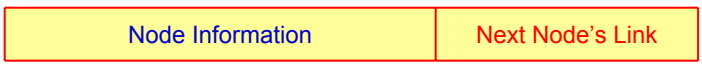
Linked Structures

Linked Lists

The Concept

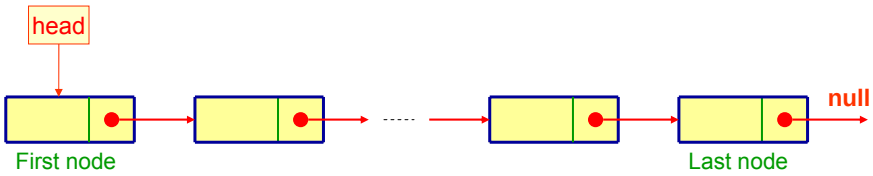
- Each element in a linked data structure is called a node
- A node contains two kind of fields:
  - The information field: Holds a reference to an element
  - Next link field(s): Hold reference(s) to one or more nodes in the structure



One node in a linked structure

Example Linked List Structure

- A linked list structure may look like this:



- The head pointer is an external reference to a node
- It always gives access to the first node in the list
- Access to other nodes is one-way and sequential through the stored links / pointers

Design of Linked Lists

Linked List Design Parameters:

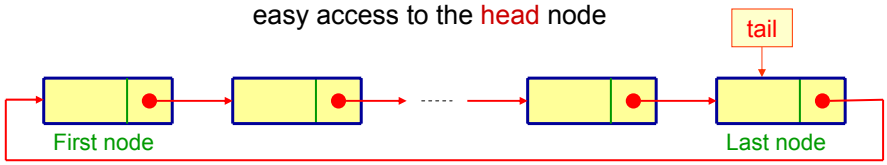
1. How to mark the ends of the list?
   - A head pointer is always used to point to the first node
   - Possibilities for the last node are:
     a) Using a null value in the pointer field of the last node
        - Expensive to locate the last node
     b) Keeping a count of number of nodes in the list
        - Easy to get the size of the list
        - Still expensive to locate the last node

Design of Linked Lists (cont.)

c) Keeping an external pointer which always points to last node
   - Gives direct access to the last node

d) Any combination of the above choices

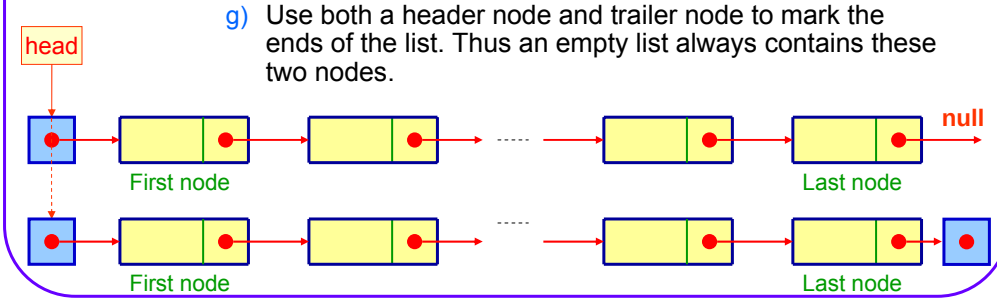
e) Making the last node's pointer point back to the first node
   - The list is called a circularly linked list
   - Keep only one pointer, called tail, which also give easy access to the head node



Design of Linked Lists (cont.)

f) Using a dummy header node, which may or may not contain any information, as the first node in the list. Thus an empty list always contains this header node
   - Simplifies some function implementations
   - No special case at the head

g) Use both a header node and trailer node to mark the ends of the list. Thus an empty list always contains these two nodes.



Design of Linked Lists (cont.)

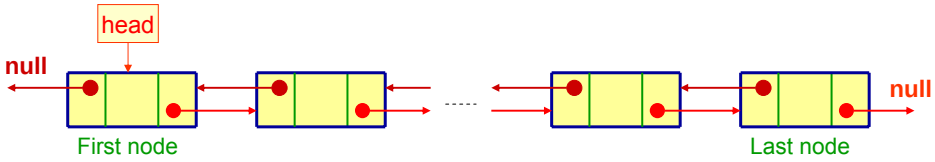
2. What external pointers give access to the list? and to which nodes?
   - At least one node of the list must be accessed through an external pointer (usually the first node)
   - Other possibilities include: A current pointer, a tail pointer, …, and so on

Design of Linked Lists (cont.)

3. How many pointers should be stored in each node that point to related nodes?
   - At least one is necessary to point to the next node. The list is called "Singly Linked List"
     - Predecessor nodes are difficult to access
     - The list can only be traversed in one direction
     - Access at any point other than head node makes those before it inaccessible.

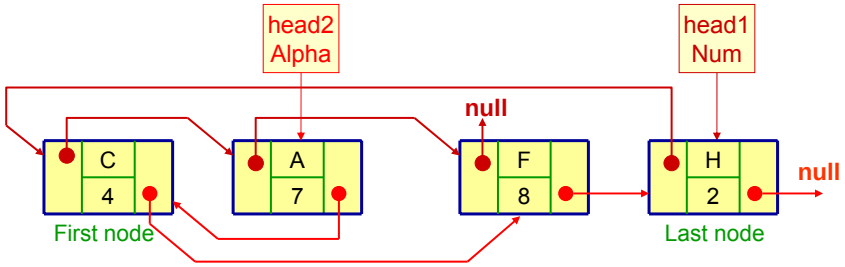
Design of Linked Lists (cont.)

- Other possibilities are:
  a) Store two pointers in each node, one points to its successor and the other points to its predecessor
     - The list is called a "Doubly Linked List" if the ordering in one direction is the logical inverse of the other
     - The list can be traversed in both directions
     - Easy to get the successor and predecessor of a node



Design of Linked Lists (cont.)

b) Store two pointers in each node, each one giving a different ordering of the node
   - Such a list is called "Multi-Linked List" of order two
   - Useful to keep one physical order and have two different logical orders



Design of Linked Lists (cont.)

c) Store more than two pointers (n pointers) in each node to get an extension of (b), called “Multi-Linked List” of order n.

Linked Structures

The Linear List

Abstract Definition

A Linear List is an ordered collection of elements from set S. The order is defined either by position or by other kind of ordering.

In other words, a linear list is either empty or can be written as:

$$(a_1, a_2, a_3, \ldots, a_n)$$

Where a_i are elements of some set S.

Specifications

- Each element of a list is assumed to have at least two fields, as shown:
- Keys are unique identities of the elements.

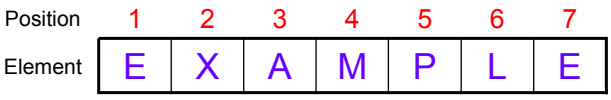
- The list contains n elements.

Structure

- There is a linear relationship between elements.
- Each element in the list has a unique position.
- If the position of the first element is k, then the position of its successor is k+1.

Example:

Element X has position 2; Its successor is A, which has position 3.



Operations

Some of the operations assume there is one element in the list designated as current.

first()
last()
next()
prior()
seek(position)
search (target)
get()
set(entry)
getPosition()
addBefore(entry)
addAfter(entry)
remove()
isElement()
isEmpty() , isFull()
size()
clear()

List Implementation
Using a Linked Structure

- Use a Circularly Doubly Linked List structure with a head pointer
- Each node in the list is an instance of a DNode class that contains the following private fields:
  - element — where the node information is stored
  - next — which is a forward link pointer
  - prev — which is a backward link pointer

- Use two external pointers into the list:
  - head — points to the first node in the list
  - current — points to the current node in the list

List Implementation
Using a Linked Structure (cont.)

- Use two integer variables:
  - position — stores the position of the current node.
  - size — stores the number of nodes currently in the list.
- An empty list is initialized by setting:
  - head = current = null
  - position = size = 0.

The Constructor

- At any time, if the list has position == 0 or current == null, then there is no current item.

List Implementation
The DNode Class Structure

- Used as a generic class with a variable data type, E, as the type of information stored in each node
- The link fields are pointers to a DNode object type
- The class has a constructor to initialize its data fields
- Functions for accessing the class fields are also members of the DNode class

```
public class DNode <E>
  private E element;
  private DNode<E> next;
  private DNode<E> prev;

public:
  DNode(E initElement,
        DNode<E> initNext,
        DNode<E> initPrev);
  void setElement(E newElement);
  void setNext(DNode<E> newNext);
  void setPrev(DNode<E> newPrev);
  E getElement( );
  DNode<E> getNext( );
  DNode<E> getPrev( );
```

List Implementation
The DLinkedList Class Structure

- Used as a generic class with a variable data type, E, as the type of information stored in each node
- The head and current fields are external pointers to the list nodes
- The class has a constructor to initialize its data fields.
- Some functions that implement the List operations are also members of the List class

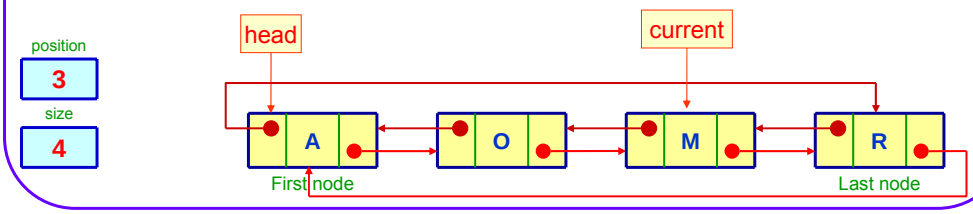
```
public class DLinkedList<E>
  private DNode<E> head;
  private DNode<E> current;
  private int position, size;

  public DLinkedList();
  public void first();   .....
  public void next();   .....
  public void search(E target);
  public void addBefore(E entry);
  public void addAfter(E entry);
  public void remove();
  public void set(E entry);
  public void clear();
  public int size();       ......
```

Mapping Operation: first() ... *O*(1)

- **Postcondition**:
  - First element on the list becomes current element.
  - If the list is empty, there is no current element.
- **Code**:

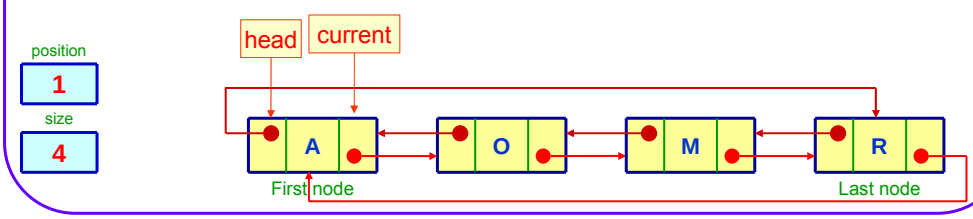
```
if (size > 0)
{
    current = head;
    position = 1;
}
```



Mapping Operation: first() ... *O*(1)

- **Postcondition**:
  - First element on the list becomes current element.
  - If the list is empty, there is no current element.
- **Code**:

