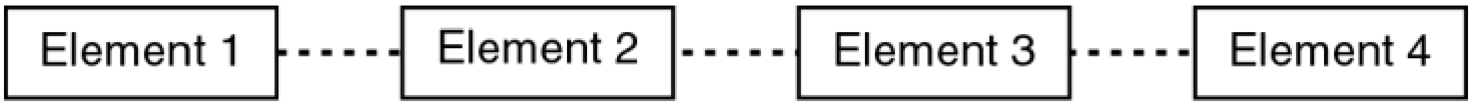
CS 242 – CS252
Data Structures

Linked lists – Part 1

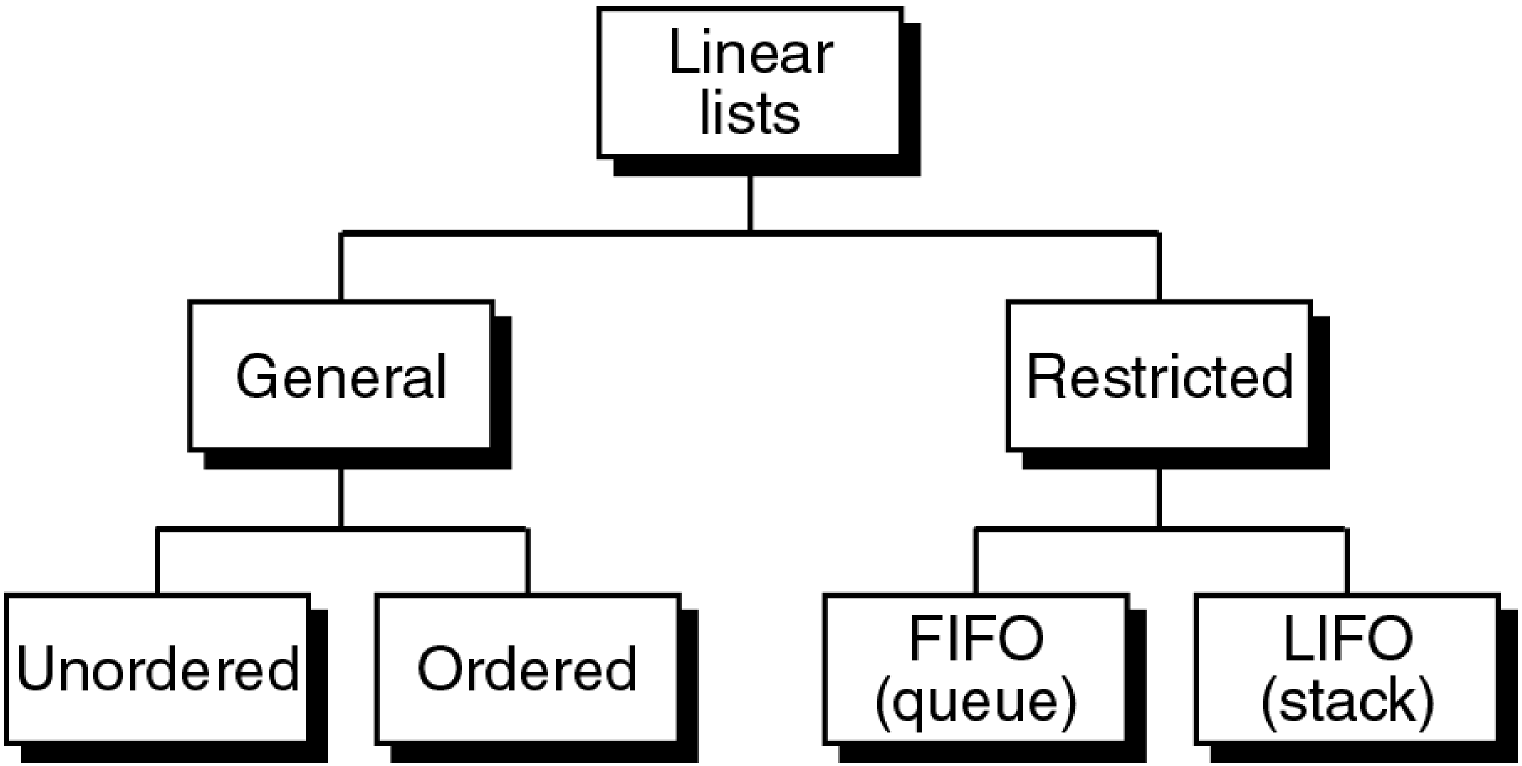
Linear List

- The sequential property of a linear list is basic to its definition and use.



- Example: Array, linked list.
- Array is the simplest linear list structure.
  - Length of arrays is fixed and unused spaces are wasted
  - Takes time and space to insert an item anywhere in the array

Linear List



Linear lists
General
Restricted
Unordered
Ordered
FIFO (queue)
LIFO (stack)

Linear List

- Linear lists can be categorized into:
  - General List
    - Data can be inserted and deleted anywhere
    - There are no restrictions on the operations that can be used to process the list.
  - Restricted List
    - Data can only be inserted or deleted at the ends of structures
    - Processing is restricted to operations on the data at the ends of the list

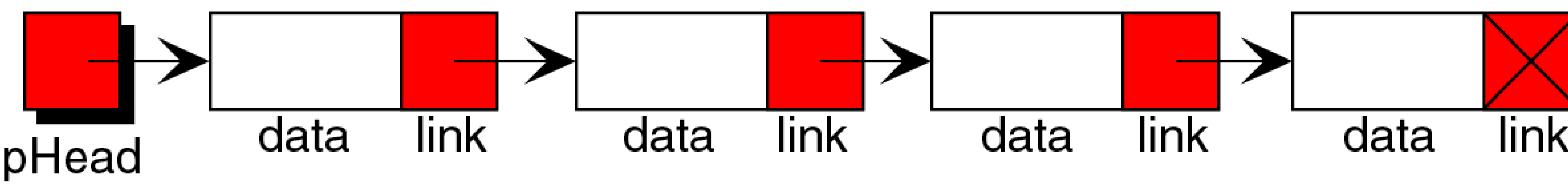
Linear List

- General List Structures
  - Un-ordered (Random) list
    - There is no ordering of the data.
  - Ordered list
    - The data arranged according a key.
    - A key is one or more fields within a structure that are used to identify the data or control their use.
- Restricted List Structures
  - FIFO list
    - The first in first out. Generally called a queue.
  - LIFO list
    - The last in first out. Generally called a stack.

Linked List

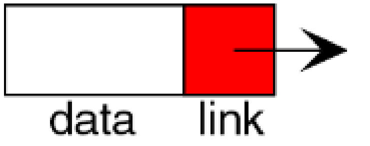
An ordered collection of data in which each element contains the location of the next element.

- The simple linked list is commonly known as a **singly linked list** because it contains only one link to a single successor.



