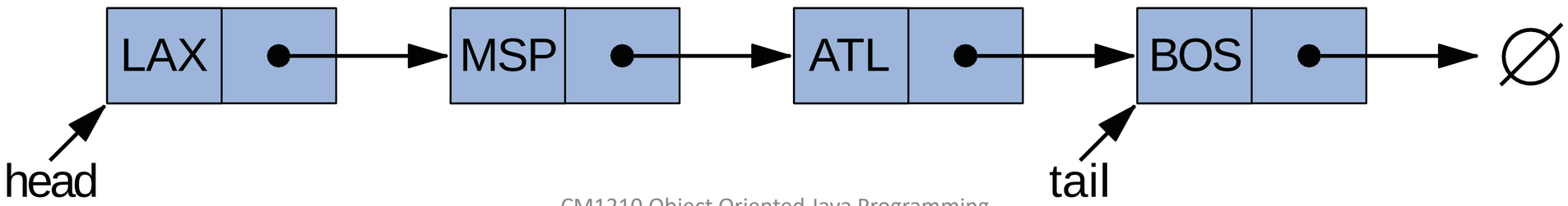
Time Complexity

- To access the node with “index” *i*, we must hop for *i* times
  - Worst case complexity O(n)
  - Average complexity O(n)



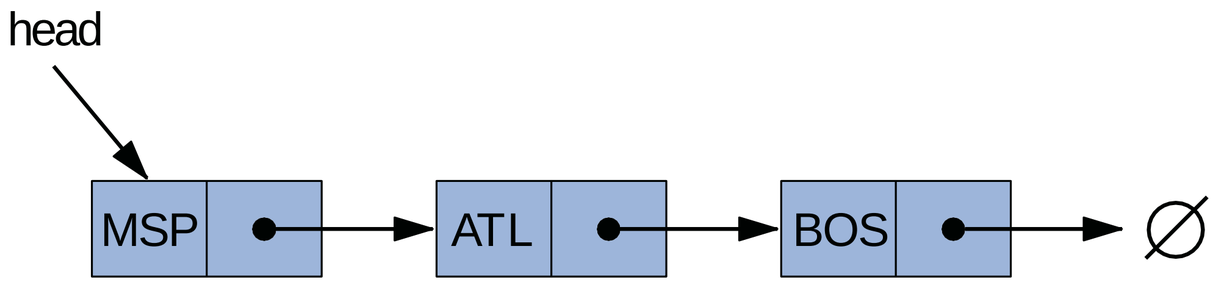
Time Complexity

- To access the node with “index” *i*, we must hop for *i* times
  - Worst case complexity O(n)
  - Average complexity O(n)
- Inefficient for accessing an arbitrary element



Adding an element

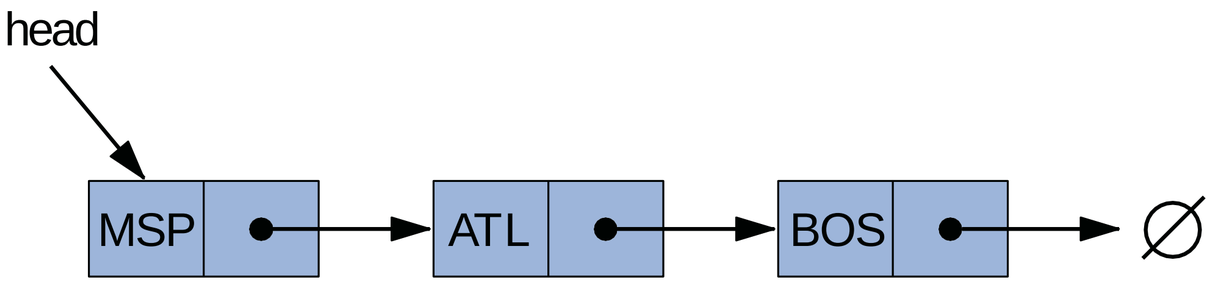
- How do we add an element to the beginning of the list?



Adding an element

- Pseudocode:

```
Algorithm addFirst(e):
    newest = Node(e)
    newest.next = head
    head = newest
    size = size + 1
```

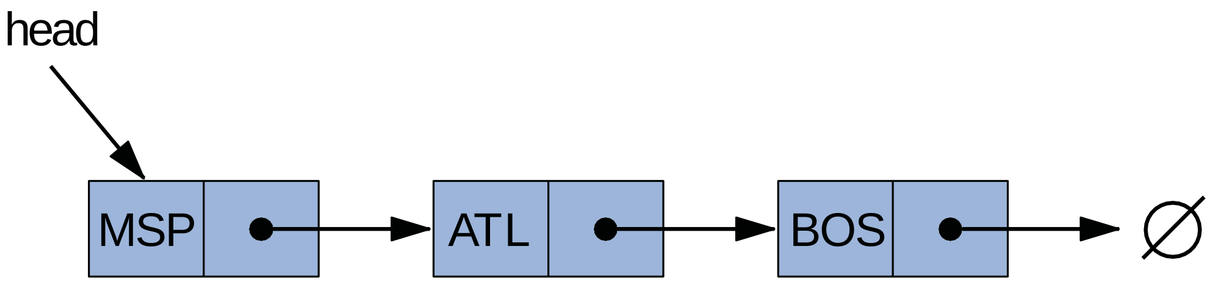


Adding an element

- Pseudocode:

Element to be added

```
Algorithm addFirst(e):
    newest = Node(e)
    newest.next = head
    head = newest
    size = size + 1
```



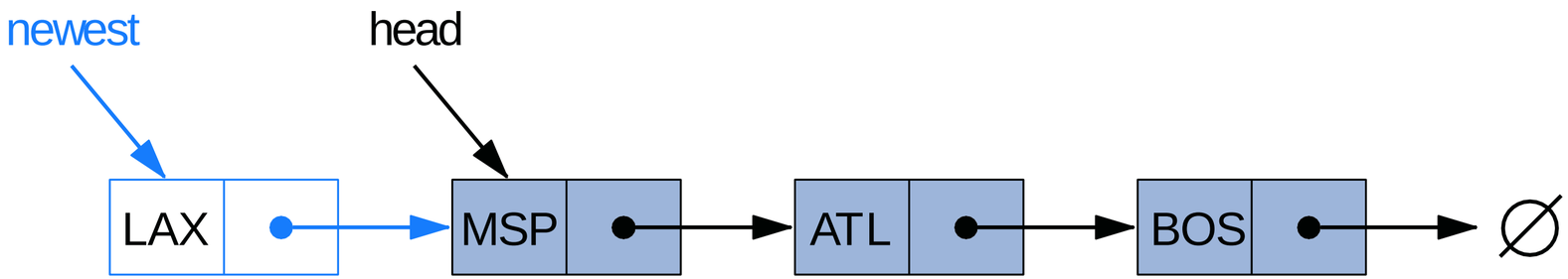
Adding an element

- Pseudocode:

Algorithm addFirst(e):

```
newest = Node(e)
newest.next = head
head = newest
size = size + 1
```