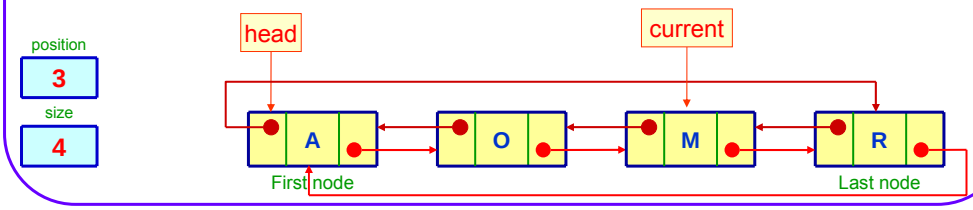
```
if (size > 0)
{
    current = head;
    position = 1;
}
```



Mapping Operation: last() ... *O*(1)

- **Postcondition**:
  - Last element on the list becomes current element.
  - If the list is empty, there is no current element.
- **Code**:

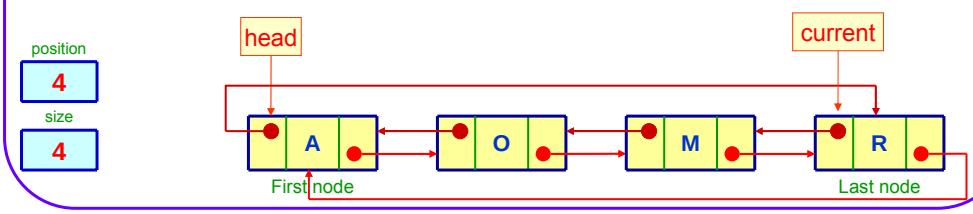
```
if (size > 0)
{
    current = head.getPrev();
    position = size;
}
```



Mapping Operation: last() ... *O*(1)

- **Postcondition**:
  - Last element on the list becomes current element.
  - If the list is empty, there is no current element.
- **Code**:

```
if (size > 0)
{
    current = head.getPrev();
    position = size;
}
```

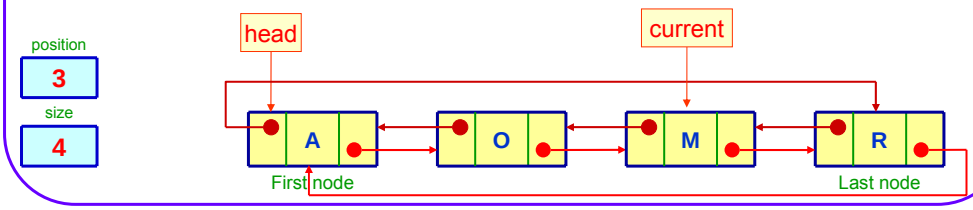


Mapping Operation: next() ... *O*(1)

- **Precondition:**
  - isElement() returns true.
- **Postcondition**:
  - If current is at the last element, then there is no current element.
  - Otherwise, the new current is the element immediately after the current one.

- **Code**:

```
assert isElement();
if (position < size)
    {
      current = current.getNext();
      ++position;
    }
    else
    {
      current = null;
      position = 0;
    }
```

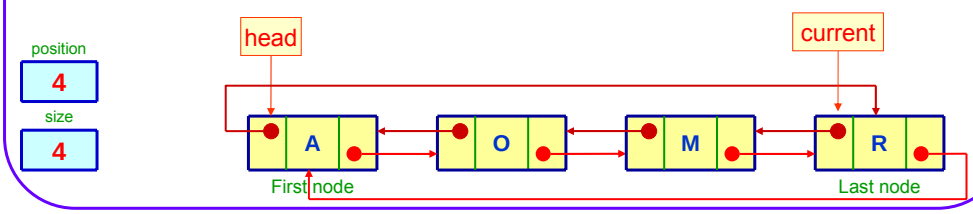


Mapping Operation: next() ... *O*(1)

- **Precondition:**
  - isElement() returns true.
- **Postcondition**:
  - If current is at the last element, then there is no current element.
  - Otherwise, the new current is the element immediately after the current one.

- **Code**:

```
assert isElement();
if (position < size)
    {
      current = current.getNext();
      ++position;
    }
    else
    {
      current = null;
      position = 0;
    }
```

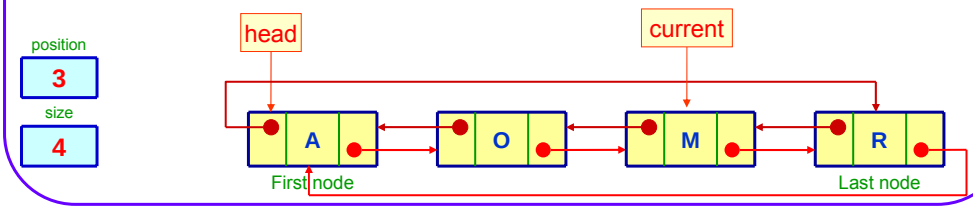


Mapping Operation: prior() ... *O*(1)

- **Precondition:**
  - isElement() returns true.
- **Postcondition**:
  - If current is at the first element, then there is no current element.
  - Otherwise, the new current is the element immediately before the current one.

- **Code**:

```
assert isElement();
if (position > 1)
    {
      current = current.getPrev();
      --position;
    }
    else
    {
      current = null;
      position = 0;
    }
```

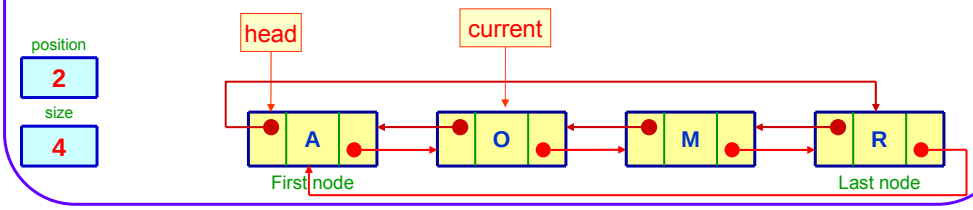


Mapping Operation: prior() ... *O*(1)

- **Precondition:**
  - isElement() returns true.
- **Postcondition**:
  - If current is at the first element, then there is no current element.
  - Otherwise, the new current is the element immediately before the current one.

- **Code**:

```
assert isElement();
if (position > 1)
    {
      current = current.getPrev();
      --position;
    }
    else
    {
      current = null;
      position = 0;
    }
```

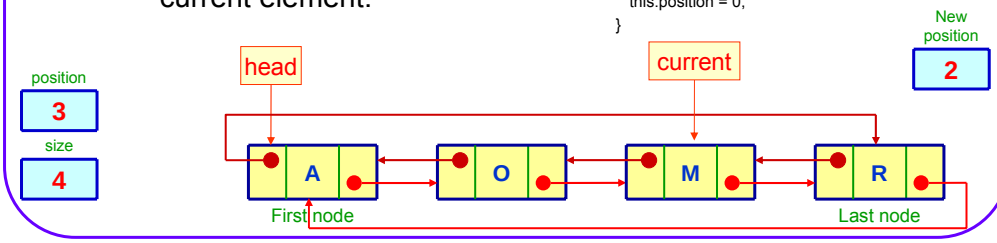



Mapping Operation: seek(position) ... *O*(n)

- **Postcondition**:
  - If the new position is within the list, the element at that position becomes the current element.
  - Otherwise, there is no current element.

- **Code**:

```
if ((position > 0) && (position <= size))  {
        current = head;
        for (k = 1; k < position; k++)
              current = current.getNext();
        this.position = position;
    }
    else  {
        current = null;
        this.position = 0;
    }
```

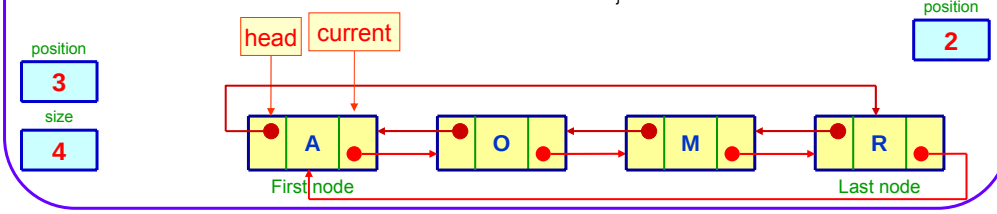


Mapping Operation: seek(position) ... *O*(n)

- **Postcondition**:
  - If the new position is within the list, the element at that position becomes the current element.
  - Otherwise, there is no current element.

- **Code**:

```
if ((position > 0) && (position <= size))  {
        current = head;
        for (k = 1; k < position; k++)
              current = current.getNext();
        this.position = position;
    }
    else  {
        current = null;
        this.position = 0;
    }
```

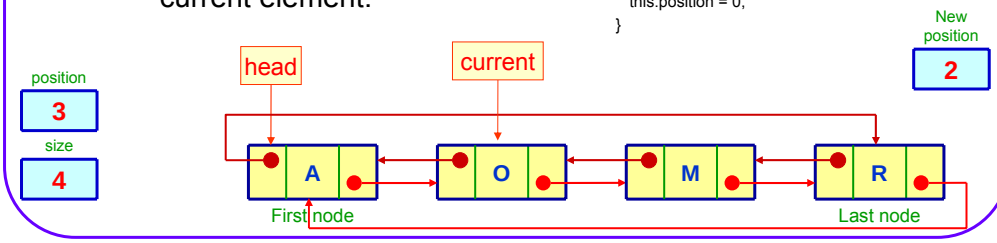



Mapping Operation: seek(position) ... *O*(n)

- **Postcondition**:
  - If the new position is within the list, the element at that position becomes the current element.
  - Otherwise, there is no current element.

- **Code**:

```
if ((position > 0) && (position <= size))  {
        current = head;
        for (k = 1; k < position; k++)
              current = current.getNext();
        this.position = position;
    }
    else  {
        current = null;
        this.position = 0;
    }
```

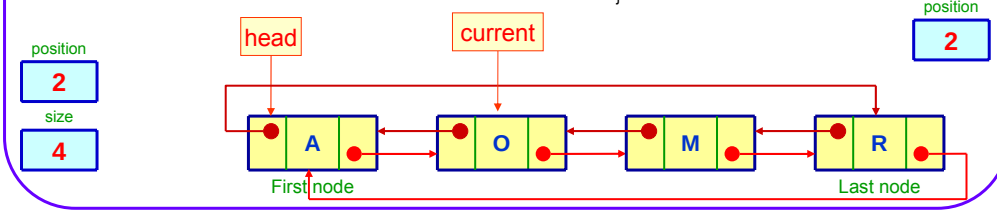


Mapping Operation: seek(position) ... *O*(n)

- **Postcondition**:
  - If the new position is within the list, the element at that position becomes the current element.
  - Otherwise, there is no current element.

- **Code**:

```
if ((position > 0) && (position <= size))  {
        current = head;
        for (k = 1; k < position; k++)
              current = current.getNext();
        this.position = position;
    }
    else  {
        current = null;
        this.position = 0;
    }
```

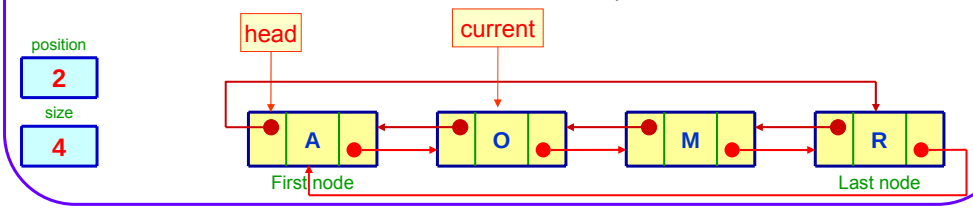


Mapping Operation: seek(position) ... *O*(n)

- **Postcondition**:
  - If the new position is within the list, the element at that position becomes the current element.
  - Otherwise, there is no current element.