Linked List

- Each element contains:
  - **Data** to be processed, useful information
  - **Link**, a pointer that identifies the next element in the list, used to chain the data together



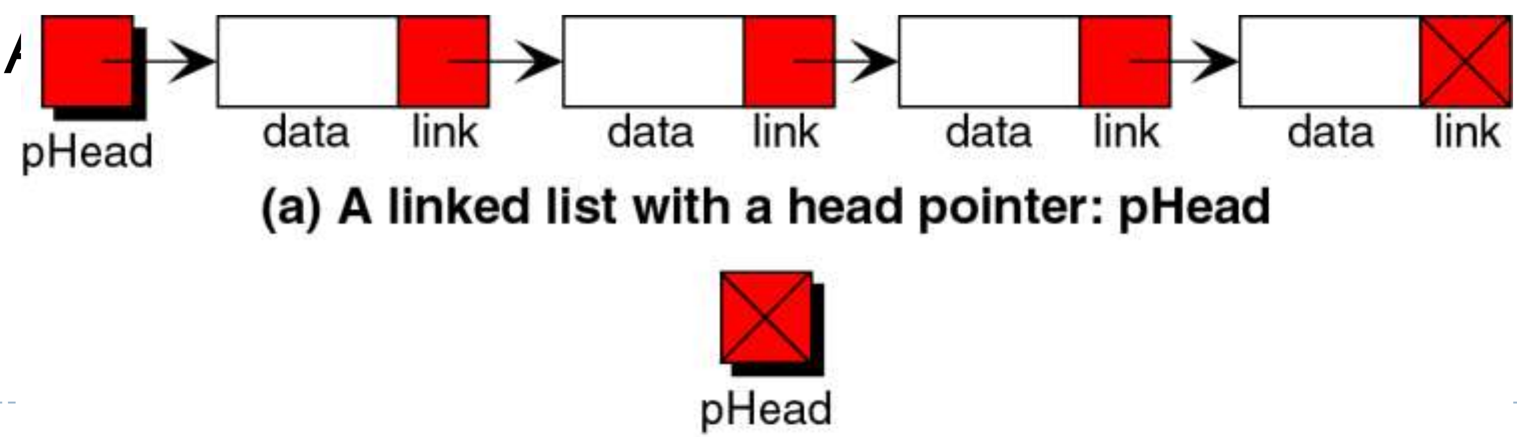
- Elements in Linked List are called **nodes.**
- Nodes are **self-referential** structures: contains a pointer member that points to a class object of the same class type.

Linked List

- Advantages
  - Data easily inserted and deleted
  - No need to shift elements of LL to make room for a new element or to delete an element
- Disadvantages
  - We are limited to a sequential search

Linked List Terminology

- Head pointer points to the beginning of the list
- A node’s successor is the next node in the sequence
- The last node has no successor. (link pointer = null)
- A node’s predecessor is the previous node in the the sequence
- The first node has no predecessor
- A list s length is the number of elements in it
- A



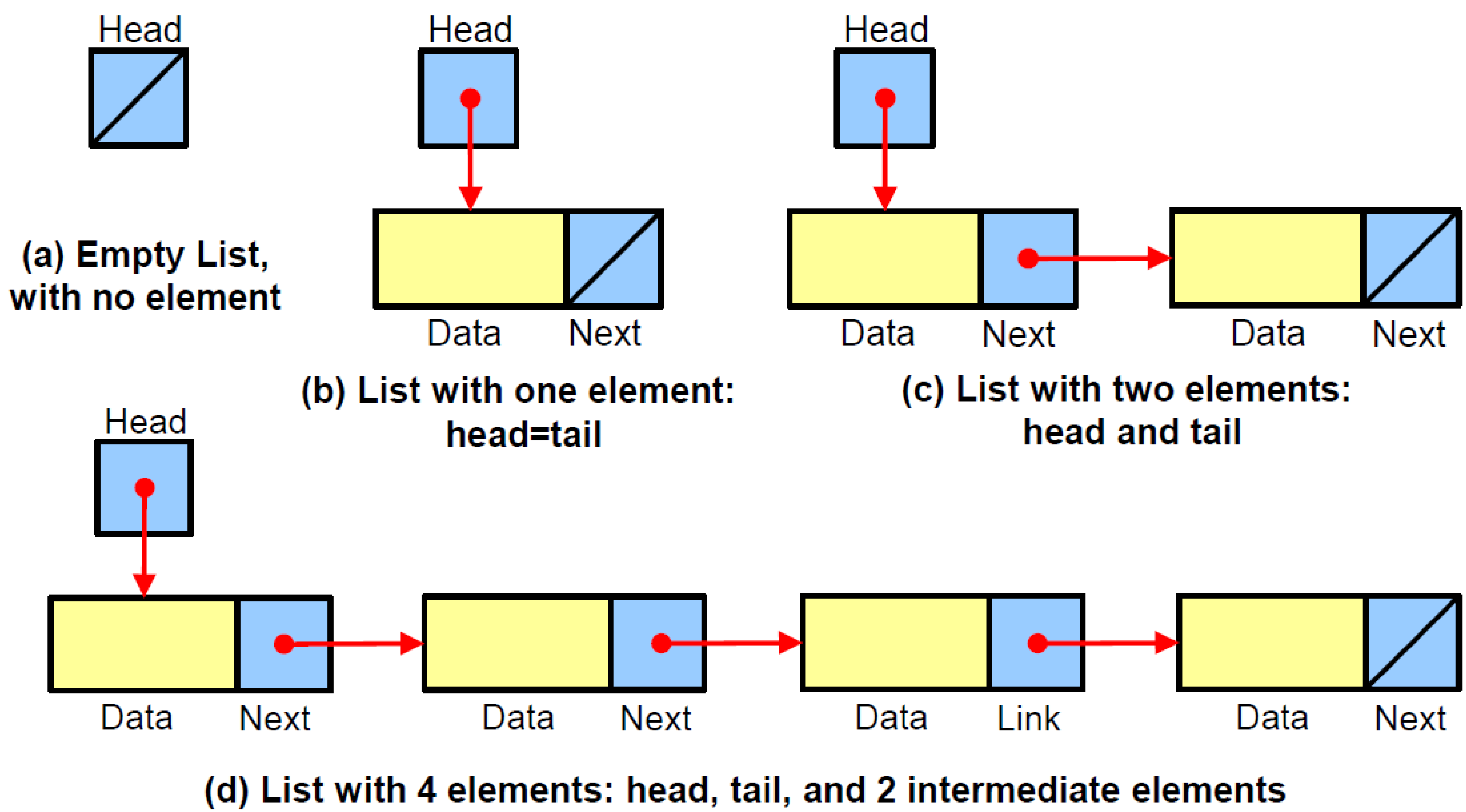
(a) A linked list with a head pointer: pHead