Create a new node for the element, set its next reference to the current head

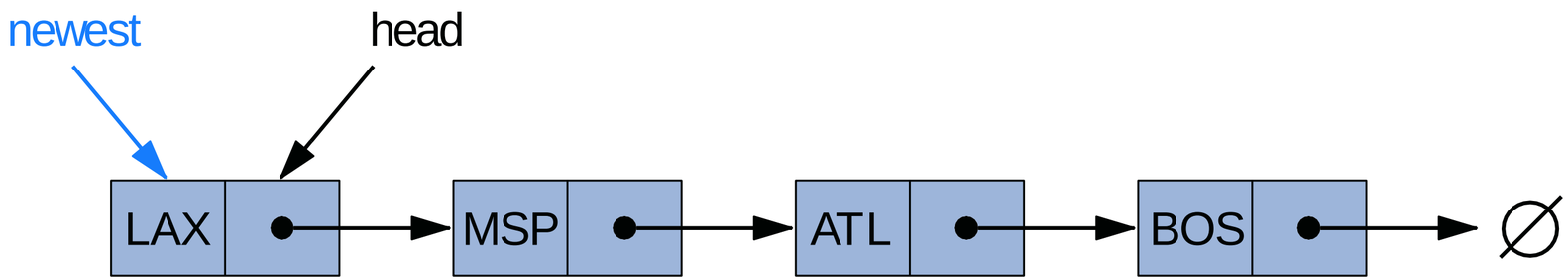


Adding an Element

- Pseudocode:

Algorithm addFirst(e):
    newest = Node(e)
    newest.next = head
    head = newest
    size = size + 1

Update head and size

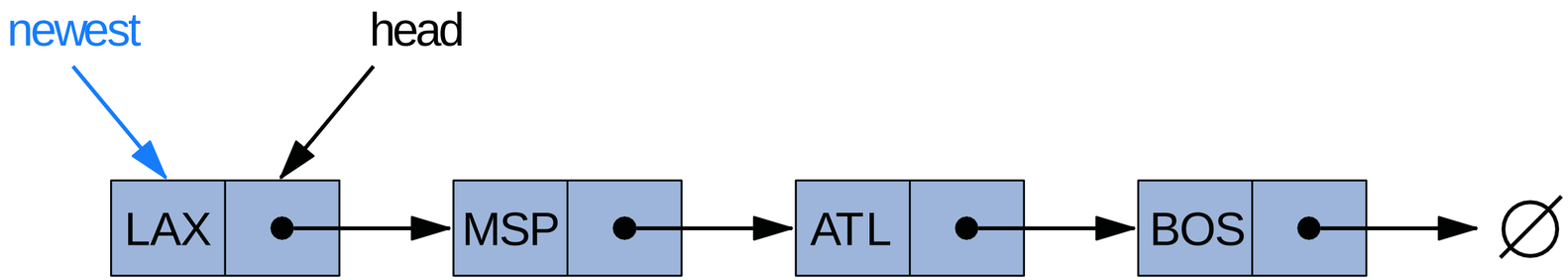


Adding an Element

- Pseudocode:

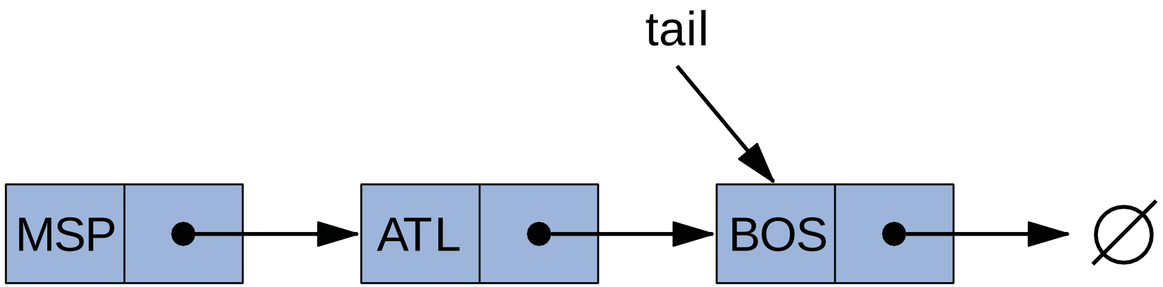
Algorithm addFirst(e):
    newest = Node(e)
    newest.next = head
    head = newest
    size = size + 1

Time complexity: O(1)



Adding an Element

- How to add an element at the back of a list?



Adding an Element

Algorithm addLast(e):

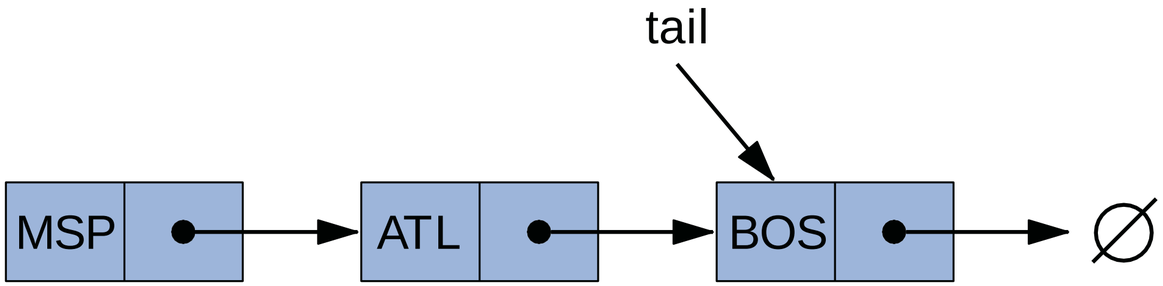
newest = Node(e)

newest.next = null

tail.next = newest

tail = newest

size = size + 1

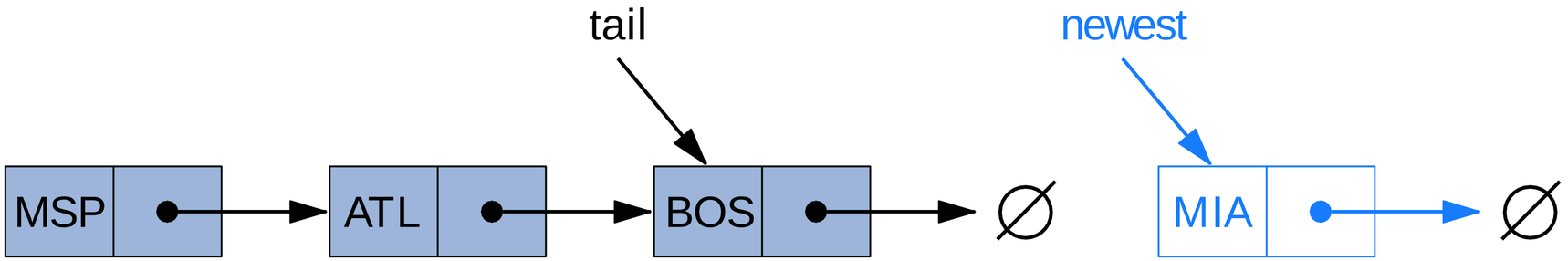


Adding an Element

Algorithm addLast(e):

```
newest = Node(e)
newest.next = null
tail.next = newest
tail = newest
size = size + 1
```