- **Code**:

```
if ((position > 0) && (position <= size))   {
        current = head;
        for (k = 1; k < position; k++)
              current = current.getNext();
        this.position = position;
    }
    else   {
        current = null;
        this.position = 0;
    }
```



Mapping Operation: get() ... *O*(1)

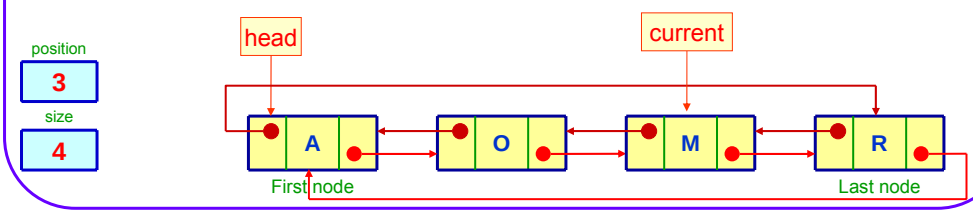
- **Precondition**:
  - isElement() returns true.
- **Postcondition**:
  - The element returned is the current element in the list.

- **Code**:

```
assert isElement();
return current.getElement();
```

- **Example**:

  get(); ➔ returns M



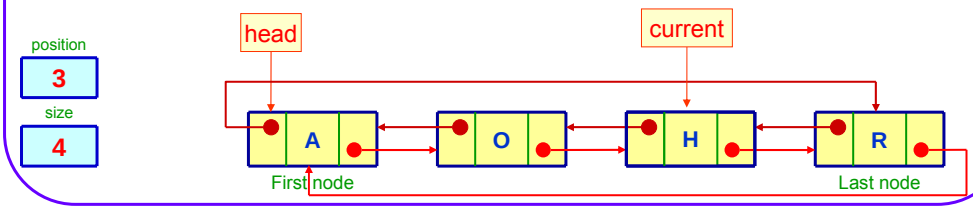
Mapping Operation: set(entry) ... *O*(1)

- **Precondition**:
  - isElement() returns true.
- **Postcondition**:
  - The value of the current element has changed to entry.
- **Code**:

```
assert isElement();
current.setElement(entry);
```

- **Example**:

```
set('H');
```



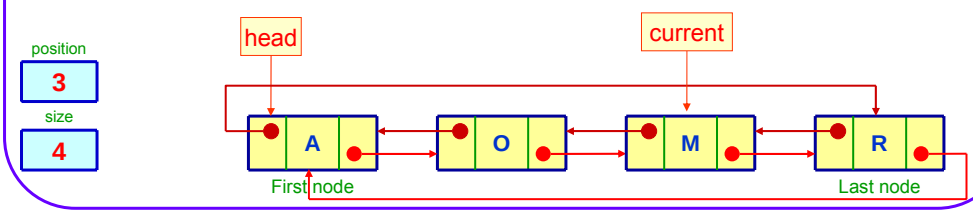
Mapping Operation: getPosition() ... *O*(1)

- **Postcondition**:
  - The returned value is the position of the current element in the list.
- **Code**:

```
return position;
```

- **Example**:

```
getPosition(); ➔ returns 3.
```



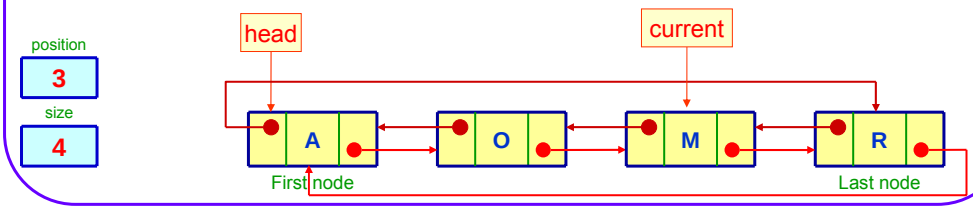
Mapping Operation: isElement() ... *O*(1)

- **Postcondition**:
  - A true return value indicates that there is a valid "current" element. A false return value indicates that there is no valid current element.
- **Code**:

```
return (position > 0);
```

- **Example**:

  isElement(); ➔ returns true.



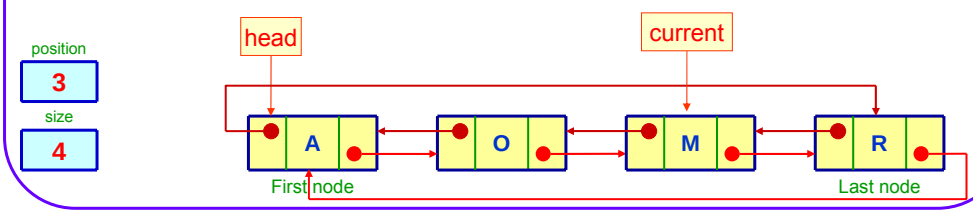
Mapping Operation: isEmpty() ... *O*(1)

- **Postcondition**:
  - The return value is true if the list has no elements, otherwise, it is false.
- **Code**:

```
return (size == 0);
```

- **Example**:

  isEmpty(); ➔ returns false.

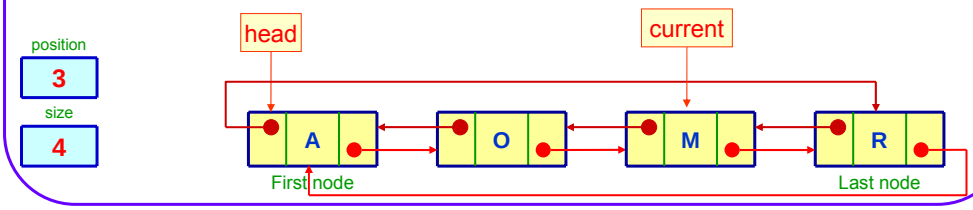


Mapping Operation: isFull() ... *O*(1)

- **Postcondition**:
  - The return value is true if the list has a number of elements equal to its capacity, otherwise it is false.
- **Code**:

  return false;
- **Example**:

  isFull(); ➔ returns false.

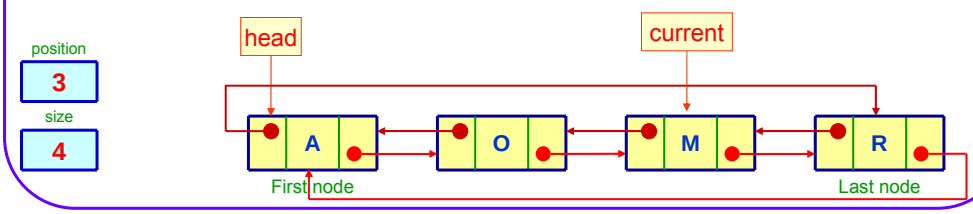


Mapping Operation: size() ... *O*(1)

- **Postcondition**:
  - The return value is the number of elements in the list.
- **Code**:

  return size;
- **Example**:

  size(); ➔ returns 4.

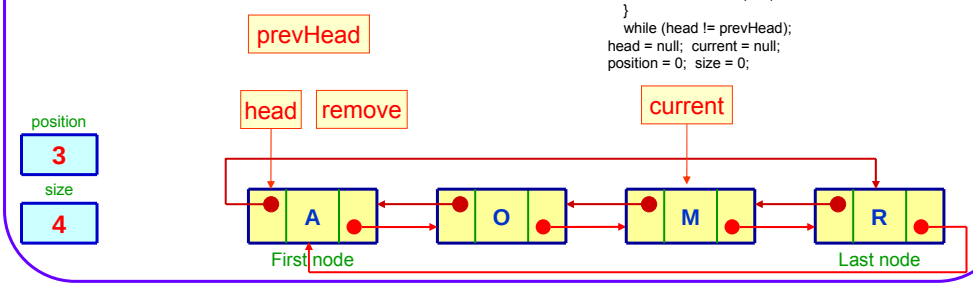


Mapping Operation: clear() ... *O*(n)

- **Postcondition**:
  - The list is now empty.

- **Code**:

```
DNode<E> prevHead, remove;
prevHead = head;
if (head != null)
   do
   {
      remove = head;
      head = head.getNext();
      remove.setNext(null);
      remove.setPrev(null);
   }
   while (head != prevHead);
head = null;  current = null;
position = 0;  size = 0;
```

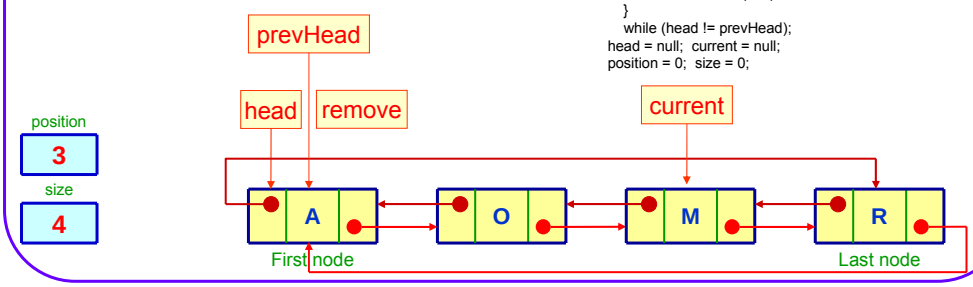


Mapping Operation: clear() ... *O*(n)

- **Postcondition**:
  - The list is now empty.

- **Code**:

```
DNode<E> prevHead, remove;
prevHead = head;
if (head != null)
   do
   {
      remove = head;
      head = head.getNext();
      remove.setNext(null);
      remove.setPrev(null);
   }
   while (head != prevHead);
head = null;  current = null;
position = 0;  size = 0;
```

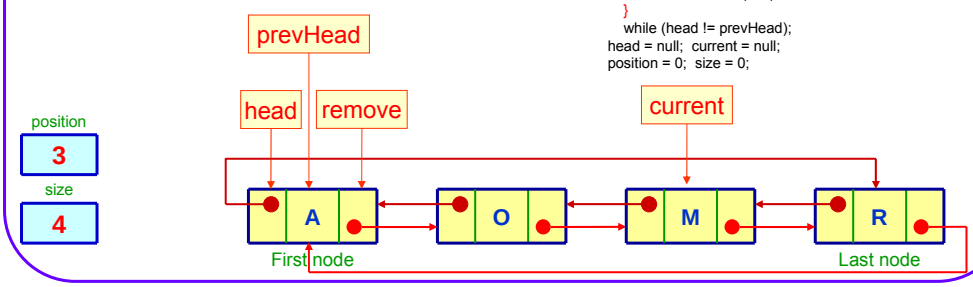


Mapping Operation: clear() ... *O*(n)

- **Postcondition**:
  - The list is now empty.