(b) An empty linked list

Linked List

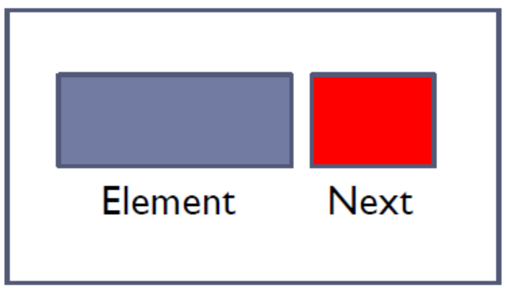
- ALWAYS REMEMBER TO SET *next* TO SOME VALUE: **EITHER ANOTHER NODE OR TO NULL!!!**

Linked List examples (Single)



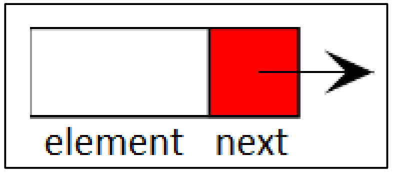
Linked List: Element (Data) Node

- Data Node Structure
  - Data type for the list elements depends entirely on the application.




```
public class Node <E>
{
        private E element;
        private Node<E> next;
        ……….
}
```

Linked List: Element Node (implementation)



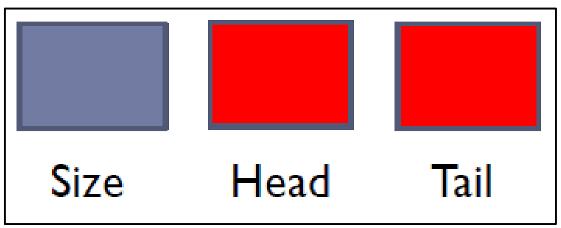

```java
public class Node <E>{

    private E element;
    private Node<E> next;

    public Node(E element, Node<E> next) {
        this.element = element;
        this.next = next;
    }

    public void setElement(E element) {
        this.element = element;
    }

    public void setNext(Node<E> next) {
        this.next = next;
    }

    public E getElement() {
        return element;
    }

    public Node<E> getNext() {
        return next;
    }
}
```

Linked List: Head Node

Head Node Structure

Contain:

1. Reference to head node (points to first node in a linked list or may be contain NULL if the linked list is empty)
2. Metadata which is a data about a list. Ex: count (size), reference to the last node (tail), current positions …etc.




```
public class SinglyLinkedList<E>
 {
        private Node<E> head;
        private Node<E> tail;
        private int size;
        ………….
}
```

Linked List: Head Node

- References to Linked Lists:
  - A linked list must always have a head pointer.
  - Depending on how to use the list you may have several other pointers. Ex:
    - Curr or pos points to a specific location
    - Tail or rear points to the last node

Singly Linked List ADT

A complete implementation of a SinglyLinkedListclass, supporting the following methods:

- **size( ):** Returns the number of elements in the list.
- **isEmpty( ):** Returns true if the list is empty, and false otherwise.
- **first( ):** Returns (but does not remove) the first element in the list.
- **last( ):** Returns (but does not remove) the last element in the list.
- **addFirst(e):** Adds a new element to the front of the list.
- **addLast(e):** Adds a new element to the end of the list.
- **removeFirst( ):** Removes and returns the first element of the list.

Singly Linked List ADT

Linked list operations:

- Basic operations: which means the operation should be to complete implementation of linked list.

1. Create List
2. Insert Node
3. Delete Node
4. Empty List
5. Size

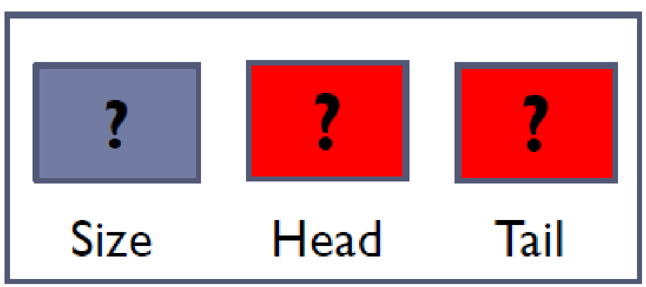
- Extra operations:

1. Search List
2. Traverse List
3. Retrieve Node
4. Destroy List

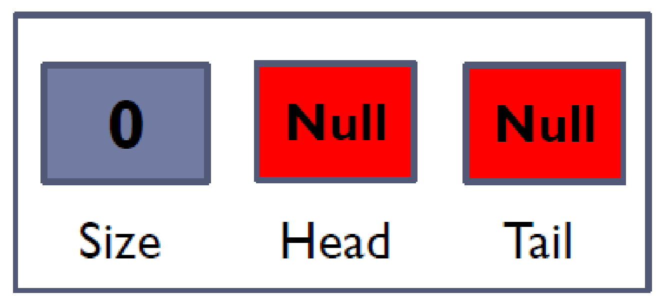
1- Create List

- Initializes the metadata for the list.

```
head = NULL
Tail = Null
Size = 0
```



A- Before create list



B- After create list

1- Create List (Constructor) implementation

```java
public class SinglyLinkedList<E> {
    private Node<E> head;
    private Node<E> tail;
    private int size;

    public SinglyLinkedList() {

        head=null;
        tail=null;
        size=0;
    }
// Operations on linked lists (Other methods)
}
```

2- Insert Node

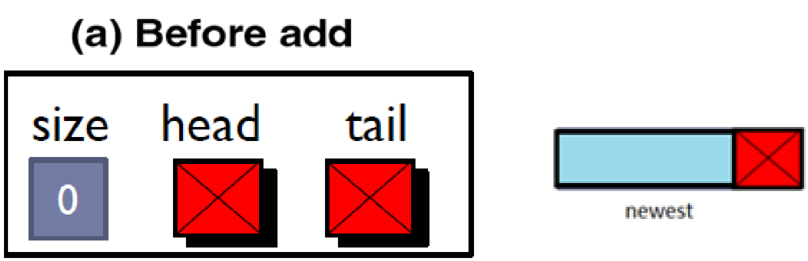
- We need only the logical predecessor to insert a node into the list.
- There are three steps for insertion:
  1. Allocate memory for the new node and insert data
  2. Point the new node to its successor.
  3. Point the new node's predecessor to the new node.

