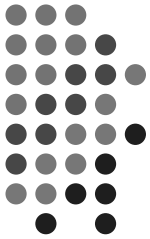
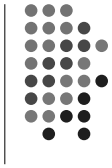
Lists
Organizing Data Linearly

2B

Linked Lists

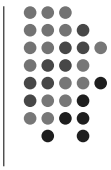




Disadvantages of Array Lists

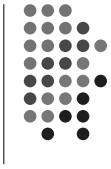
- If a data entry is added to or removed from an array-based list, data needs to be shifted to update the list.
- In the worst case, for an array-based list with n data entries, an add and a remove takes O(n) time.
- Also, all data in the array-based list must be stored sequentially in memory. Large lists will require significant contiguous blocks of memory.

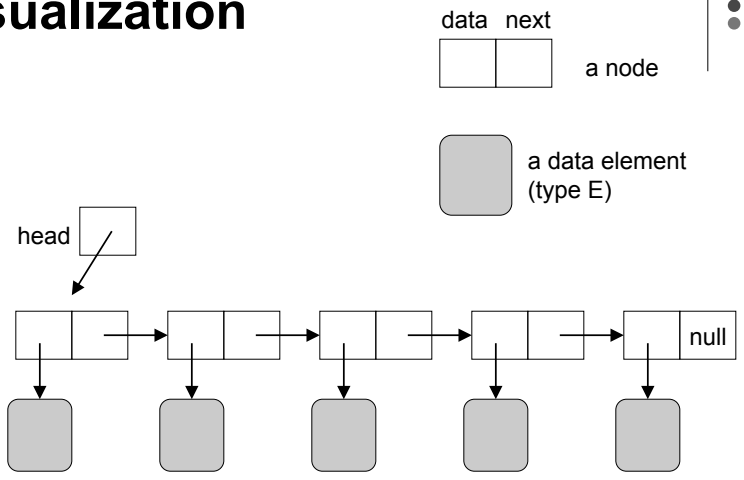
Singly Linked Lists



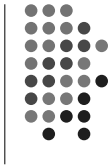
- Each **data** entry is stored in its own **node**.
- Each node has a reference to the node that contains the **next** data entry.
- A reference named the **head** points to the node with the first data entry.
- The last node in the list contains a reference of null since it does not point to any other data.
- An empty list would have a head reference equal to null.

Visualization



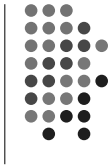


Inner classes



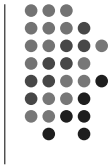
- Since a node is specific to a linked list, we will define the `Node` class to be an **inner class** of the `SinglyLinkedList` class.
- The inner class is only accessible by the class that enclosed it.
- Fields defined in the inner class are accessible by its enclosing class. (No accessors are needed.)

Node class



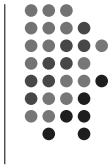
```
private static class Node<E> {
    private E data;
    private Node<E> next;
    private Node(E element) {
       data = element;
       next = null;
    }
    private Node(E element, Node<E> nodeRef) {
       data = element;
       next = nodeRef;
    }
}
```

Initializing an empty linked list



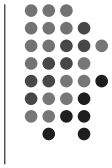
```
public class SinglyLinkedList<E> {

  private Node<E> head;
  private int numElements;

  public SinglyLinkedList() {
      head = null;
      numElements = 0;
  }
```


Add new first list element
Get size of list



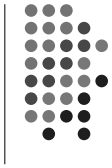
```
  public void addFirst(E element) {
      head = new Node<E>(element, head);
      numElements++;
  }

  public int size() {
      return numElements;
  }
```

Removing the first node



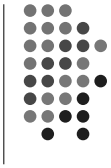
```
public E removeFirst() {
    if (head == null)
         throw new NoSuchElementException();
    E result = head.data;
    head = head.next;
    numElements--;
    return result;
}
```


Traversing the list



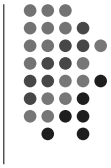
```
public String toString() {
    String result = "";
    Node<E> nodeRef = head;
    while (nodeRef != null) {
         result += nodeRef.data + " ==> ";
         nodeRef = nodeRef.next;
    }
    return result;
}
```