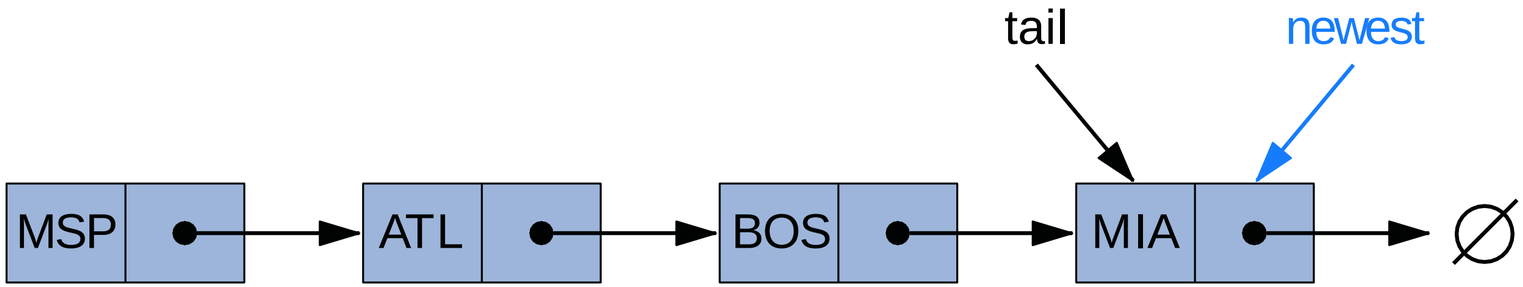
Create a new node for the element, set its `next` reference to null



Adding an Element

Algorithm addLast(e):
    newest = Node(e)
    newest.next = null
    tail.next = newest
    tail = newest
    size = size + 1

Update the `next` reference of the current tail to the new node, set the new node as tail



Adding an Element

Algorithm addLast(e):

newest = Node(e)

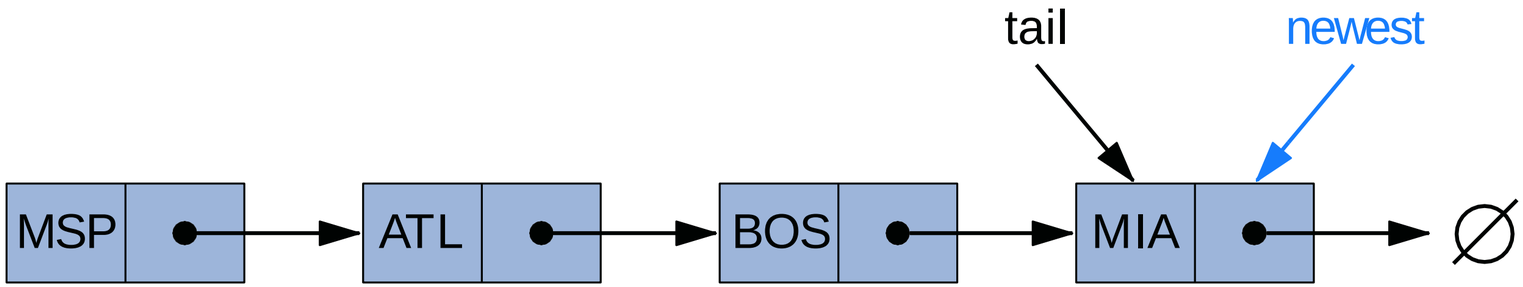
newest.next = null

tail.next = newest

tail = newest

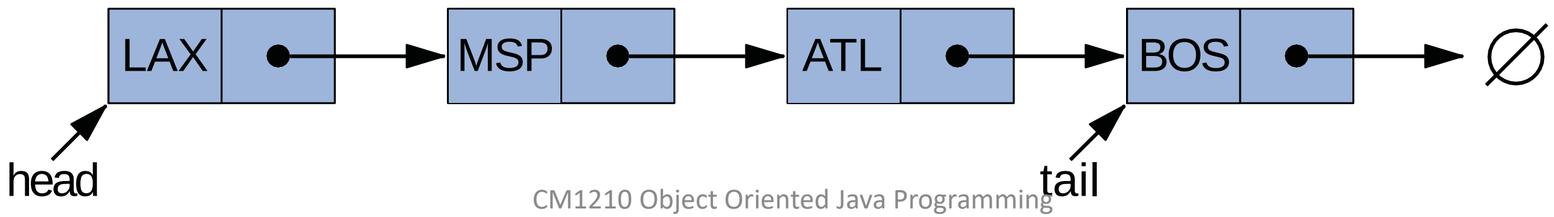
size = size + 1

Time complexity: O(1)



Adding an Element

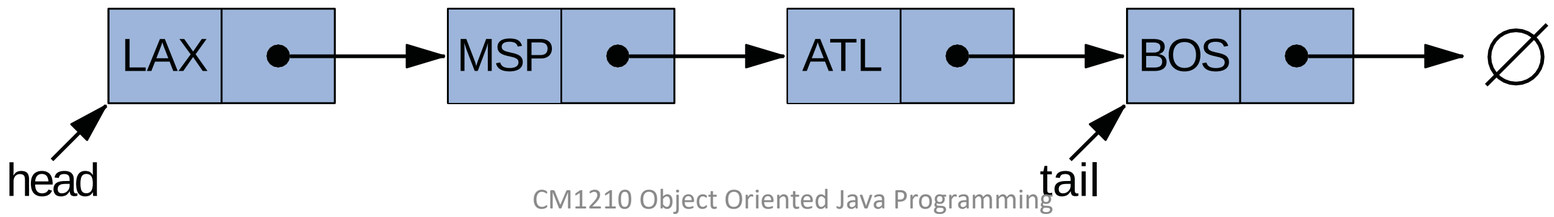
- Add an element *e* after node *n*



Adding an Element

- Add an element *e* after node *n*

Algorithm addAfter(n, e):
    newest = Node(e)
    newest.next = n.next
    n.next = newest
    size = size + 1