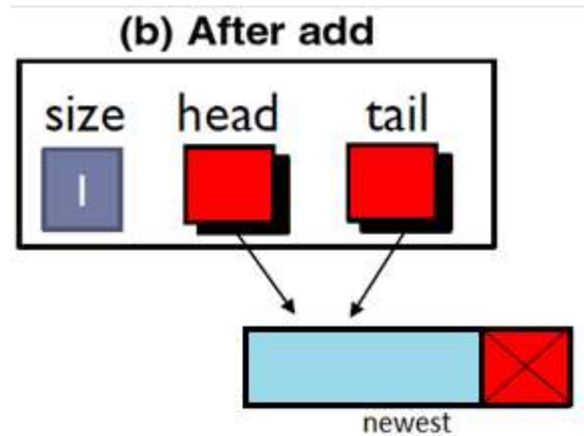
2- Insert Node

- Insertion cases:
  - Insert into empty list
  - Insert at beginning
  - Insert in middle
  - Insert at end

2- Insert Node

- Insert into ***empty*** list





Algorithm addFirst(e):

…

newest = Node(e)

newest.next= head

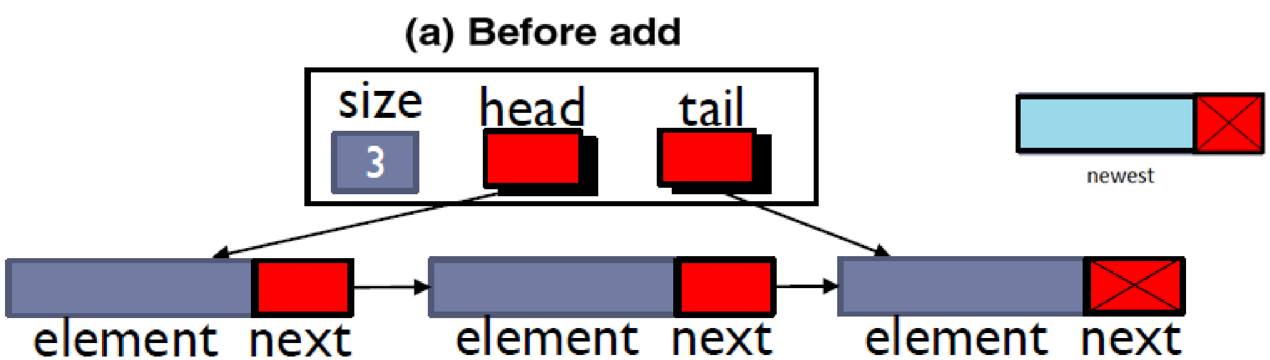
head = newest

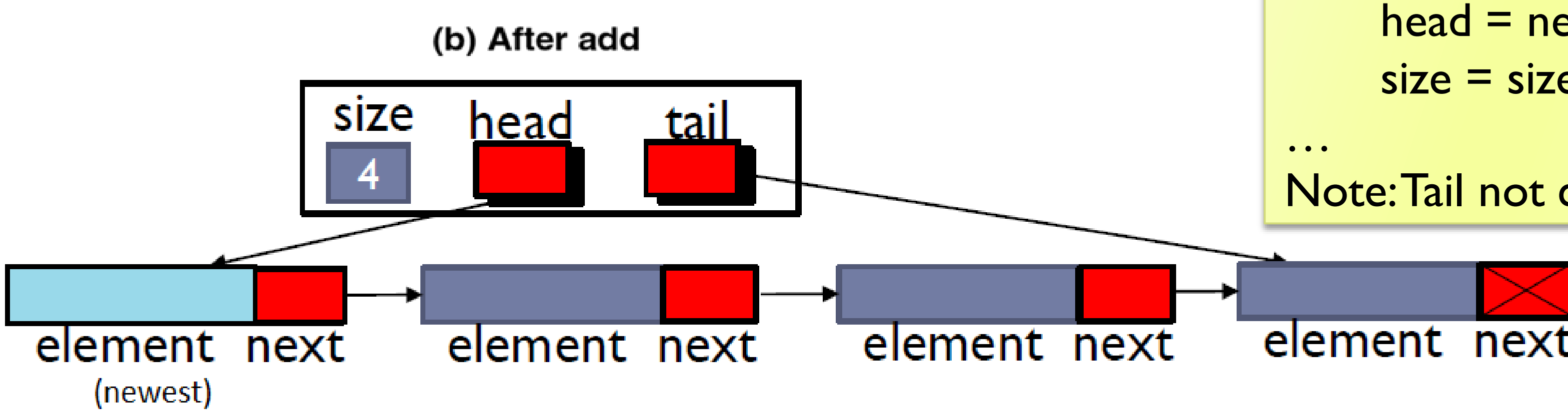
Tail=head

size = size+1

2- Insert Node

- Insert at ***beginning (non-empty list)***





Algorithm addFirst(e):

…

newest = Node(e)