Add at given index

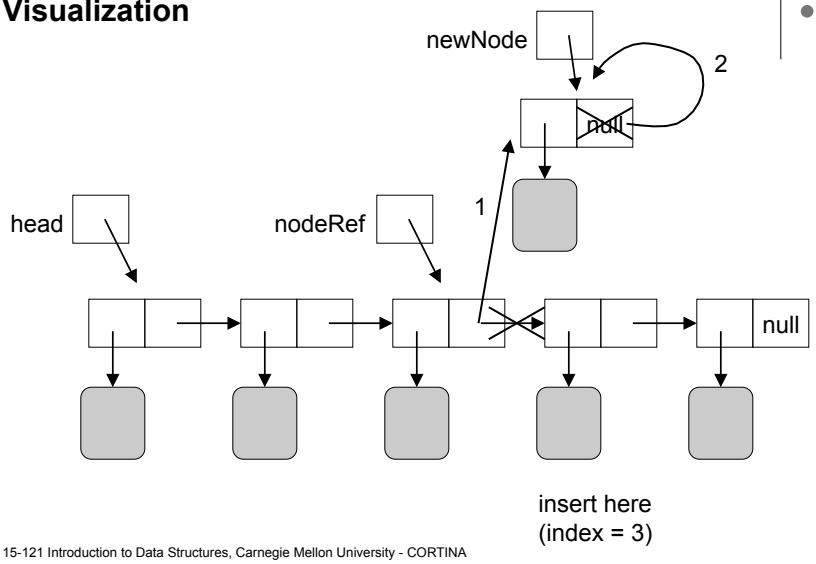


```
public void add(int index, E element) {
  if (index < 0 || index > size())
     throw new
         IndexOutOfBoundsException();

  if (index == 0) {
         addFirst(element);
         return;
  }
                                  // cont'd
```


Add at given index WRONG

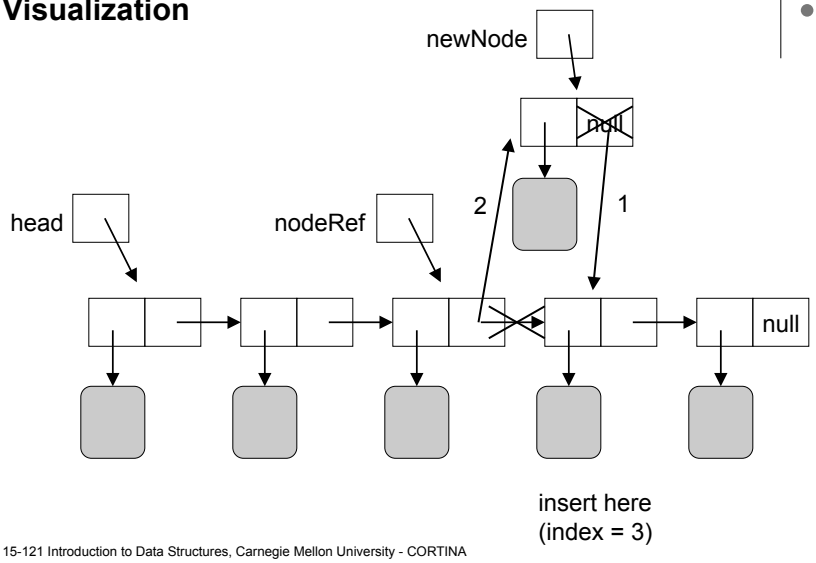


Visualization



Add at given index RIGHT

Visualization

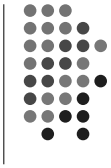


Add at given index

(cont'd)

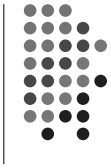
```
    Node<E> newNode
      = new Node<E>(element);
    Node<E> nodeRef = head;
    for (int i = 1; _____________; i++)
        nodeRef = nodeRef.next;
    newNode.next = nodeRef.next;
    nodeRef.next = newNode;
    numElements++;
}
```