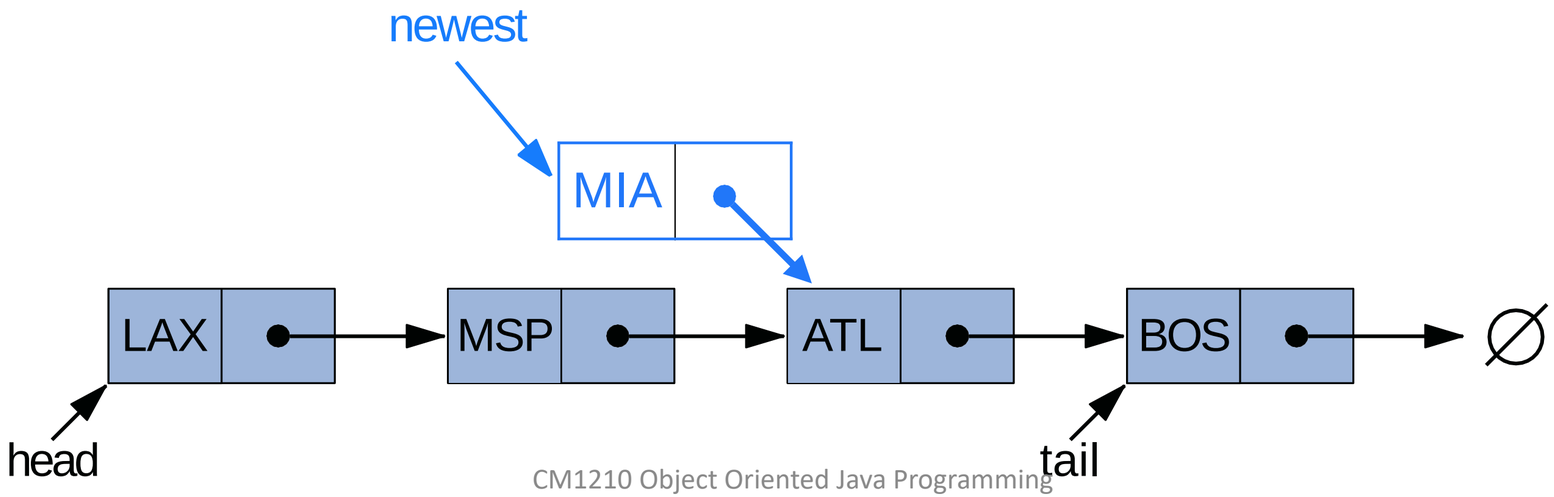


Adding an Element

- Example: add an element “MIA” after node “MSP”

Algorithm addAfter(n, e):
```
newest = Node(e)
newest.next = n.next
n.next = newest
size = size + 1
```

Create a new node for the element, set its `next` reference to the node after n

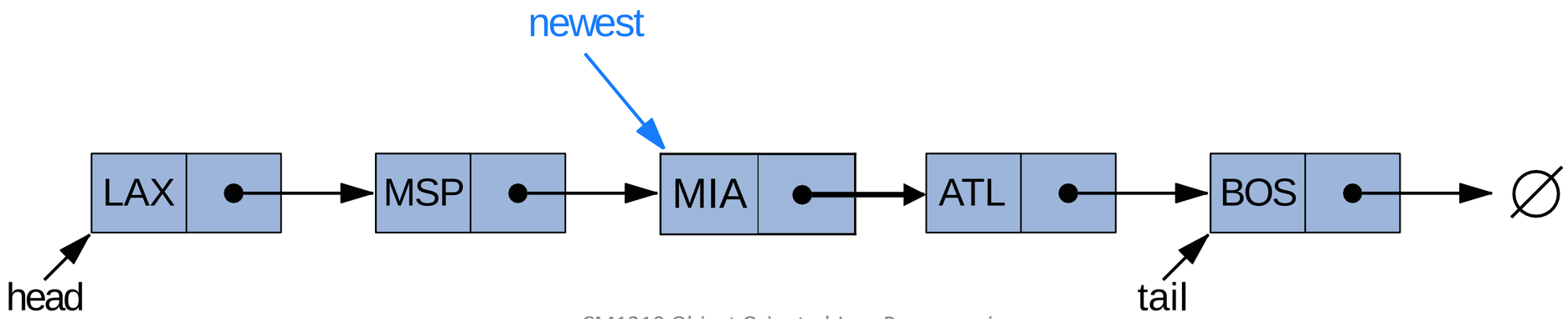


Adding an Element

- Example: add an element “MIA” after node “MSP”

Algorithm addAfter(n, e):
    newest = Node(e)
    newest.next = n.next
    n.next = newest
    size = size + 1

Update the `next` reference in n to the new node, and update size

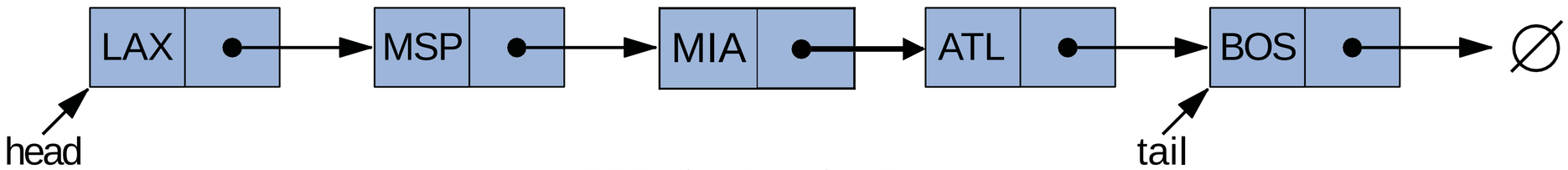


Adding an Element

- Example: add an element “MIA” after node “MSP”

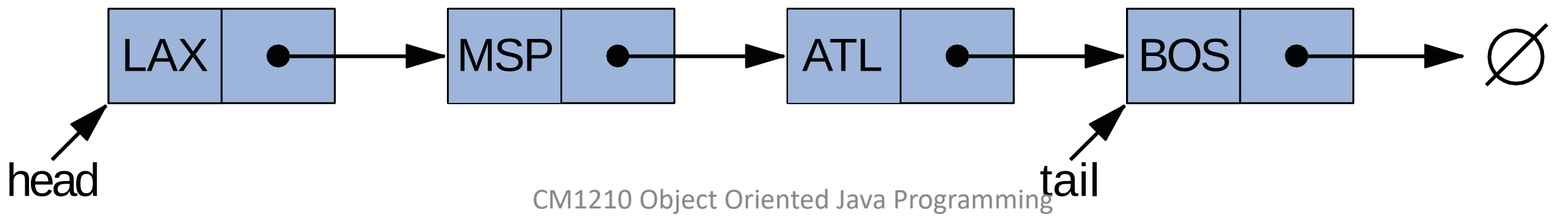
Algorithm addAfter(n, e):
    newest = Node(e)
    newest.next = n.next
    n.next = newest
    size = size + 1

Time complexity: O(1)