newest.next= head

head = newest

size = size+1

…

Note: Tail not changed

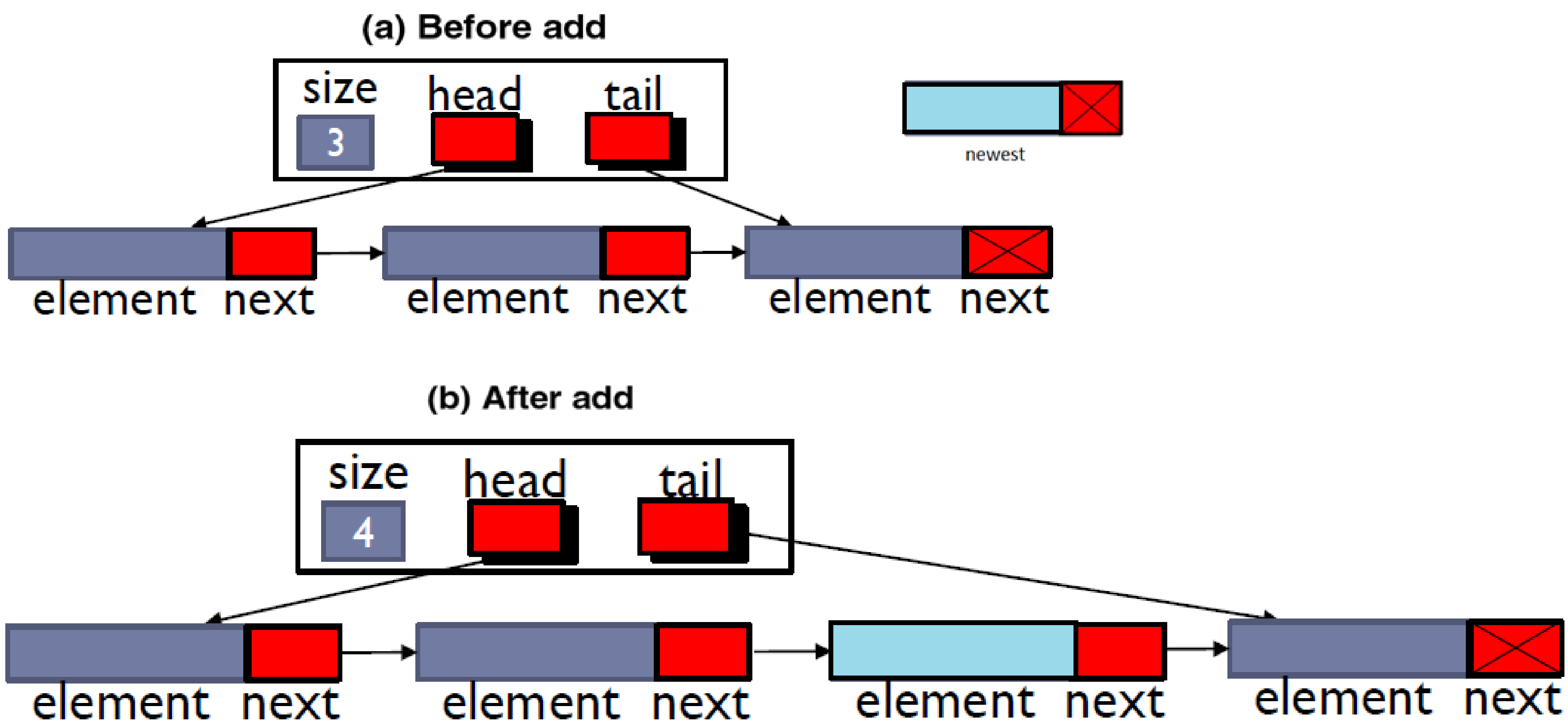
Insert Node

- Insert into an empty LL or at the beginning of the LL is an insertion at the head.

```
public void addFirst(E e) { // adds element e to the front of the list
 head = new Node<>(e, head); // create and link a new node
 if (size == 0)
 tail = head; // special case: new node becomes tail also
 size++;
}
```

2- Insert Node

- Insert in ***middle***

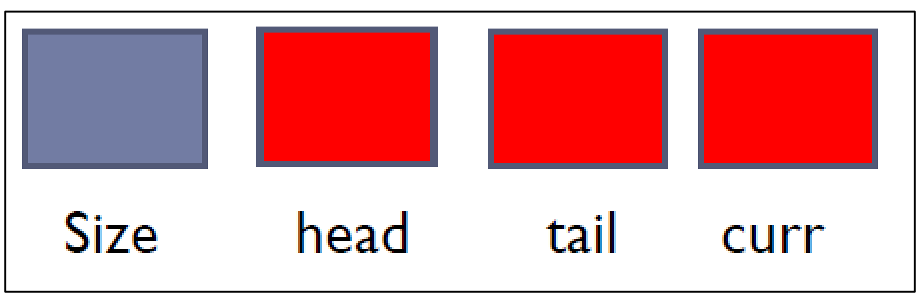


2- Insert Node (in middle)

- add(e,i): adding new node at index i
  - We have to make sure that the value of index i is within the range
- We need a reference to traverse the list
  - Contain:

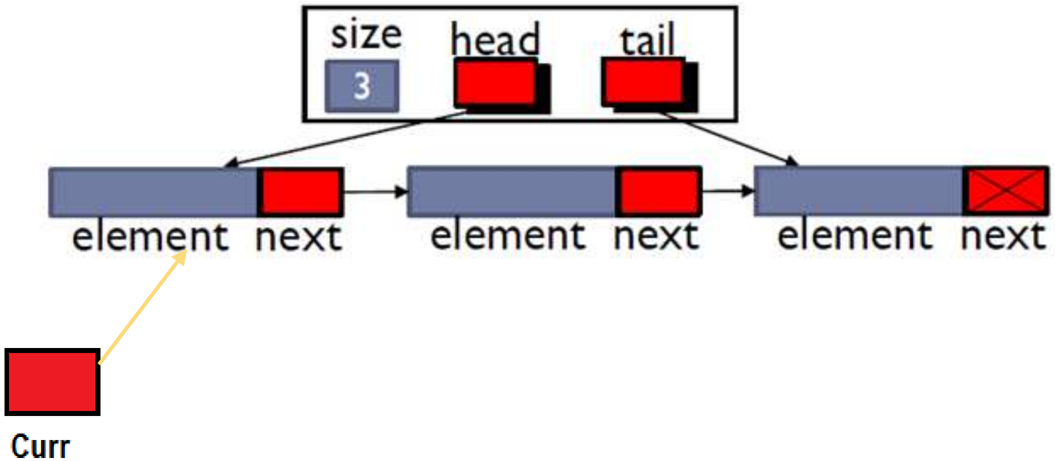
1. Reference to head node.
2. Reference to tail node.
3. Curr reference to a specific location.

```
public class SinglyLinkedList<E> {
        private Node<E> head;
        private Node<E> tail;
        private Node<E> curr;
        private int size;
        ………….
}
```

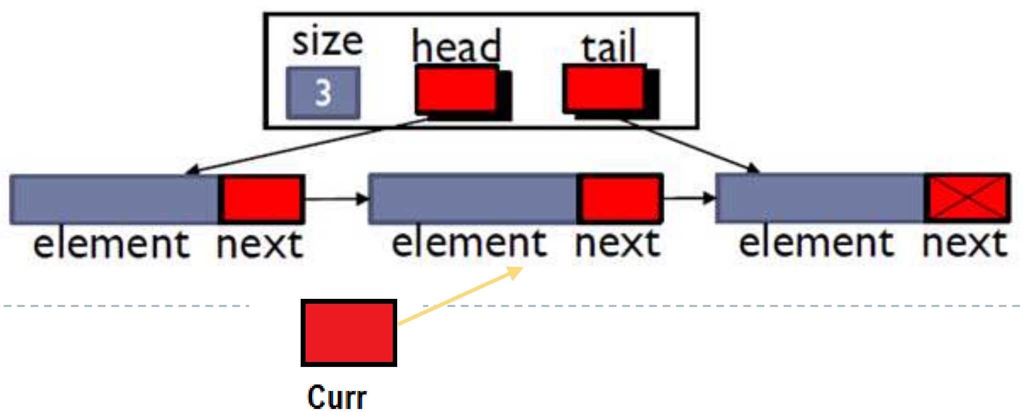


2- Insert Node (in middle)

1.