Remove at given index



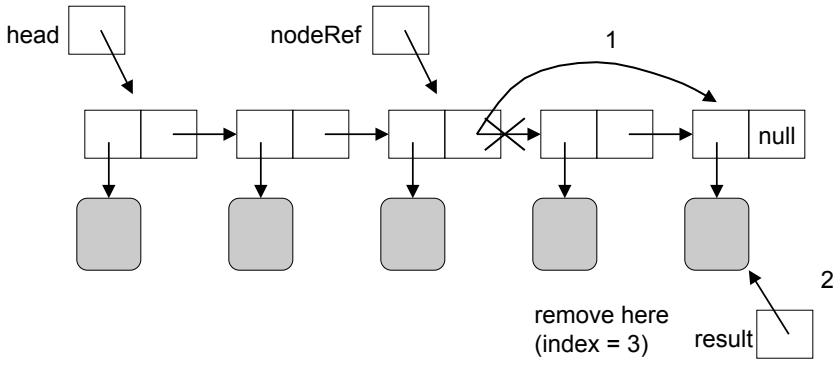
```
public E remove(int index) {
    if (index < 0 || index >= size())
       throw new IndexOutOfBoundsException();

    if (index == 0) {
        return removeFirst();
    }

                                    // cont'd
```


Remove at given index

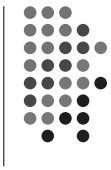


Visualization

WRONG

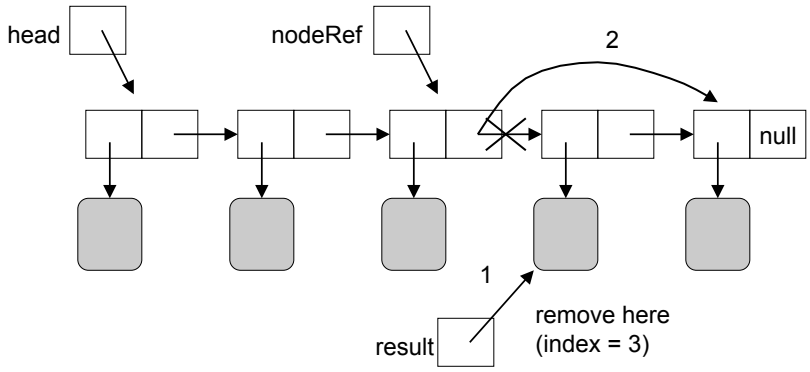


Remove at given index

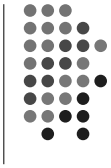


Visualization

RIGHT

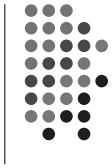


Remove at given index
(cont'd)



```
    Node<E> nodeRef = head;
    for (int i = 1; _______________; i++)
        nodeRef = nodeRef.next;

    E result = nodeRef.next.data;

    nodeRef.next = nodeRef.next.next;
    numElements--;
    return result;
}
```

Complexity



On a singly linked list with n nodes:

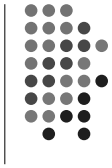
`addFirst` _____

`removeFirst` _____

`add` _____

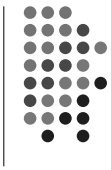
`remove` _____

Disadvantages of Singly Linked Lists



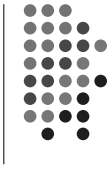
- Insertion into a list is generally linear.
- In order to insert a node at an index greater than 0, we need a reference to the previous node.
- In order to remove a node that is not the first node, we need a reference to the previous node.
- We can only traverse the list in one direction.

Doubly-Linked Lists

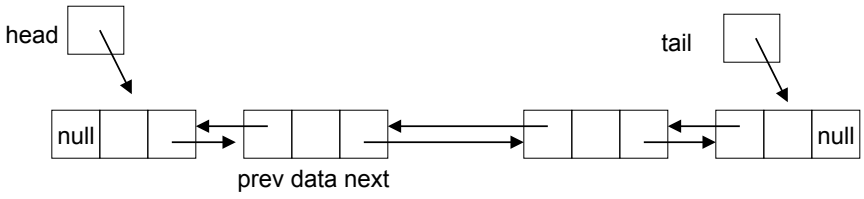


- Each **data** entry is stored in its own **node**.
- Each node has a reference to the node that contains the **next** data entry and a reference to the node that contains the previous data entry (**prev**).
- A reference named the **head** points to the node with the first data entry and a reference named the **tail** points to the node with the last data entry.
- The last node in the list contains a next reference of null and the first node in the list contains a prev reference of null.
- An empty list would have head and tail references equal to null.