- **Code**:

```
DNode<E> prevHead, remove;
prevHead = head;
if (head != null)
   do
   {
      remove = head;
      head = head.getNext();
      remove.setNext(null);
      remove.setPrev(null);
   }
   while (head != prevHead);
head = null;  current = null;
position = 0;  size = 0;
```

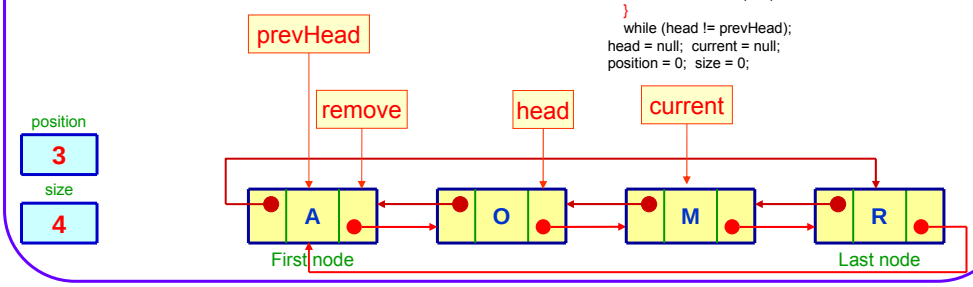


Mapping Operation: clear() ... *O*(n)

- **Postcondition**:
  - The list is now empty.

- **Code**:

```
DNode<E> prevHead, remove;
prevHead = head;
if (head != null)
   do
   {
      remove = head;
      head = head.getNext();
      remove.setNext(null);
      remove.setPrev(null);
   }
   while (head != prevHead);
head = null;  current = null;
position = 0;  size = 0;
```

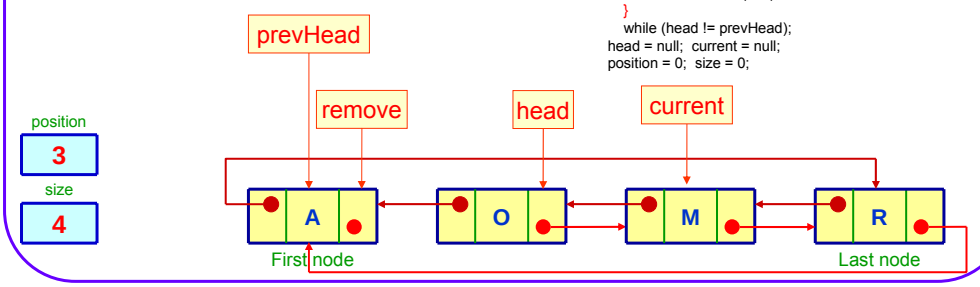


Mapping Operation: clear() ... *O*(n)

- **Postcondition**:
  - The list is now empty.

- **Code**:

```
DNode<E> prevHead, remove;
prevHead = head;
if (head != null)
  do
  {
     remove = head;
     head = head.getNext();
     remove.setNext(null);
     remove.setPrev(null);
  }
  while (head != prevHead);
head = null;  current = null;
position = 0;  size = 0;
```

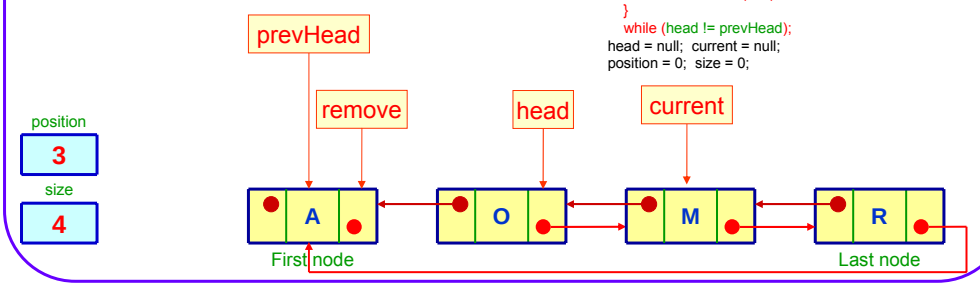


Mapping Operation: clear() ... *O*(n)

- **Postcondition**:
  - The list is now empty.

- **Code**:

```
DNode<E> prevHead, remove;
prevHead = head;
if (head != null)
  do
  {
     remove = head;
     head = head.getNext();
     remove.setNext(null);
     remove.setPrev(null);
  }
  while (head != prevHead);
head = null;  current = null;
position = 0;  size = 0;
```

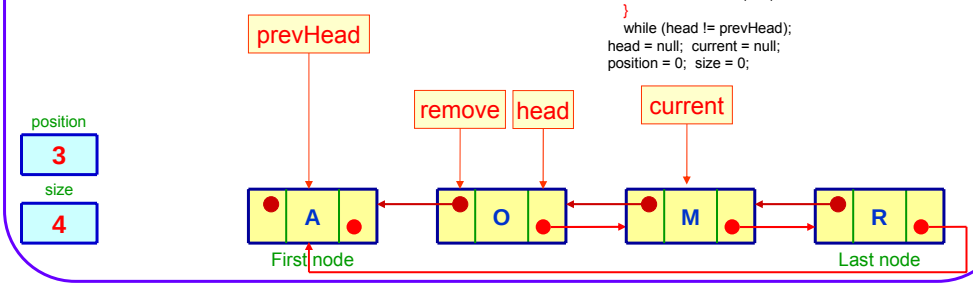



Mapping Operation: clear() ... *O*(n)

- **Postcondition**:
  - The list is now empty.

- **Code**:

```
DNode<E> prevHead, remove;
prevHead = head;
if (head != null)
  do
  {
    remove = head;
    head = head.getNext();
    remove.setNext(null);
    remove.setPrev(null);
  }
  while (head != prevHead);
head = null;  current = null;
position = 0;  size = 0;
```

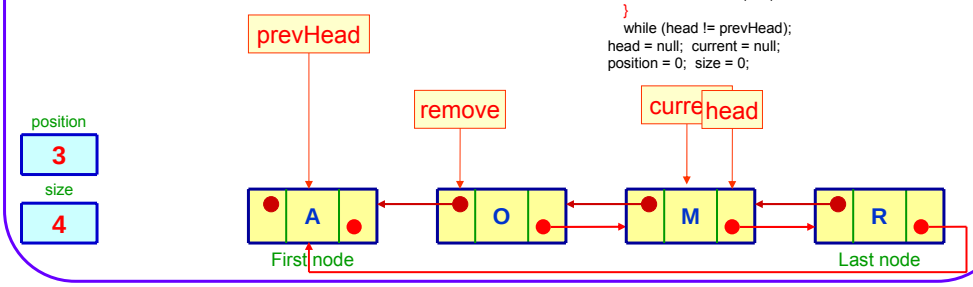


Mapping Operation: clear() ... *O*(n)

- **Postcondition**:
  - The list is now empty.

- **Code**:

```
DNode<E> prevHead, remove;
prevHead = head;
if (head != null)
  do
  {
    remove = head;
    head = head.getNext();
    remove.setNext(null);
    remove.setPrev(null);
  }
  while (head != prevHead);
head = null;  current = null;
position = 0;  size = 0;
```

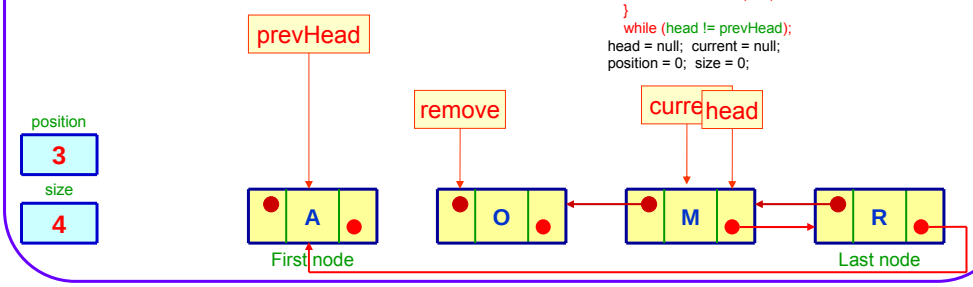


Mapping Operation: clear() ... *O*(n)

- **Postcondition**:
  - The list is now empty.

- **Code**:

```
DNode<E> prevHead, remove;
prevHead = head;
if (head != null)
   do
   {
      remove = head;
      head = head.getNext();
      remove.setNext(null);
      remove.setPrev(null);
   }
   while (head != prevHead);
head = null;  current = null;
position = 0;  size = 0;
```

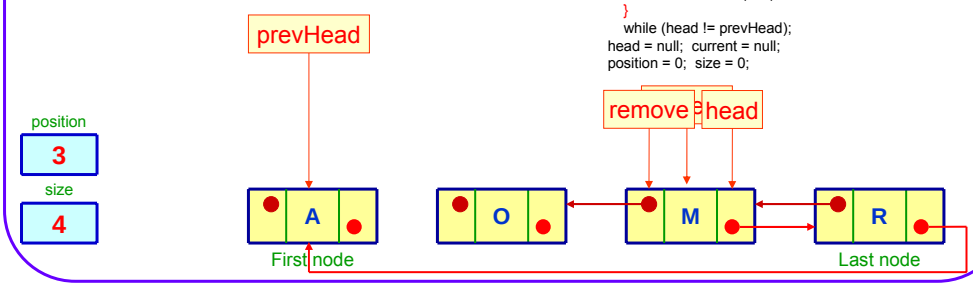


Mapping Operation: clear() ... *O*(n)

- **Postcondition**:
  - The list is now empty.

- **Code**:

```
DNode<E> prevHead, remove;
prevHead = head;
if (head != null)
   do
   {
      remove = head;
      head = head.getNext();
      remove.setNext(null);
      remove.setPrev(null);
   }
   while (head != prevHead);
head = null;  current = null;
position = 0;  size = 0;
```

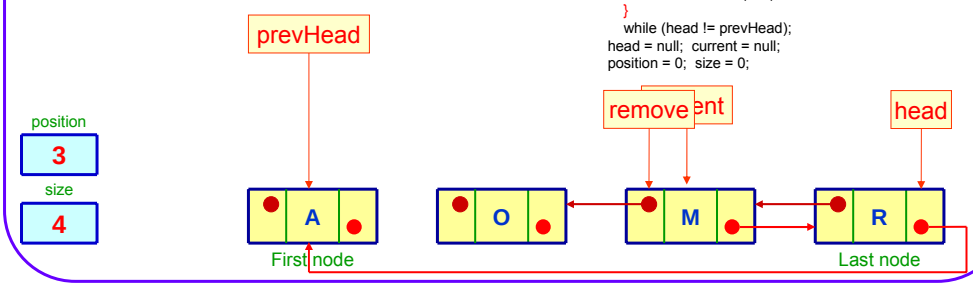


Mapping Operation: clear() ... *O*(n)

- **Postcondition**:
  - The list is now empty.