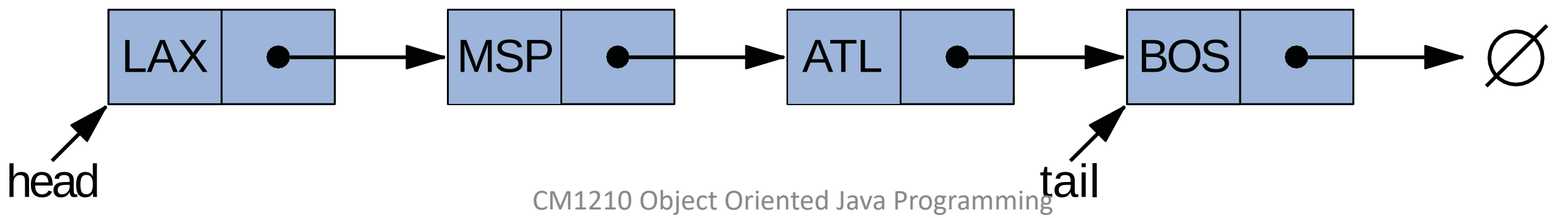
Adding an Element

- Add an element *e* **before** node *n*



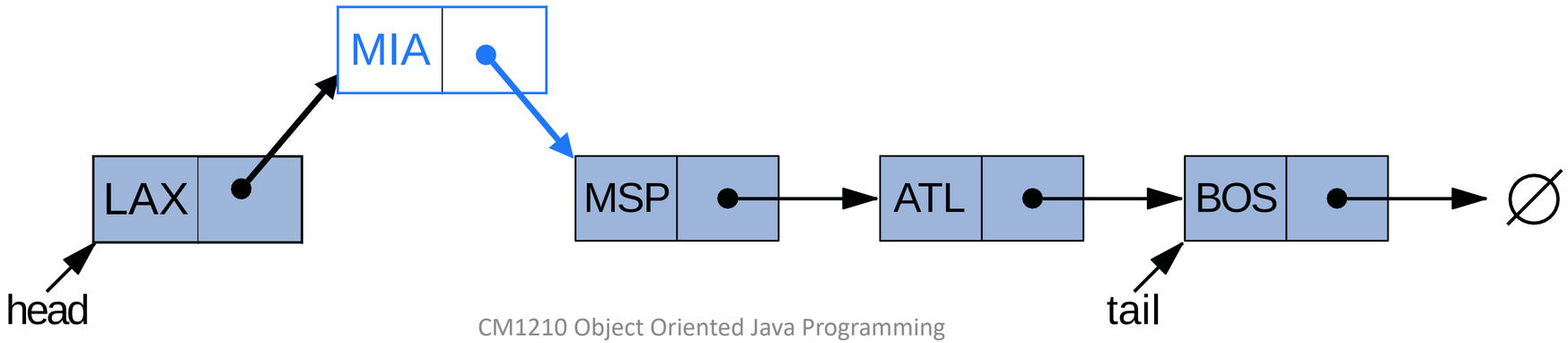
Adding an Element

- Add an element *e* **before** node *n*
  - If n is the head, simply add e to the beginning of the list



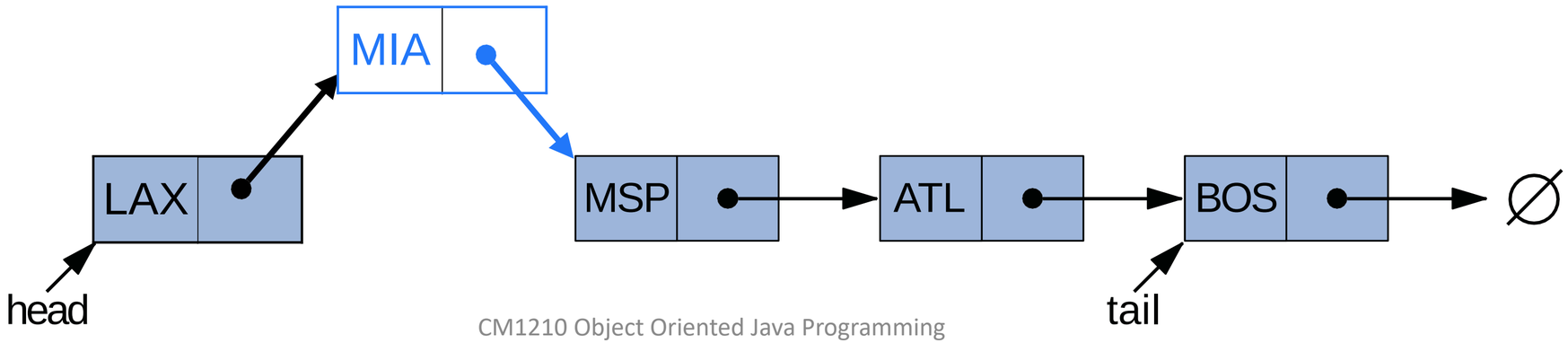
Adding an Element

- Add an element *e* **before** node *n*
  - If n is not the head, we must update the node before n, and point its `next` reference to the new node



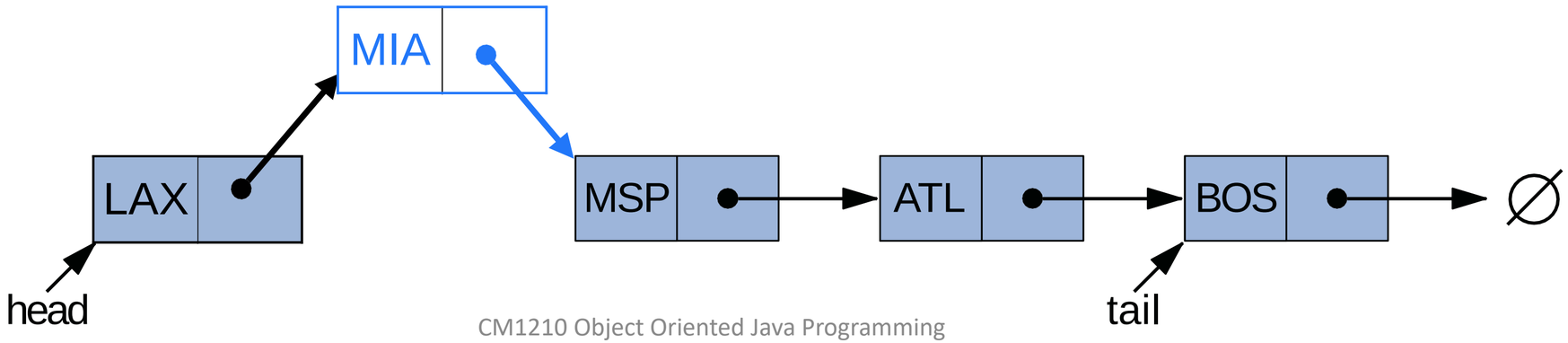
Adding an Element

- Add an element *e* **before** node *n*
  - If n is not the head, we must update the node before n, and point its `next` reference to the new node
  - However, from node n alone we do not know its preceding node
  - We must traverse the list from head to locate the preceding node



Adding an Element

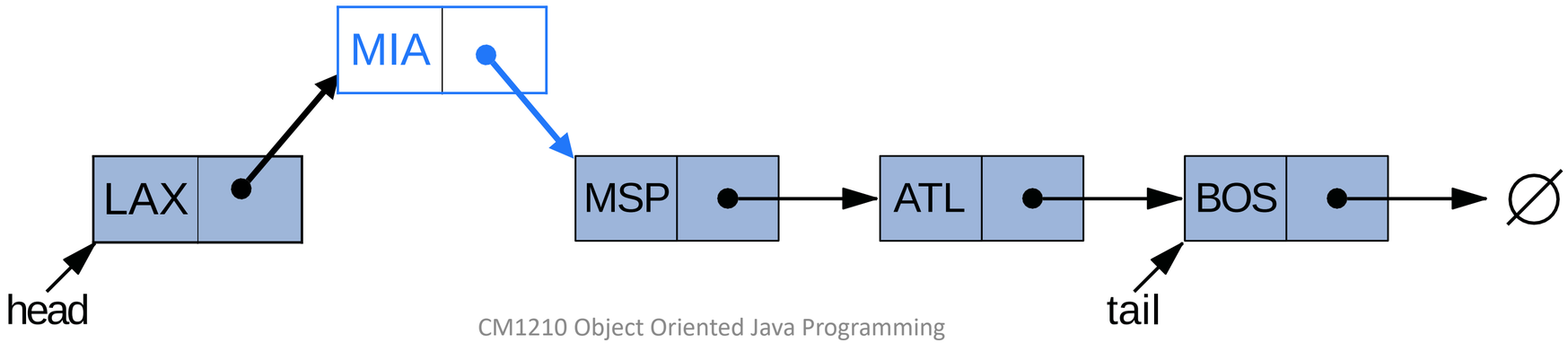
```
Algorithm addBefore(n, e):
        if (n == head) then
        addFirst(e)
        else {
         p = precedingNode(n)
         addAfter(p, e)
        }
```