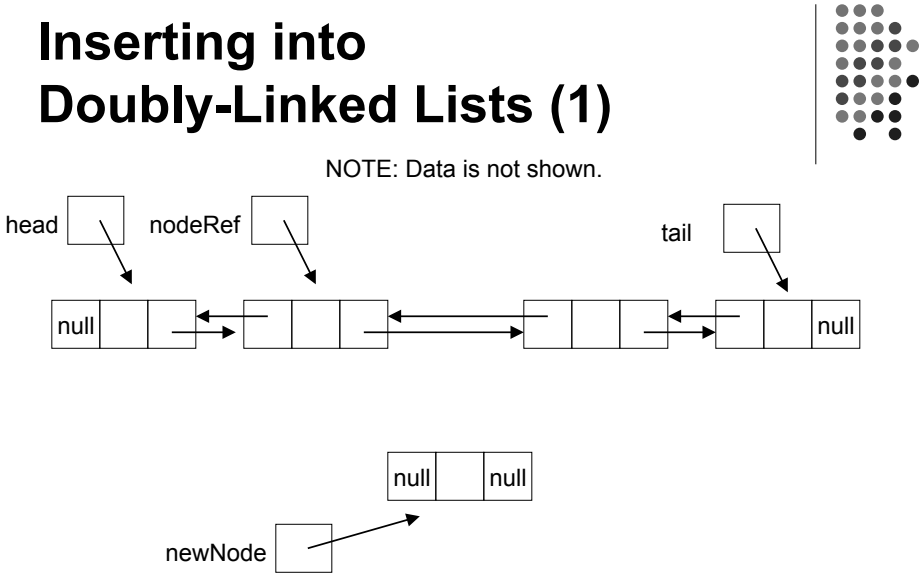
Doubly-Linked Lists



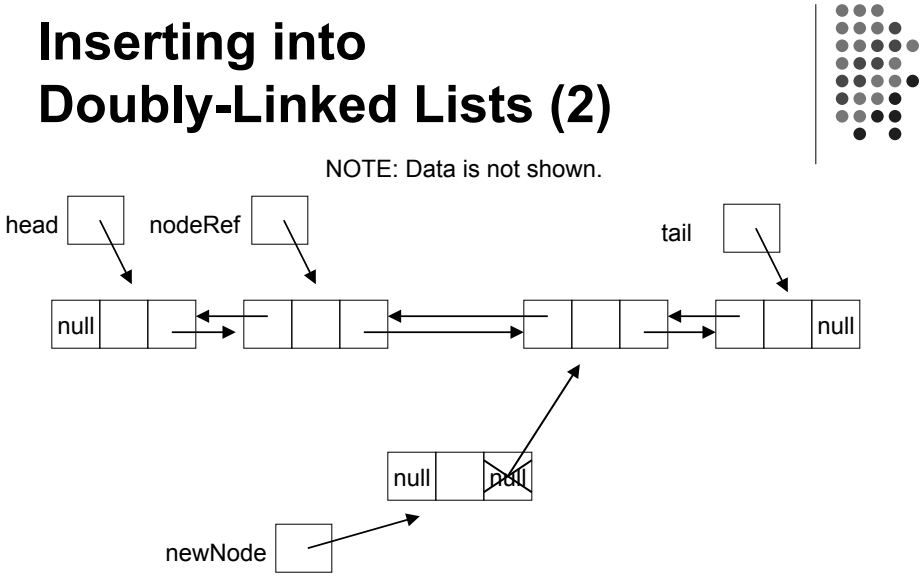
NOTE: Data is not shown.



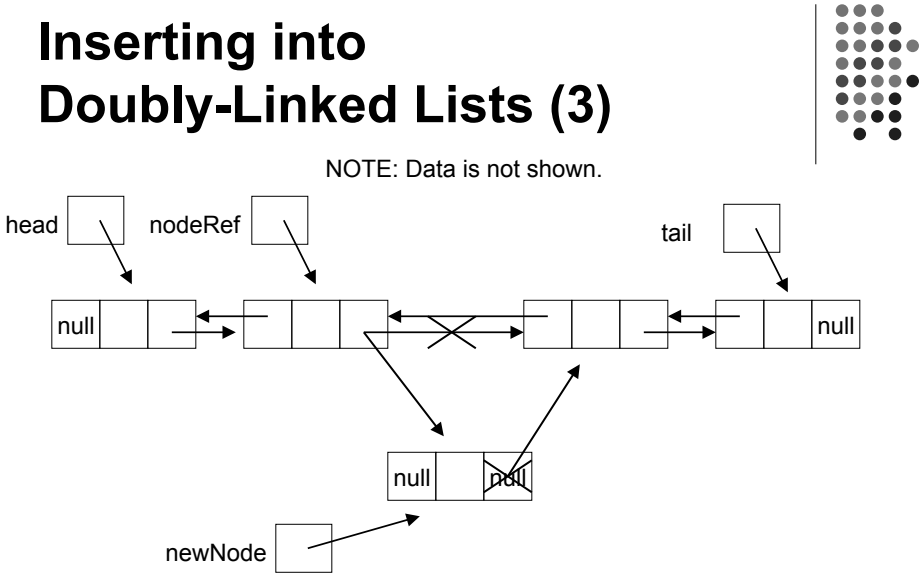
Inserting into Doubly-Linked Lists (1)