- **Code**:

```
DNode<E> prevHead, remove;
prevHead = head;
if (head != null)
  do
  {
     remove = head;
     head = head.getNext();
     remove.setNext(null);
     remove.setPrev(null);
  }
  while (head != prevHead);
head = null;  current = null;
position = 0;  size = 0;
```

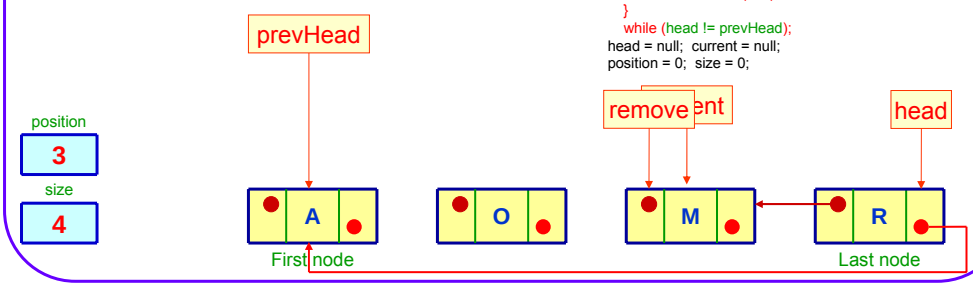


Mapping Operation: clear() ... *O*(n)

- **Postcondition**:
  - The list is now empty.

- **Code**:

```
DNode<E> prevHead, remove;
prevHead = head;
if (head != null)
  do
  {
     remove = head;
     head = head.getNext();
     remove.setNext(null);
     remove.setPrev(null);
  }
  while (head != prevHead);
head = null;  current = null;
position = 0;  size = 0;
```

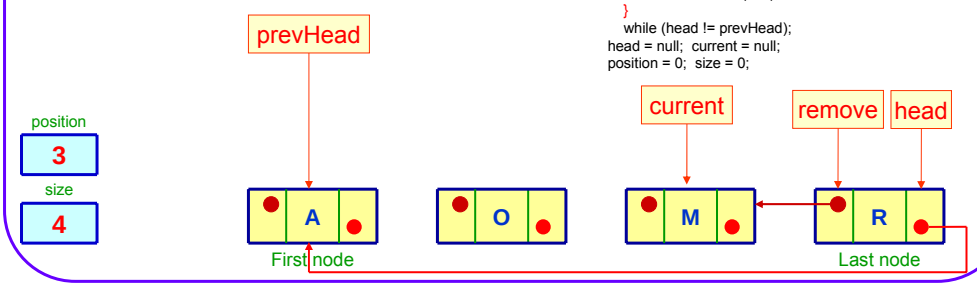


Mapping Operation: clear() ... *O*(n)

- **Postcondition**:
  - The list is now empty.

- **Code**:

```
DNode<E> prevHead, remove;
prevHead = head;
if (head != null)
  do
  {
    remove = head;
    head = head.getNext();
    remove.setNext(null);
    remove.setPrev(null);
  }
  while (head != prevHead);
head = null;  current = null;
position = 0;  size = 0;
```

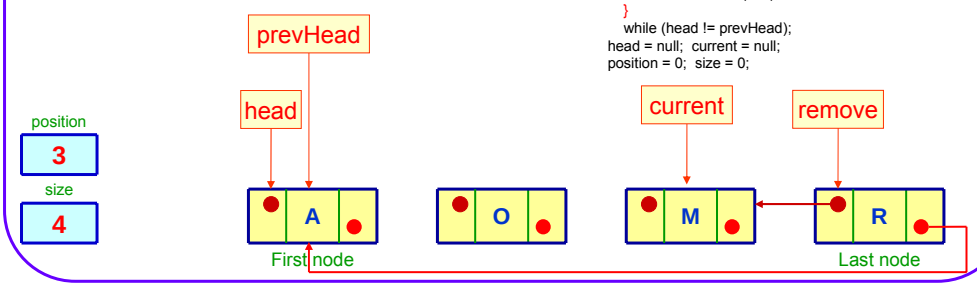


Mapping Operation: clear() ... *O*(n)

- **Postcondition**:
  - The list is now empty.

- **Code**:

```
DNode<E> prevHead, remove;
prevHead = head;
if (head != null)
  do
  {
    remove = head;
    head = head.getNext();
    remove.setNext(null);
    remove.setPrev(null);
  }
  while (head != prevHead);
head = null;  current = null;
position = 0;  size = 0;
```

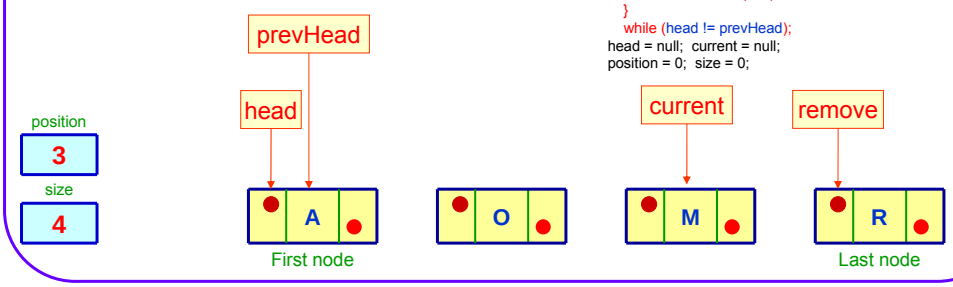


Mapping Operation: clear() ... *O*(n)

- **Postcondition**:
  - The list is now empty.

- **Code**:

```
DNode<E> prevHead, remove;
prevHead = head;
if (head != null)
   do
   {
      remove = head;
      head = head.getNext();
      remove.setNext(null);
      remove.setPrev(null);
   }
   while (head != prevHead);
head = null;  current = null;
position = 0;  size = 0;
```



Mapping Operation: clear() ... *O*(n)

- **Postcondition**:
  - The list is now empty.

- **Code**:

```
DNode<E> prevHead, remove;
prevHead = head;
if (head != null)
   do
   {
      remove = head;
      head = head.getNext();
      remove.setNext(null);
      remove.setPrev(null);
   }
   while (head != prevHead);
head = null;  current = null;
position = 0;  size = 0;
```

Mapping Operation: search(target) ... *O*(n)

- **Postcondition**:
  - The list has been searched for the target. If the target was present, then the found target is now the current element. Otherwise, there is no current element .

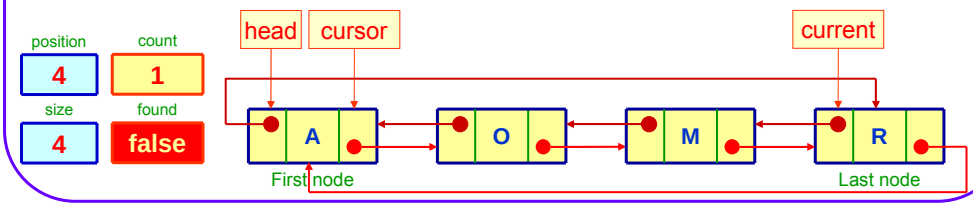
- **Code**:

```
int count = 1;
boolean found = false;
DNode<E> cursor = head;

while ((cursor.getNext() != head) && !found) {
   if (cursor.getElement() == target)
      found = true;
   else {
      cursor = cursor.getNext();
      count++;
   }
}
if (found) {
   current = cursor;
   position = count;
}
else {
   current = null;
   position = 0;
}
```

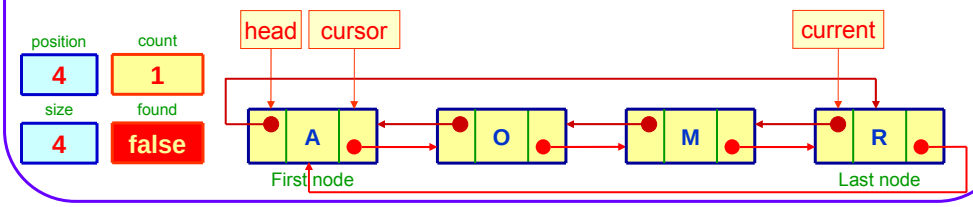
Mapping Operation: search(target) Example: search('M');

```
int count = 1;
boolean found = false;
DNode<E> cursor = head;
```



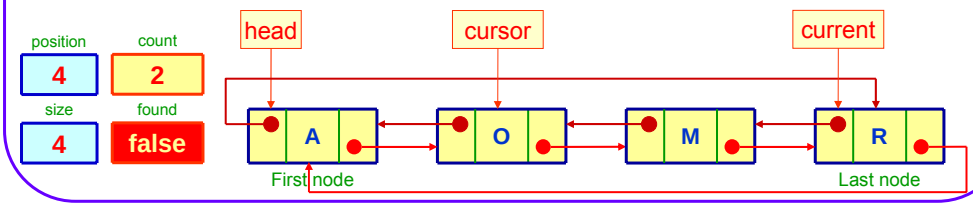
Mapping Operation: search(target)
Example: search('M');

```
while ((cursor.getNext() != head) && !found)  {
   if (cursor.getElement() == target)
      found = true;
   else  {
      cursor = cursor.getNext();
      count++;
   }
}
```



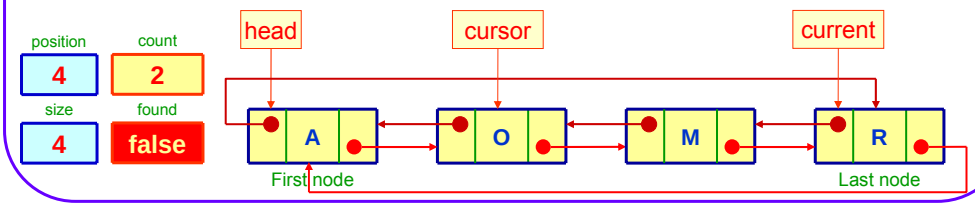
Mapping Operation: search(target)
Example: search('M');

```
while ((cursor.getNext() != head) && !found)  {
   if (cursor.getElement() == target)
      found = true;
   else  {
      cursor = cursor.getNext();
      count++;
   }
}
```



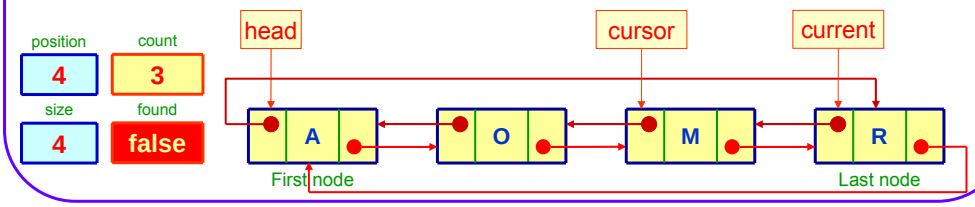
Mapping Operation: search(target)
Example: search('M');

```
while ((cursor.getNext() != head) && !found)  {
   if (cursor.getElement() == target)
      found = true;
   else  {
      cursor = cursor.getNext();
      count++;
   }
}
```



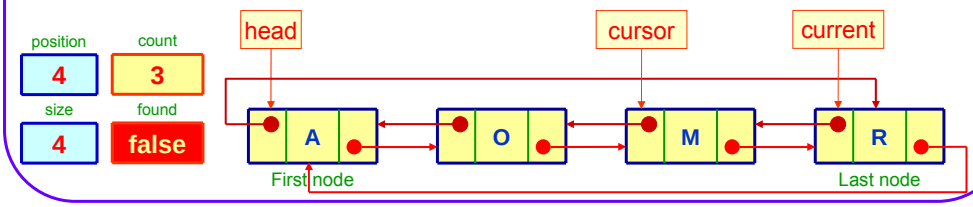
Mapping Operation: search(target)
Example: search('M');

```
while ((cursor.getNext() != head) && !found)  {
   if (cursor.getElement() == target)
      found = true;
   else  {
      cursor = cursor.getNext();
      count++;
   }
}
```