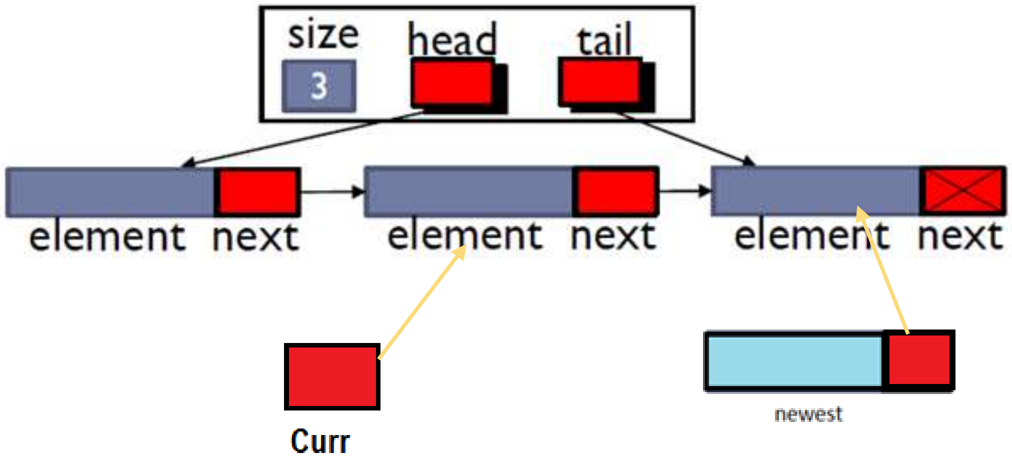


2. First, move (curr) until index-1

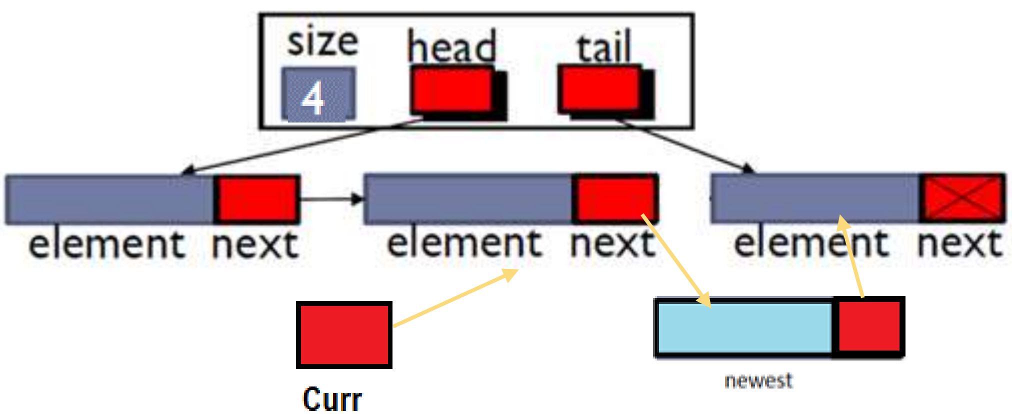


2- Insert Node (in middle)

3. Second, connect the new node to the successor of the



4. Finally, connect current node to the new node and increment the size

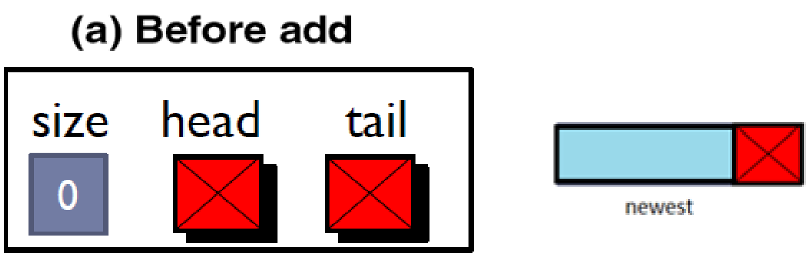


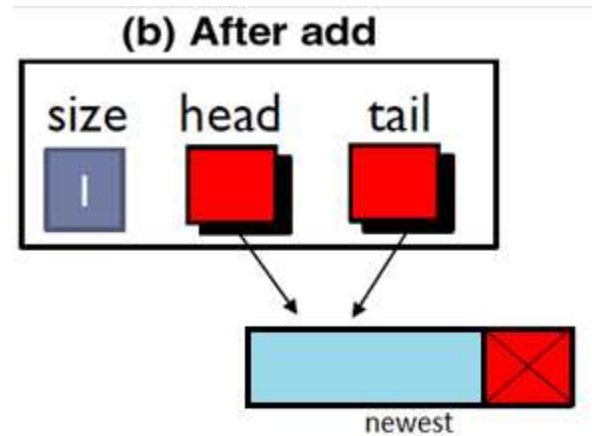
2- Insert Node (in middle)

```
public void add(E element,int index)
{
        if(index<0 || index>size)
        {   System.out.println("Out of bound!");
            return;
        }
        Node<E> newest=new Node<E>(element,null);
        if(index==0) // add at front
        {   newest.setNext(head);
            head=newest;
            if(tail==null)
                tail=head;
        }
        else // add the middle
        {   curr=head;
            for(int i=0;i<index-1;i++)
            {
                curr=curr.getNext();
            }
            newest.setNext(curr.getNext());
            curr.setNext(newest);
            if(tail==curr)
                tail=tail.getNext();
        }
        size++;
}
```