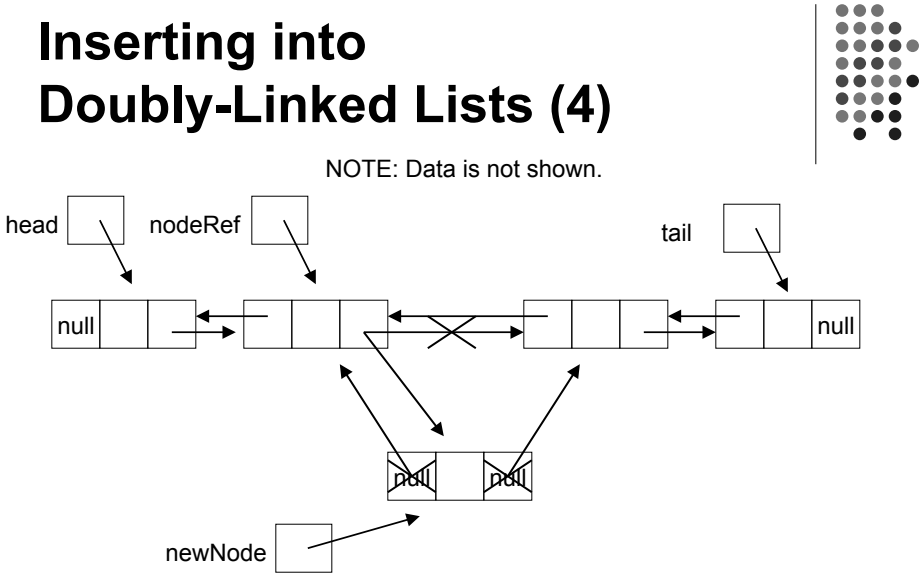
Inserting into Doubly-Linked Lists (2)



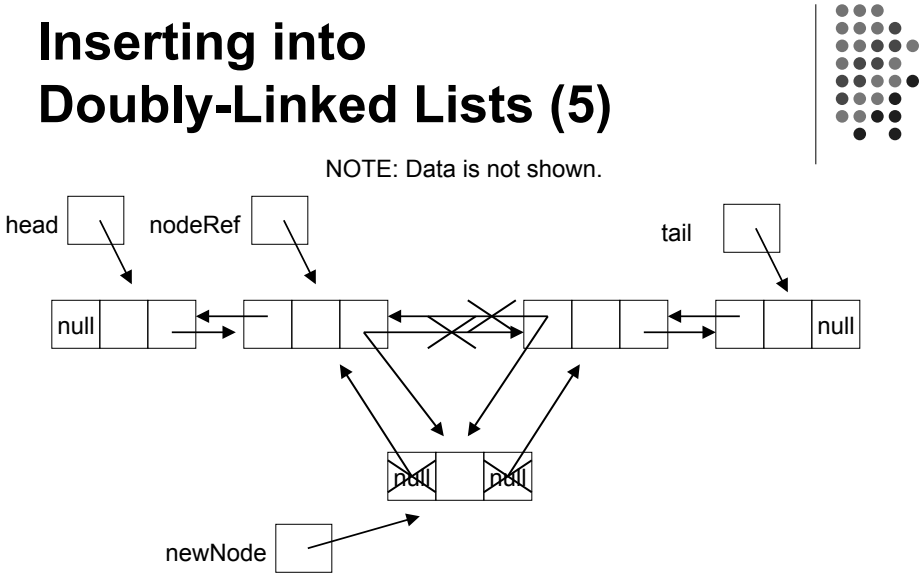
Inserting into Doubly-Linked Lists (3)