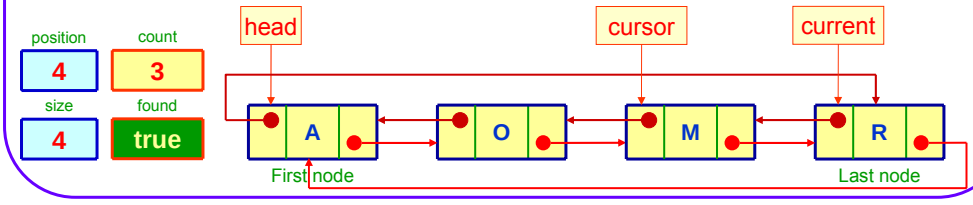
Mapping Operation: search(target)
Example: search('M');

```
while ((cursor.getNext() != head) && !found)  {
   if (cursor.getElement() == target)
      found = true;
   else  {
      cursor = cursor.getNext();
      count++;
   }
}
```



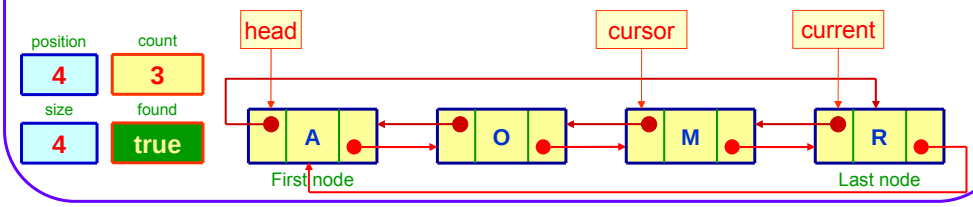
Mapping Operation: search(target)
Example: search('M');

```
while ((cursor.getNext() != head) && !found)  {
   if (cursor.getElement() == target)
      found = true;
   else  {
      cursor = cursor.getNext();
      count++;
   }
}
```



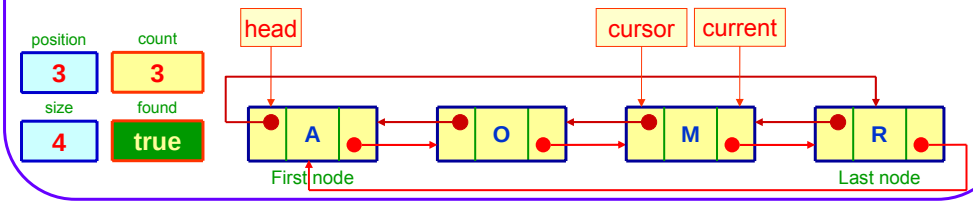
Mapping Operation: search(target)
Example: search('M');

```
while ((cursor.getNext() != head) && !found)  {
   if (cursor.getElement() == target)
      found = true;
   else  {
      cursor = cursor.getNext();
      count++;
   }
}
```



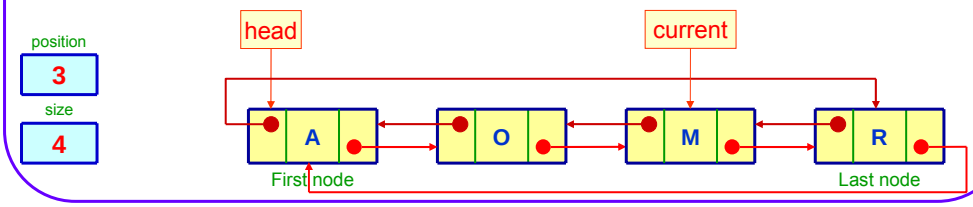
Mapping Operation: search(target)
Example: search('M');

```
if (found)  {
   current = cursor;
   position = count;
}
else  {
   current = null;
   position = 0;
}
```



Mapping Operation: search(target)
Example: search('M');

```
if (found)  {
    current = cursor;
    position = count;
}
else  {
    current = null;
    position = 0;
}
```



Mapping Operation: addAfter(entry) ... *O*(1)

- **Precondition**:
  - size() < CAPACITY.
- **Postcondition**:
  - A copy of entry has been inserted in the list after the current element. If there was no current element, then the new entry has been attached to the end of the list. In either case, the newly inserted element becomes current.

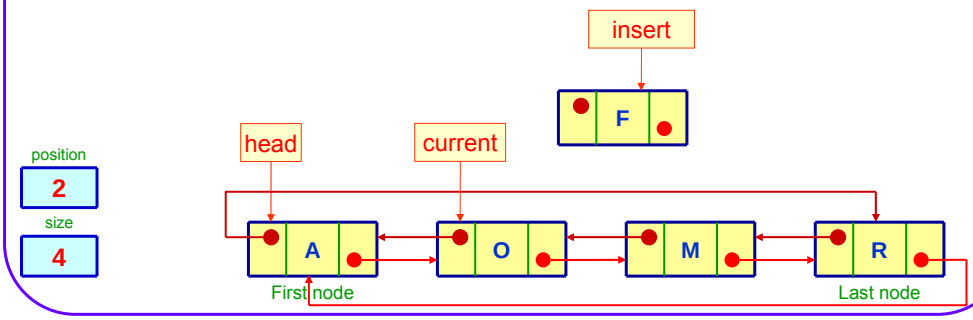
- **Code**:

```
DNode<E> insert;
insert = new DNode<E>(entry, null, null);
if (head == null)  {
    insert.setNext(insert);
    insert.setPrev(insert);
    head = insert;
    position = 1;
}
else  {
    if (position == 0) last();
    insert.setNext(current.getNext());
    insert.setPrev(current);
    current.getNext().setPrev(insert);
    current.setNext(insert);
    ++position;
}
current = insert;
++size;
```

Mapping Operation: addAfter(entry)
Example: addAfter('F');

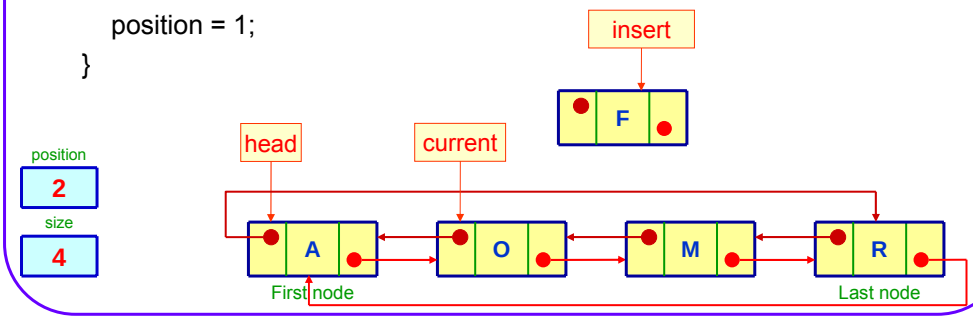
DNode<E> insert;

insert = new DNode<E>('F', null, null);



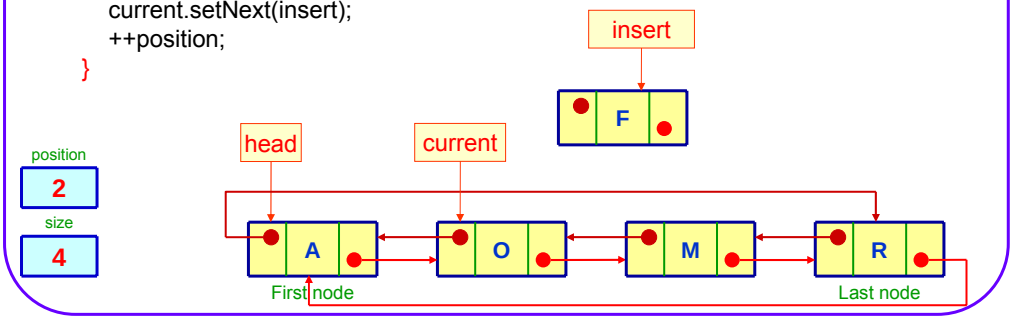
Mapping Operation: addAfter(entry)
Example: addAfter('F');

```
if (head == null)  {
    insert.setNext(insert);
    insert.setPrev(insert);
    head = insert;
    position = 1;
}
```



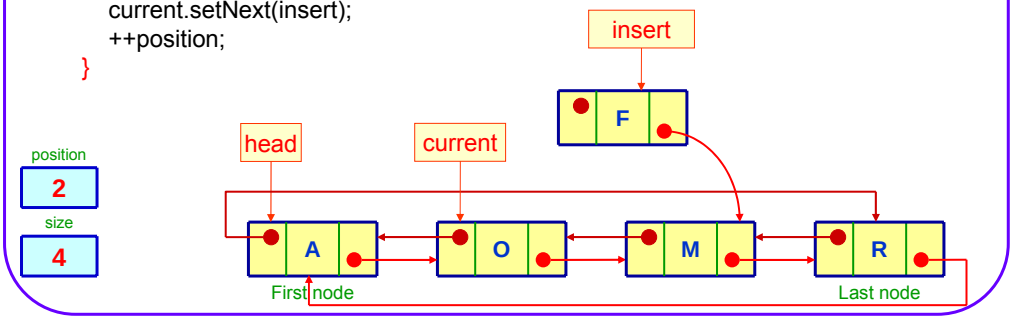
Mapping Operation: addAfter(entry)
Example: addAfter('F');

```
else {
   if (position == 0) last();
   insert.setNext(current.getNext());
   insert.setPrev(current);
   current.getNext().setPrev(insert);
   current.setNext(insert);
   ++position;
}
```



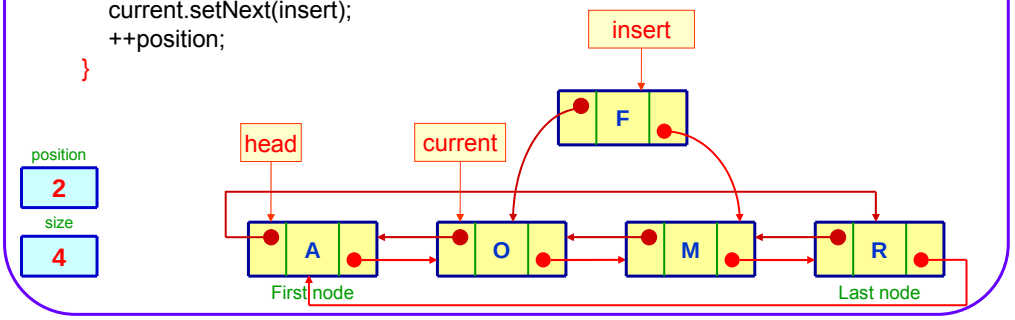
Mapping Operation: addAfter(entry)
Example: addAfter('F');

```
else {
   if (position == 0) last();
   insert.setNext(current.getNext());
   insert.setPrev(current);
   current.getNext().setPrev(insert);
   current.setNext(insert);
   ++position;
}
```