Adding an Element

```
Algorithm addBefore(n, e):
    if (n == head) then
      addFirst(e)
    else {
      p = precedingNode(n)
      addAfter(p, e)
    }
```

If n is the head, add e to the beginning

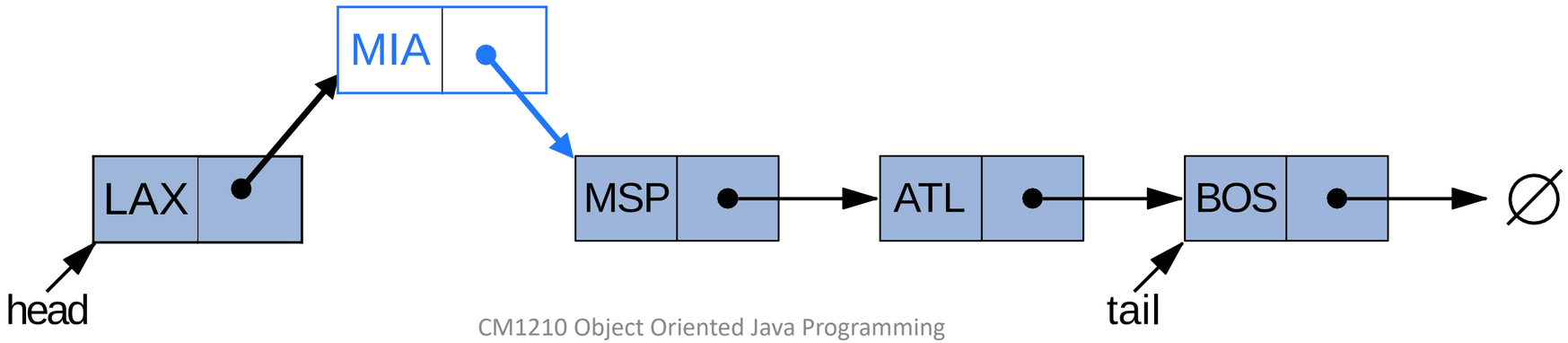


Adding an Element

Algorithm addBefore(n, e):
if (n == head) **then**
addFirst(e)

```
else {
  p = precedingNode(n)
  addAfter(p, e)
}
```

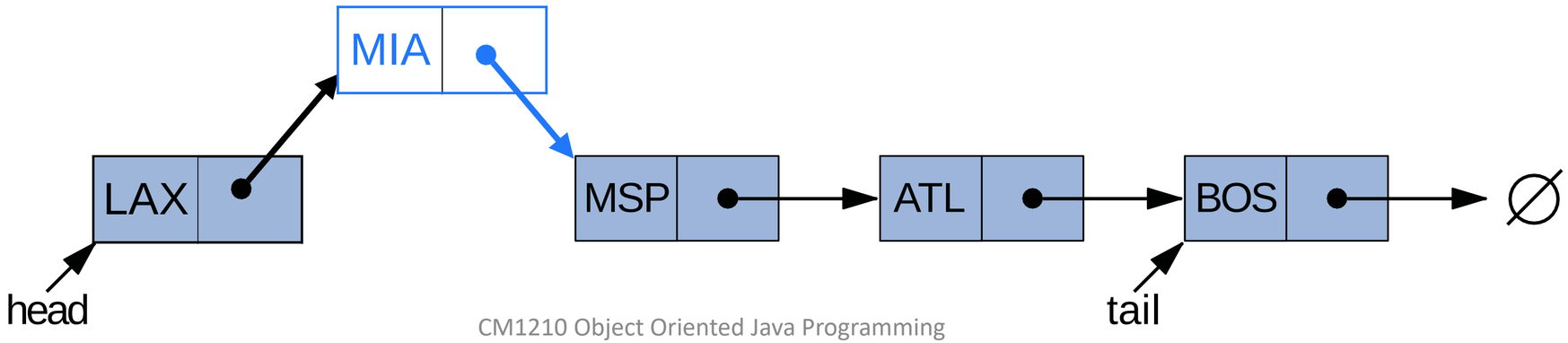
Otherwise: search for node p before n, add e after p



Adding an Element

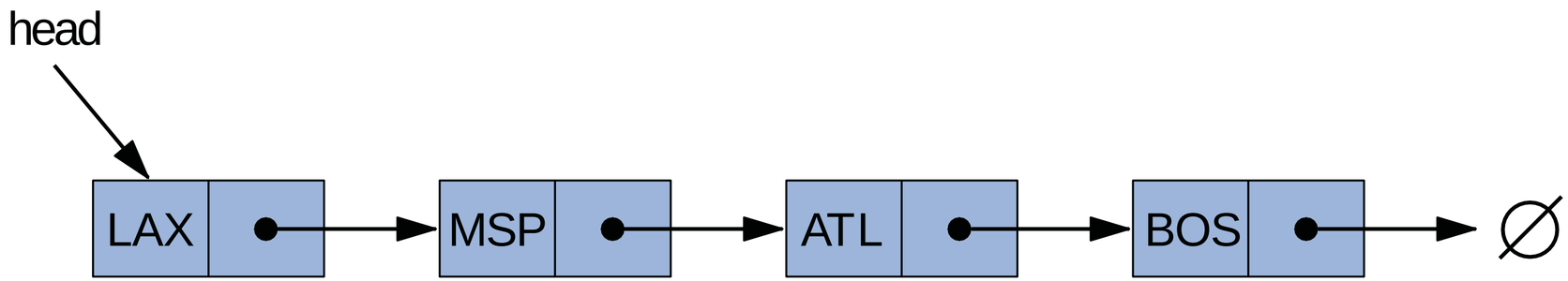
```
Algorithm addBefore(n, e):
        if (n == head) then
        addFirst(e)
        else {
         p = precedingNode(n)
          addAfter(p, e)
        }
```

Worst Case Time complexity: O(n)