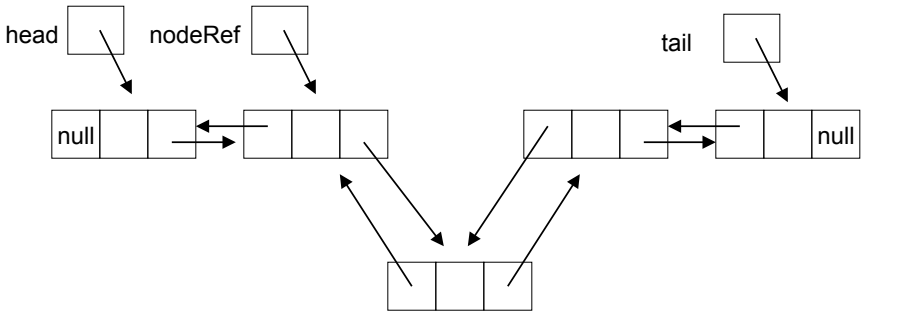
Inserting into Doubly-Linked Lists (4)



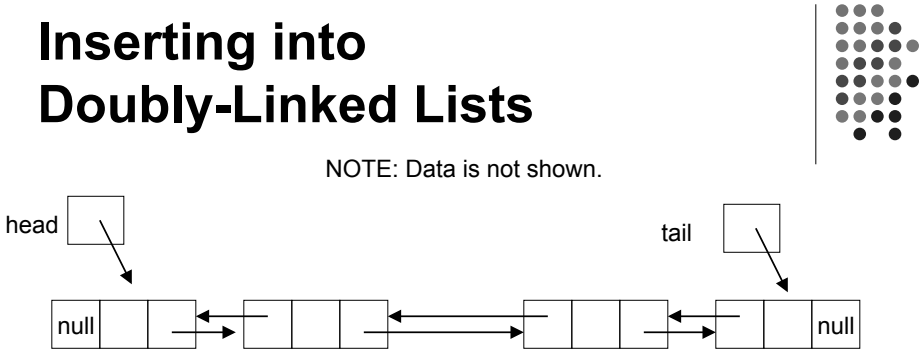
Inserting into Doubly-Linked Lists (5)



Inserting into Doubly-Linked Lists



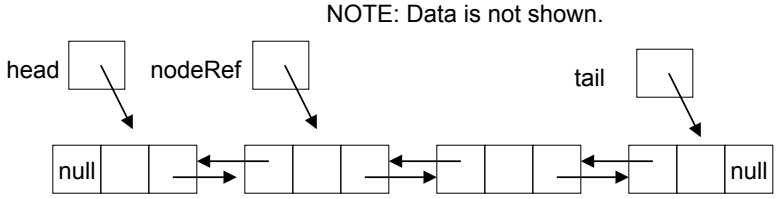
Inserting into Doubly-Linked Lists



How would you implement addFirst?

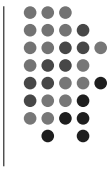
How would you implement addLast?

Removing from Doubly-Linked Lists

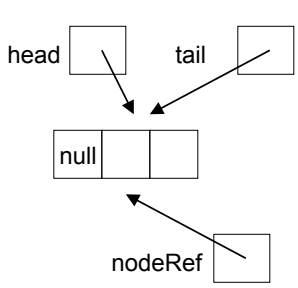


How would you remove the node referenced by nodeRef?

Removing from Doubly-Linked Lists

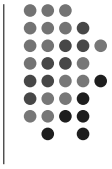


NOTE: Data is not shown.



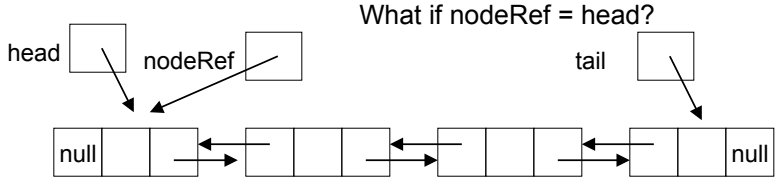
What if head = tail?

Removing from Doubly-Linked Lists

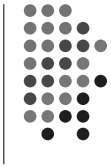


NOTE: Data is not shown.

What if nodeRef = head?



Removing from Doubly-Linked Lists



NOTE: Data is not shown.