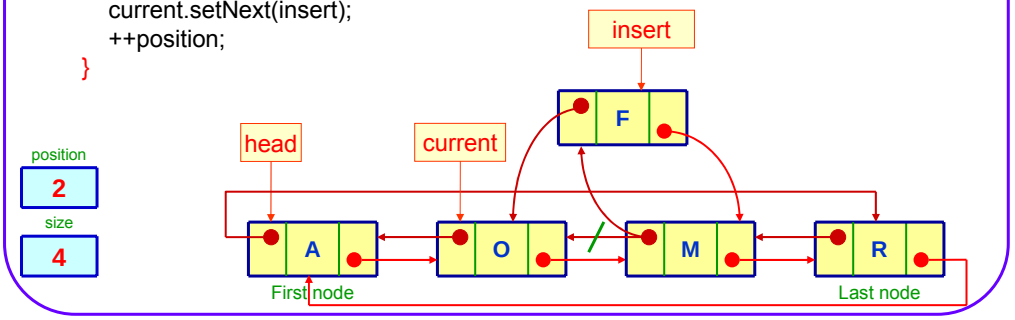
Mapping Operation: addAfter(entry)
Example: addAfter('F');

```
else {
    if (position == 0) last();
    insert.setNext(current.getNext());
    insert.setPrev(current);
    current.getNext().setPrev(insert);
    current.setNext(insert);
    ++position;
}
```



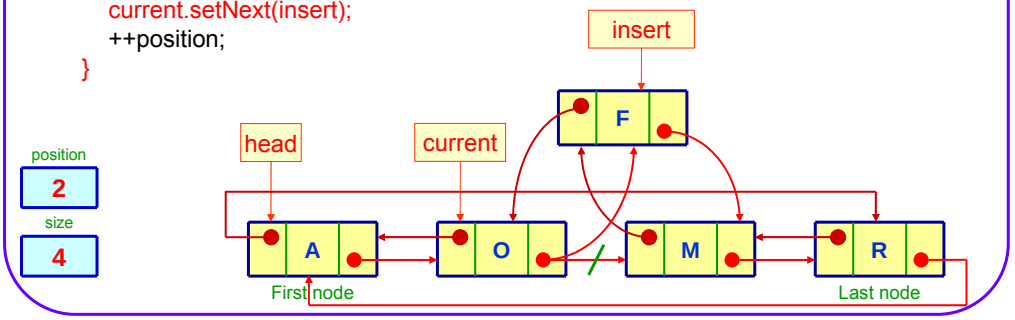
Mapping Operation: addAfter(entry)
Example: addAfter('F');

```
else {
    if (position == 0) last();
    insert.setNext(current.getNext());
    insert.setPrev(current);
    current.getNext().setPrev(insert);
    current.setNext(insert);
    ++position;
}
```



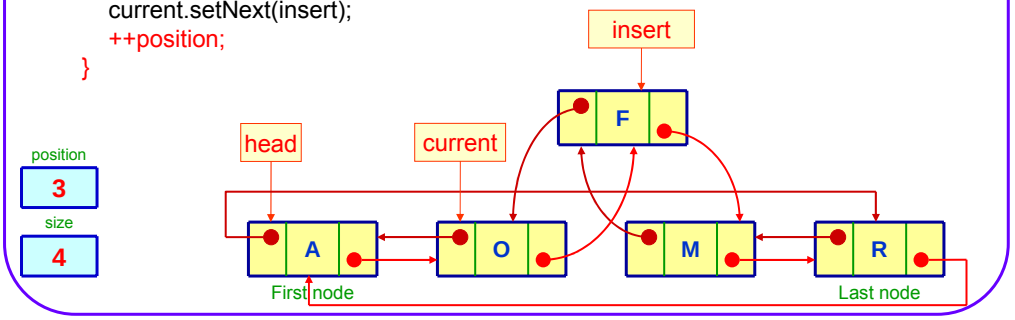
Mapping Operation: addAfter(entry)
Example: addAfter('F');

```
else {
    if (position == 0) last();
    insert.setNext(current.getNext());
    insert.setPrev(current);
    current.getNext().setPrev(insert);
    current.setNext(insert);
    ++position;
}
```



Mapping Operation: addAfter(entry)
Example: addAfter('F');

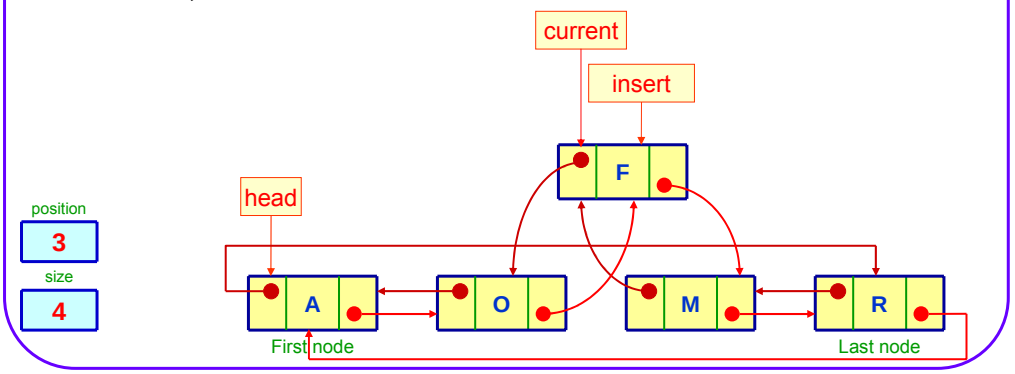
```
else {
    if (position == 0) last();
    insert.setNext(current.getNext());
    insert.setPrev(current);
    current.getNext().setPrev(insert);
    current.setNext(insert);
    ++position;
}
```



Mapping Operation: addAfter(entry)
Example: addAfter('F');

current = insert;

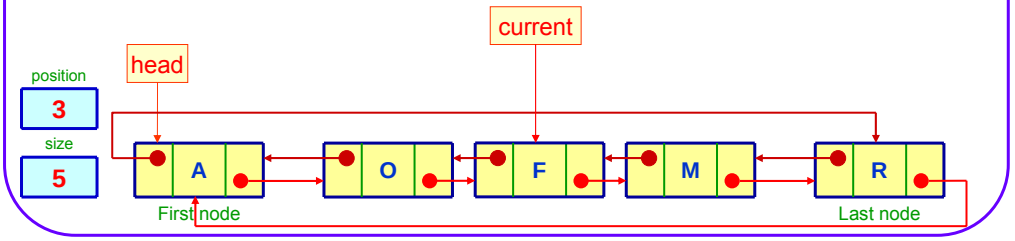
++size;



Mapping Operation: addAfter(entry)
Example: addAfter('F');

current = insert;

++size;



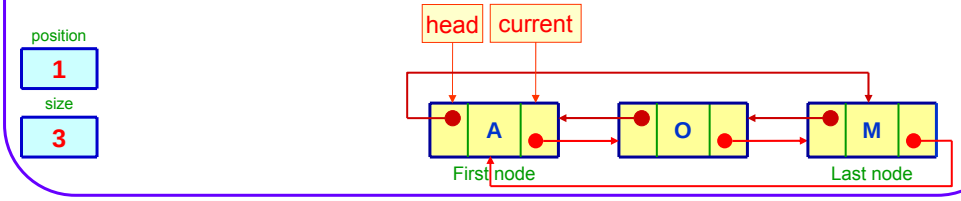
Mapping Operation: addBefore(entry) ... *O*(1)

- **Precondition**:
  - size() < CAPACITY.
- **Postcondition**:
  - A copy of entry has been inserted in the list before the current element. If there was no current element, then the new entry has been inserted at the front of the list. In either case, the newly inserted element becomes current.
- **Code**:

```
if ((head != null) &&
    (position != 0))  {
   current = current.getPrev();
   --position;
}
addAfter(entry);
if (current.getNext() == head)  {
   head = current;
   position = 1;
}
```

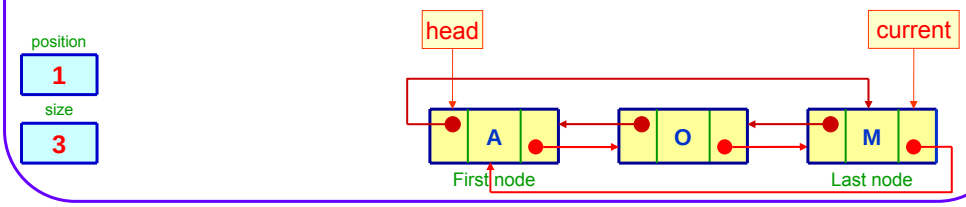
Mapping Operation: addBefore(entry) Example: addBefore('F');

```
if ((head != null) && (position != 0))  {
    current = current.getPrev();
    --position;
}
```



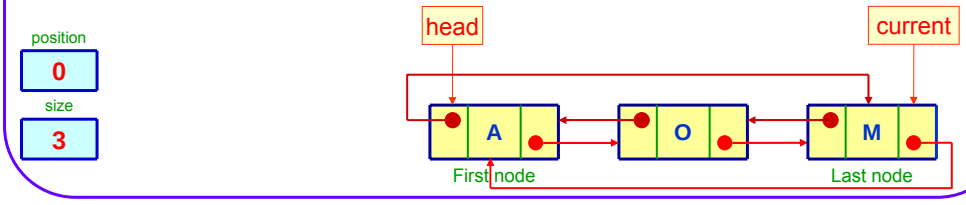
Mapping Operation: addBefore(entry)
Example: addBefore('F');

```
if ((head != null) && (position != 0)) {
    current = current.getPrev();
    --position;
}
```



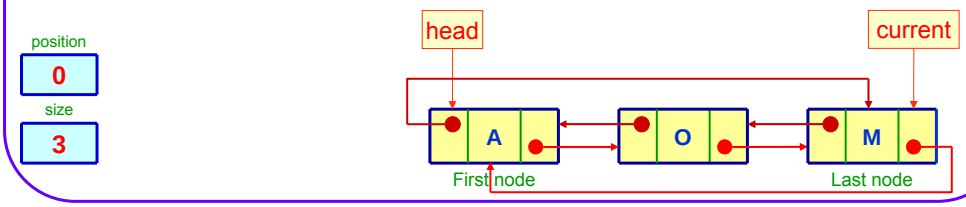
Mapping Operation: addBefore(entry)
Example: addBefore('F');

```
if ((head != null) && (position != 0)) {
    current = current.getPrev();
    --position;
}
```



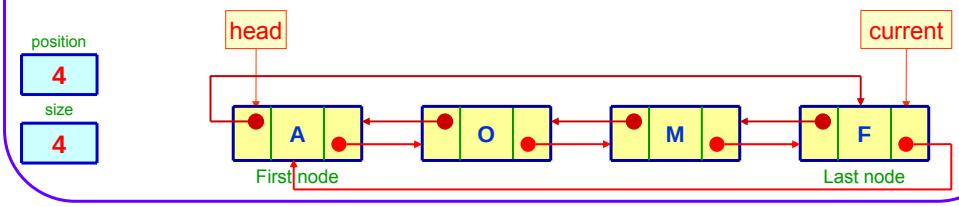
Mapping Operation: addBefore(entry)
Example: addBefore('F');

```
addAfter('F');
if (current.getNext() == head)  {
    head = current;
    position = 1;
}
```