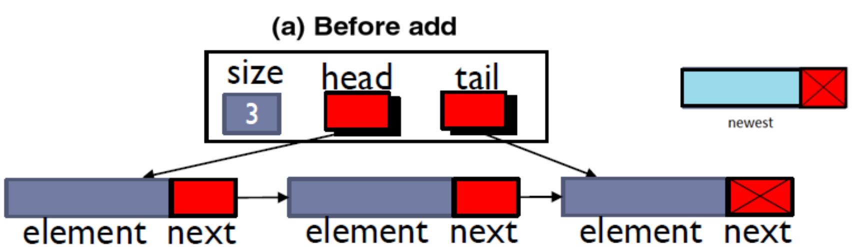
2- Insert Node

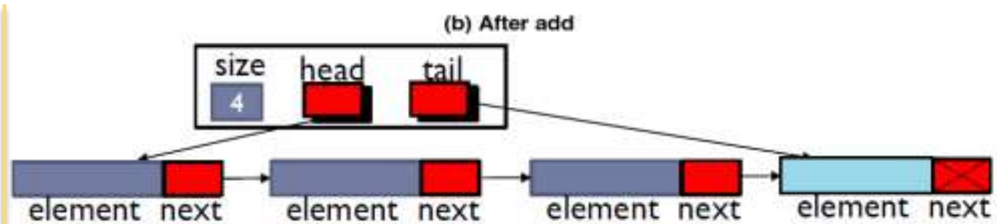
- Insert at ***end***
  - empty list





  - Non – empty list





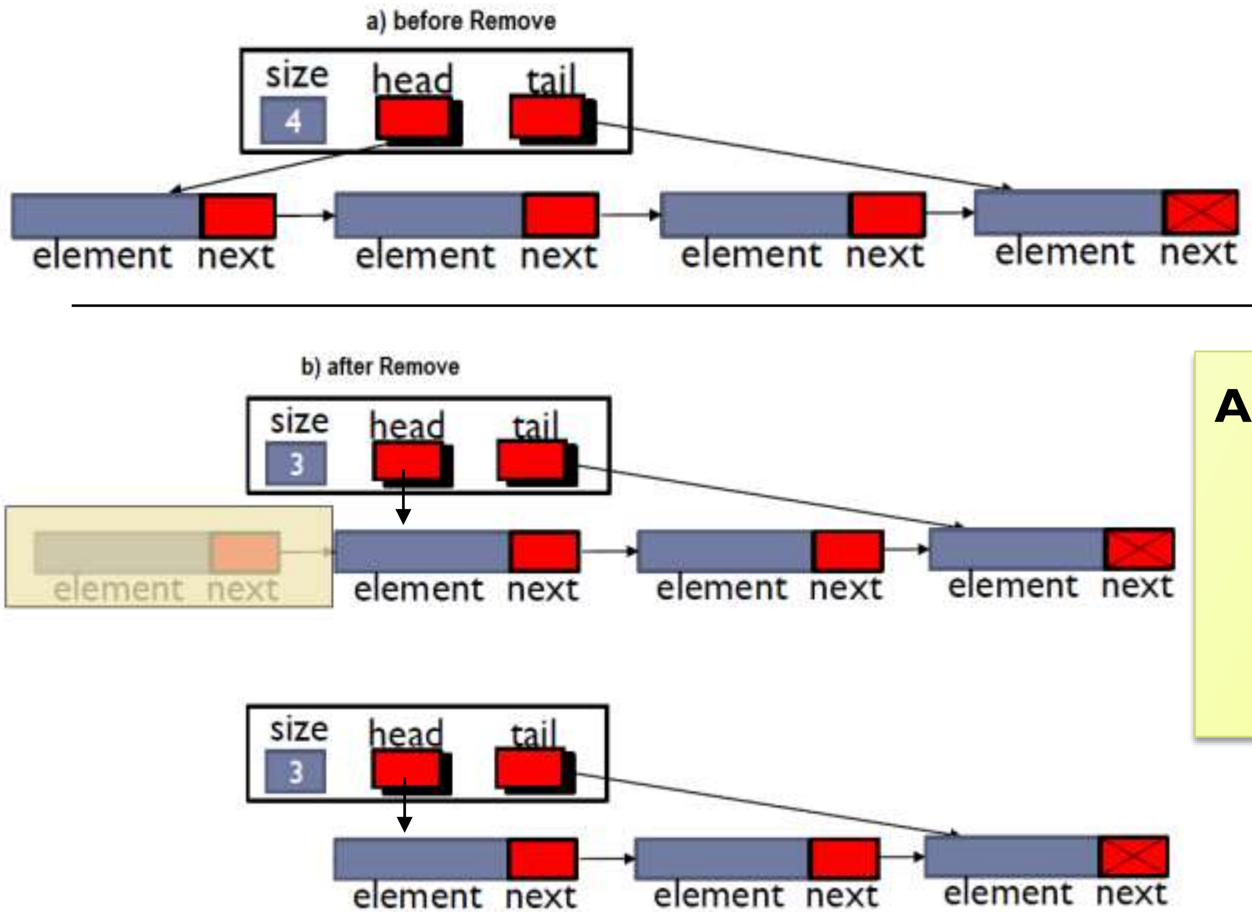
2- Insert Node at end

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

```
public void addLast(E e) { // adds element e to the end of the list
Node<E> newest = new Node<>(e, null); // node will eventually be the tail
if (isEmpty( ))
head = newest; // special case: previously empty list
else
 tail.setNext(newest); // new node after existing tail
 tail = newest; // new node becomes the tail
 size++;
}
}
```

3- Delete (Remove) Node

A. Remove ***beginning*** (at head)



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

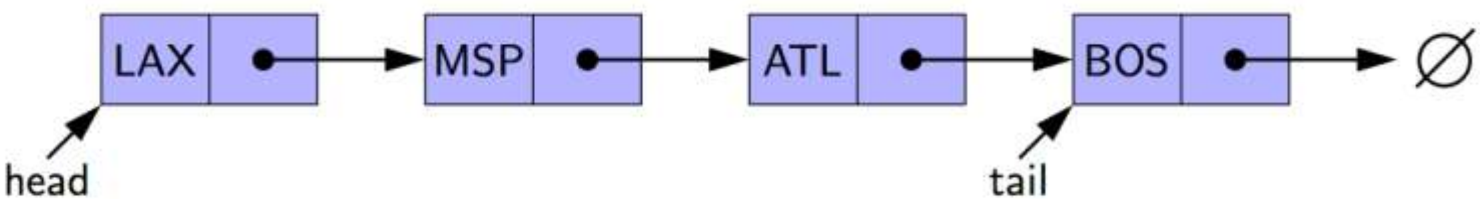
3- Delete (Remove) Node.

```java
public E removeFirst( ) { // removes and returns the first element
if (isEmpty( )) return null; // nothing to remove
E answer = head.getElement( );
head = head.getNext( ); // will become null if list had only one node
size--;
if (size == 0)
tail = null; // special case as list is now empty
return answer;
}
```

3- Delete (Remove) Node

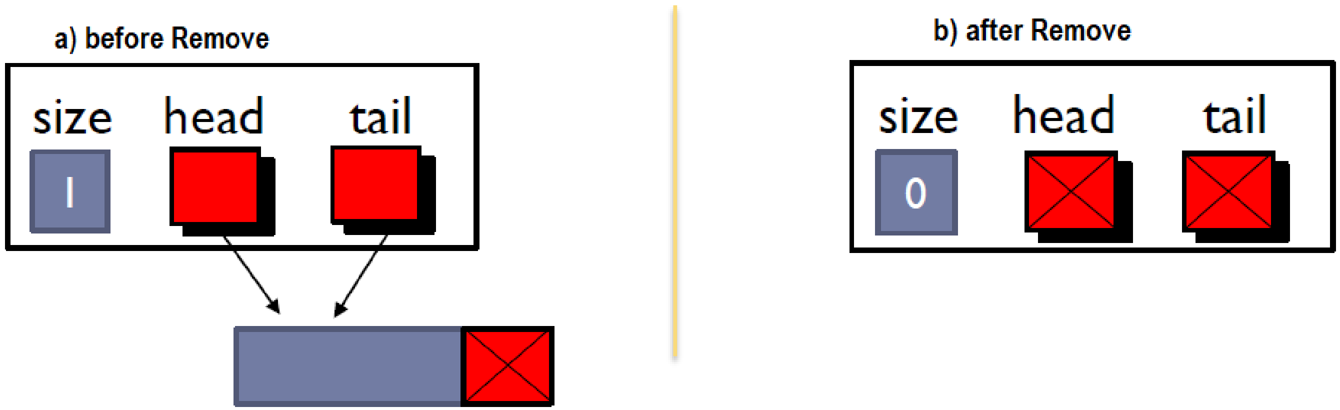
B. Remove ***End or last*** (at tail)

- Removing at the tail of a singly linked list is not efficient!
  - we must be able to access the node before the last node.
  - The only way to access this node is to start from the head of the list and search all the way through the list which is time consuming.