Removing an Element

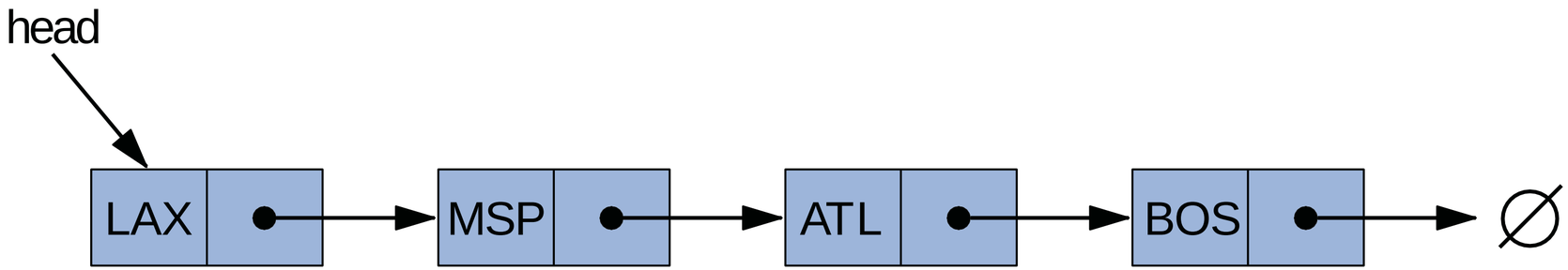
- Removing the element at the front



Removing an Element

- Removing the element at the front

Algorithm removeFirst():
if head == null **then**
the list is empty.
head = head.next
size = size − 1



Removing an Element

- Removing the element at the front

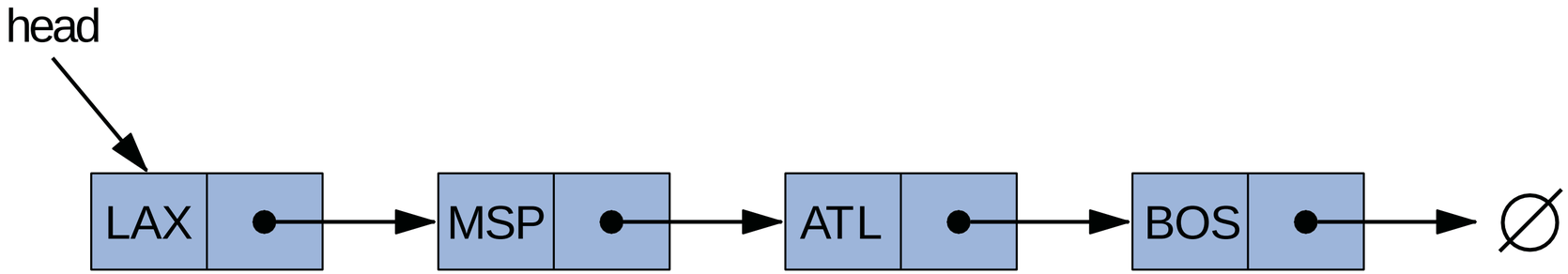
Algorithm removeFirst():

if head == null **then**
    the list is empty.

Sanity check

head = head.next

size = size − 1



Removing an Element

- Removing the element at the front

Algorithm removeFirst():

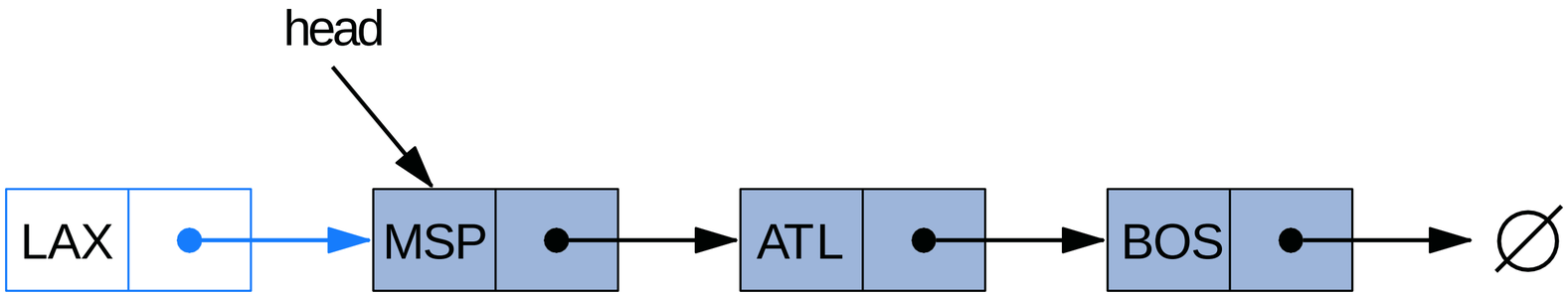
if head == null **then**

the list is empty.

head = head.next

size = size − 1

If the list is not empty, point the head to the next node

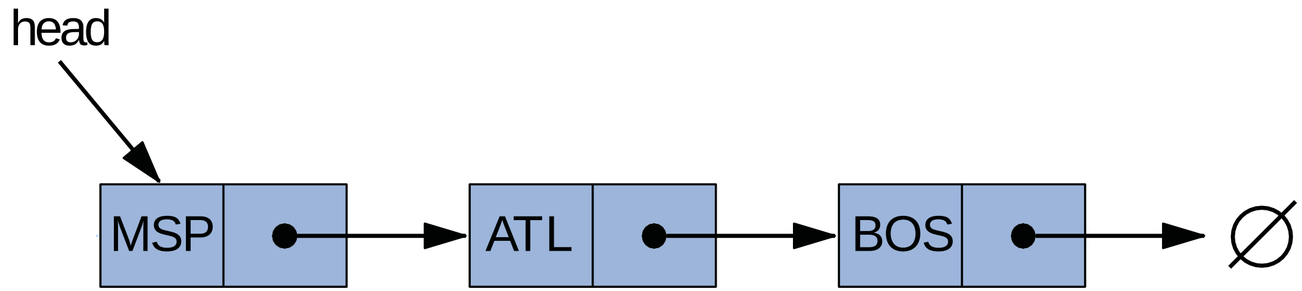


Removing an Element

- Removing the element at the front

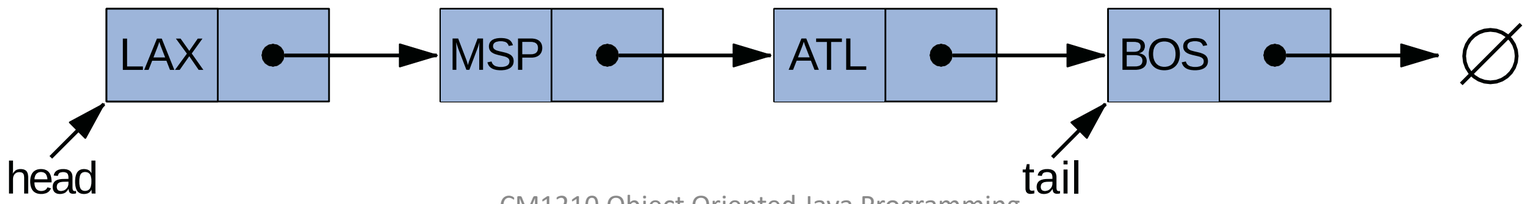
```
Algorithm removeFirst():
    if head == null then
        the list is empty.
    head = head.next
    size = size − 1
```

Time complexity: O(1)