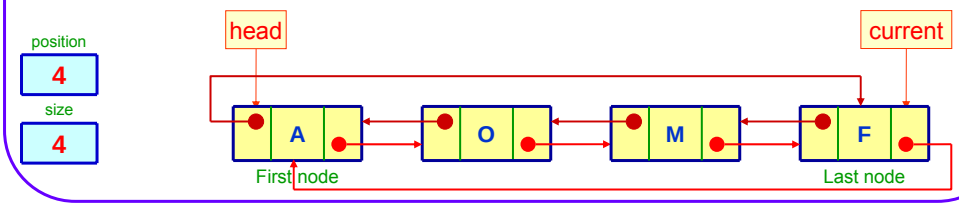
Mapping Operation: addBefore(entry)
Example: addBefore('F');

```
addAfter('F');
if (current.getNext() == head)  {
    head = current;
    position = 1;
}
```



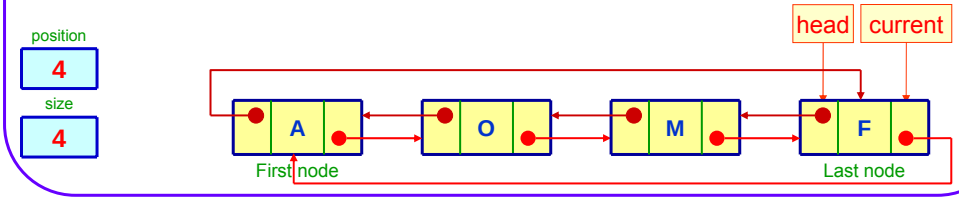
Mapping Operation: addBefore(entry)
Example: addBefore('F');

```
addAfter('F');
if (current.getNext() == head)  {
    head = current;
    position = 1;
}
```



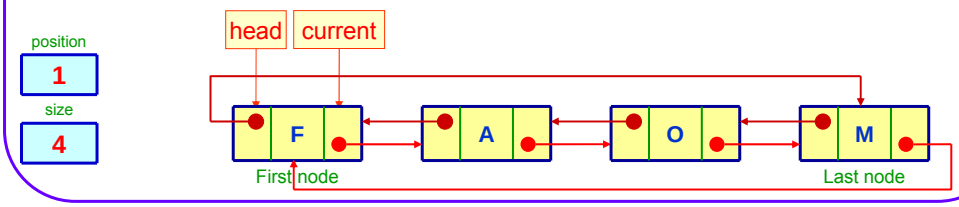
Mapping Operation: addBefore(entry)
Example: addBefore('F');

```
addAfter('F');
if (current.getNext() == head)  {
    head = current;
    position = 1;
}
```



Mapping Operation: addBefore(entry)
Example: addBefore('F');

```
addAfter('F');
if (current.getNext() == head)  {
    head = current;
    position = 1;
}
```



Mapping Operation: remove() ... *O*(1)

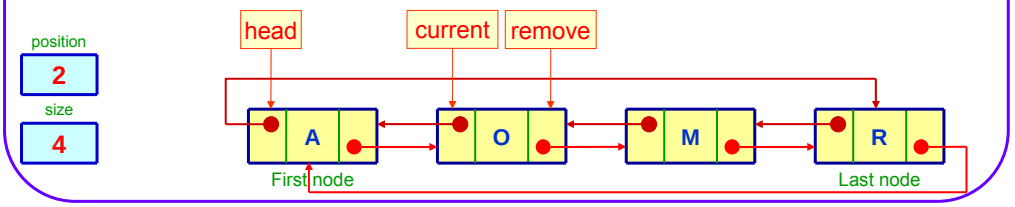
- **Precondition**:
  - isElement() returns true.
- **Postcondition**:
  - The current element has been removed from the list, and the one after it (if there is one) becomes current.
- **Example**:
  Remove();

- **Code**:

```
DNode<E> remove;
assert isElement();
remove = current;
current.getPrev().setNext(current.getNext());
current.getNext().setPrev(current.getPrev());
current = current.getNext();
if (remove.getNext() == head)
    position = 1;
if (size == 1)  {
    head = null;
    current =null;
}
else  if (head == remove)
    head = current;
size--;
remove.setNext(null);
remove.setPrev(null);
```

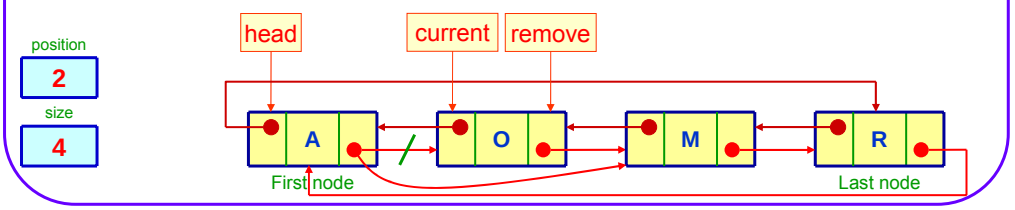
Mapping Operation: remove()
Example: remove()

```
DNode<E> remove;
assert isElement();
remove = current;
```



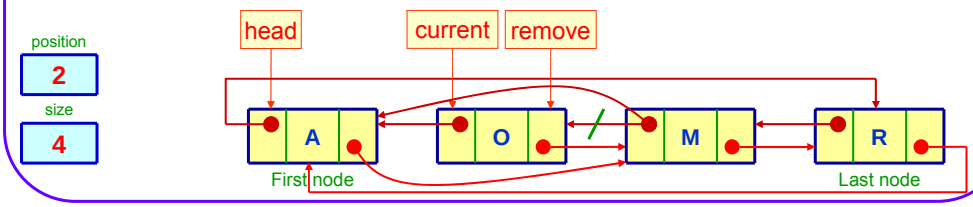
Mapping Operation: remove()
Example: remove()

```
current.getPrev().setNext(current.getNext());
current.getNext().setPrev(current.getPrev());
current = current.getNext();
```



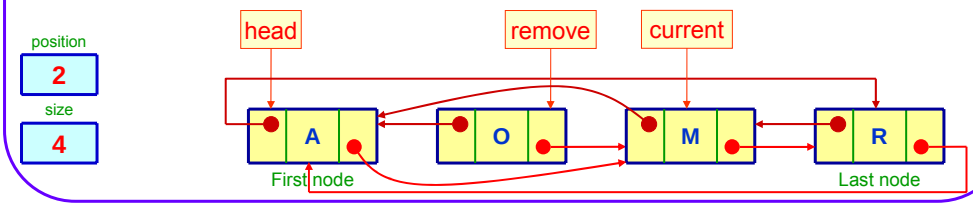
Mapping Operation: remove()
Example: remove()

current.getPrev().setNext(current.getNext());
current.getNext().setPrev(current.getPrev());
current = current.getNext();



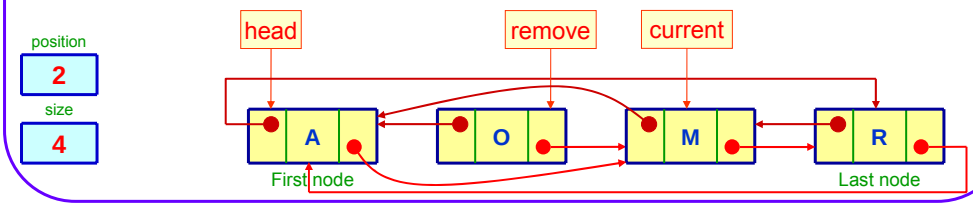
Mapping Operation: remove()
Example: remove()

current.getPrev().setNext(current.getNext());
current.getNext().setPrev(current.getPrev());
current = current.getNext();



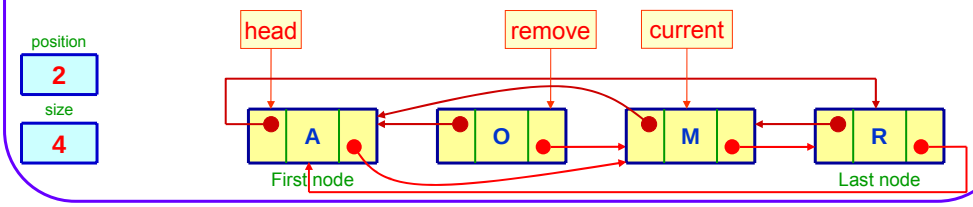
Mapping Operation: remove()
Example: remove()

```
if (remove.getNext() == head)
    position = 1;
```



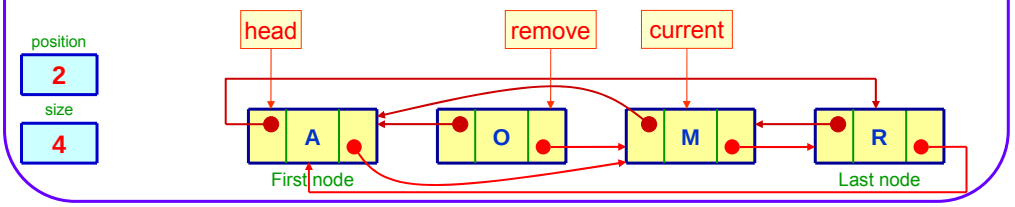
Mapping Operation: remove()
Example: remove()

```
if (size == 1)
{
    head = null;
    current = null;
}
```



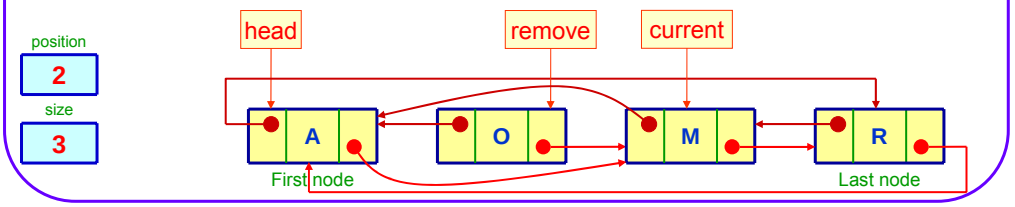
Mapping Operation: remove()
Example: remove()

```
else
   if (head == remove)
      head = current;
```



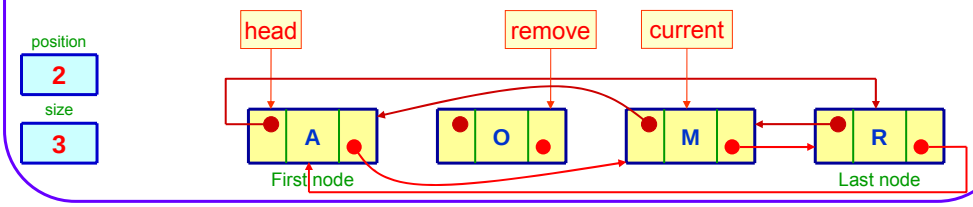
Mapping Operation: remove()
Example: remove()

```
size--;
remove.setNext(null);
remove.setPrev(null);
```



Mapping Operation: remove()
Example: remove()

size--;
remove.setNext(null);
remove.setPrev(null);



Mapping Operation: remove()
Example: remove()

size--;
remove.setNext(null);
remove.setPrev(null);