3- Delete (Remove) Node

C. Deleting from a list of only one element



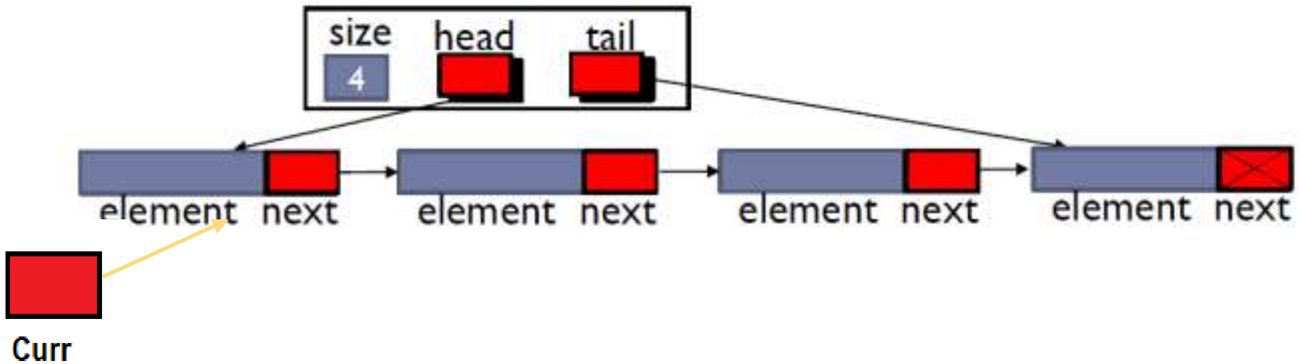
Head = tail = null
size = 0

3- Delete (Remove) Node

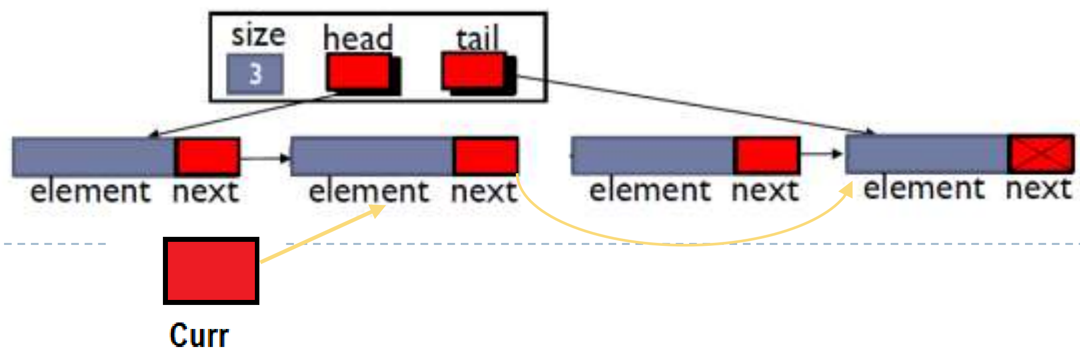
D. Remove from *middle* (at index i)

- **remove(e,i):** removing node at index iand return its element.
- Similar to add(e,i) in slide 26.

1. First, move curr until index -1



2. Second, connect the node at i-1 with the node at i+1



3- Delete (Remove) Node

- Remove from ***middle*** (at index i)

```
public E remove(int index)
{
        if(index<0 || index>=size)
        {   System.out.println("Out of bound!");
            return null;
        }
        E element;
        if(index==0) // remove from front
        {   element=head.getElement();
            head=head.getNext();
            if(head==null)
                tail=null;
        }
        else
        {   curr=head;
            for(int i=0;i<index-1;i++)
            { curr=curr.getNext();}
            element=curr.getNext().getElement();
            if(tail== curr.getNext())
            {   tail=curr;}
            curr.setNext(curr.getNext().getNext());
        }

        size--;
        return element;
}
```

4- Empty list & 5- Size

Algorithm size():
return size

AlgorithmisEmpty()
return head==null

Singly Linked List ADT
Node class

```java
public class Node <E>{

    private E element;
    private Node<E> next;

    public Node(E element, Node<E> next) {
        this.element = element;
        this.next = next;
    }

    public void setElement(E element) {
        this.element = element;
    }

    public void setNext(Node<E> next) {
        this.next = next;
    }

    public E getElement() {
        return element;
    }

    public Node<E> getNext() {
        return next;
    }
}
```

Singly Linked List ADT
SLL class

```
// instance variables of the SinglyLinkedList
public class SinglyLinkedList<E> {

    private Node<E> head;
    private Node<E> tail;
    private int size;

    public SinglyLinkedList() {

        head = null;
        tail = null;
        size = 0;
    }
```

Singly Linked List ADT
SLL class

```
// access methods
    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public E first() { // returns (but does not remove) the first element
        if (isEmpty()) {
            return null;
        }
        return head.getElement();
    }

    public E last() { // returns (but does not remove) the last element
        if (isEmpty()) {
            return null;
        }
        return tail.getElement();
    }
```

Singly Linked List ADT
SLL class

```
// update methods
    public void addFirst(E e) { // adds element e to the front of the list
        head = new Node<>(e, head); // create and link a new node
        if (size == 0) {
            tail = head; // special case: new node becomes tail also
        }
        size++;
    }

    public void addLast(E e) { // adds element e to the end of the list
        Node<E> newest = new Node<>(e, null); // node will eventually be the tail
        if (isEmpty()) {
            head = newest; // special case: previously empty list
        } else {
            tail.setNext(newest); // new node after existing tail
        }
        tail = newest; // new node becomes the tail
        size++;
    }
```

Linked Lists: Performance Analysis

Method	Big-O
size( )	O(1)
isEmpty( )	O(1)
first( )	O(1)
last( )	O(1)
addFirst(e)	O(1)
addLast(e)	O(1)
removeFirst( )	O(1)

Using SinglyLinkedList

```java
public static void main(String[] args) {
    // TODO code application logic here
    SinglyLinkedList<String> MyFirstList=new SinglyLinkedList<String>();

    System.out.println("List size="+MyFirstList.size());

    MyFirstList.addLast("Fahad");
    MyFirstList.addLast("Khlid");
    MyFirstList.addLast("Norah");
    MyFirstList.addLast("Sara");

    System.out.println("List size="+MyFirstList.size());
    System.out.println("First element="+MyFirstList.first());
    System.out.println("Last element="+MyFirstList.last());

    MyFirstList.removeFirst();
    MyFirstList.removeFirst();

    System.out.println("List size="+MyFirstList.size());
    System.out.println("First element="+MyFirstList.first());
    System.out.println("Last element="+MyFirstList.last());
}
```

Example: Output

```
run:
List size=0
List size=4
First element=Fahad
Last element=Sara
List size=2
First element=Norah
Last element=Sara
BUILD SUCCESSFUL (total time: 0 seconds)
```