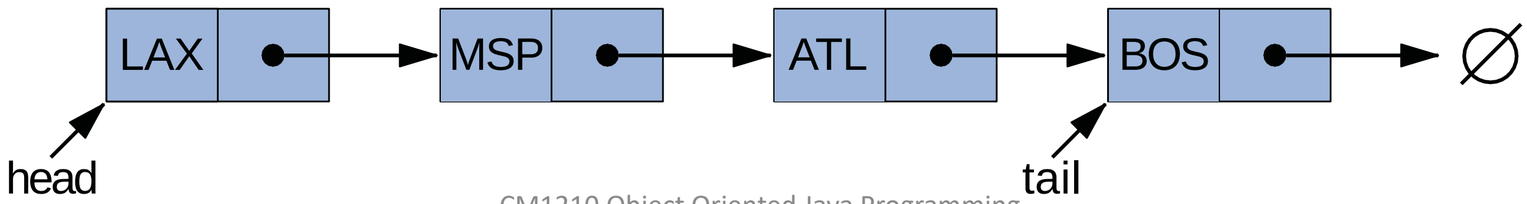
Removing an Element

- Removing the element at the back



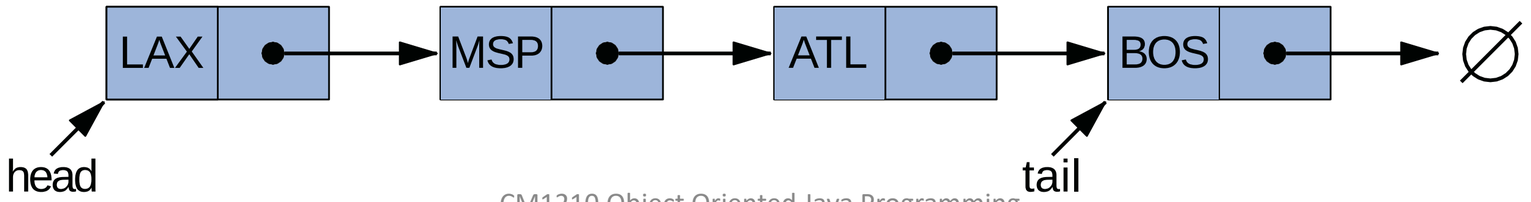
Removing an Element

- Removing the element at the back
  - If there is an element before tail, then we must set its `next` to `null`



Removing an Element

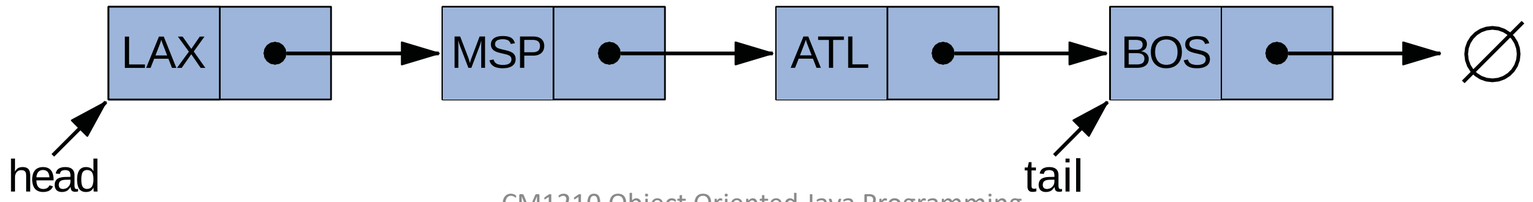
```
Algorithm removeLast():
        if (tail == head) then{
         tail = null, head = null
        }
        else {
         p = precedingNode(n)
         p.next = null
         tail = p
         size = size - 1
        }
```



Removing an Element

```
Algorithm removeLast():
    if (tail == head) then{
     tail = null, head = null
    }
    else {
     p = precedingNode(n)
     p.next = null
     tail = p
     size = size - 1
    }
```

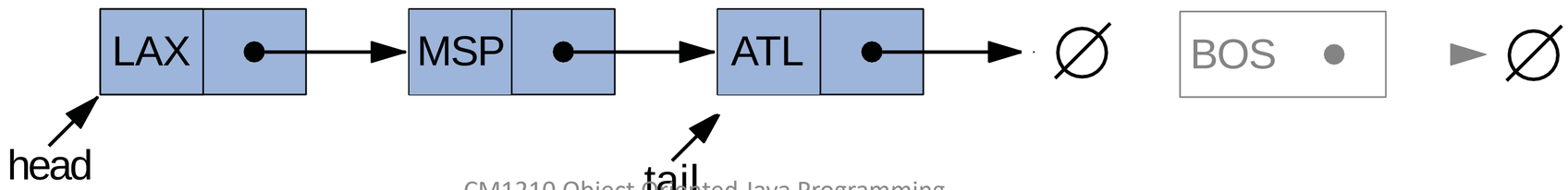
Sanity check for the case where the list is empty or has only one node



Removing an Element

```
Algorithm removeLast():
        if (tail == head) then{
         tail = null, head = null
        }
        else {
         p = precedingNode(n)
         p.next = null
         tail = p
         size = size - 1
        }
```

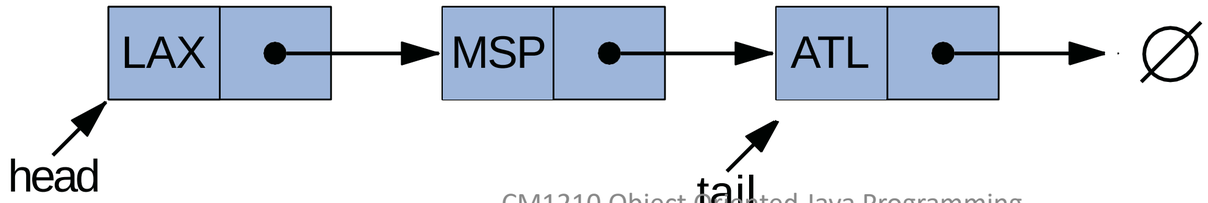
Update the preceding node's `next` reference; point tail to the preceding node



Removing an Element

```
Algorithm removeLast():
        if (tail == head) then{
         tail = null, head = null
        }
        else {
         p = precedingNode(n)
         p.next = null
         tail = p
         size = size - 1
        }
```

Time complexity: O(n)



Advantages of using LinkedList

- Dynamic Data Structure
  - Linked list is a dynamic data structure so it can grow and shrink at runtime by allocating and deallocating memory. So there is no need to give initial size of linked list.
- Insertion and Deletion
  - Insertion and deletion of nodes are really easier. Unlike array here it is not needed to shift elements after insertion or deletion of an element.
  - In linked list we just have to update the address present in next pointer of a node.
- No Memory Wastage
  - As size of linked list can increase or decrease at run time so there is no memory wastage.
    - memory is allocated only when required.
- Implementation
  - Data structures such as stack and queues can be easily implemented using linked list.

Disadvantages of using LinkedList

- Memory Usage
  - More memory is required to store elements in linked list as compared to array. Because in linked list each node contains data and a pointer and that requires extra memory.
- Traversal/Random access of elements
  - We can not randomly access any element as we do in array by index. For example if we want to access a node at position ***n*** then we have to traverse all the nodes before it. Time required to access a node is large.
- Reverse Traversing
  - In linked list reverse traversing is really difficult. In case of doubly linked list its easier but extra memory is required for back pointer hence wastage of memory.