# Chapter Data Structures
# Lists, Stacks, Queues

## Limitations of arrays

- Once an array is created, its size cannot be altered.
- Array provides inadequate support for inserting, deleting, sorting, and searching operations.

## Four Classic Data Structures

Four classic dynamic data structures to be introduced in this chapter are lists, stacks, queues, and binary trees. A list is a collection of data stored sequentially. It supports insertion and deletion anywhere in the list. A stack can be perceived as a special type of the list where insertions and deletions take place only at the one end, referred to as the top of a stack. A queue represents a waiting list, where insertions take place at the back (also referred to as the tail of) of a queue and deletions take place from the front (also referred to as the head of) of a queue. A binary tree is a data structure to support searching, sorting, inserting, and deleting data efficiently.

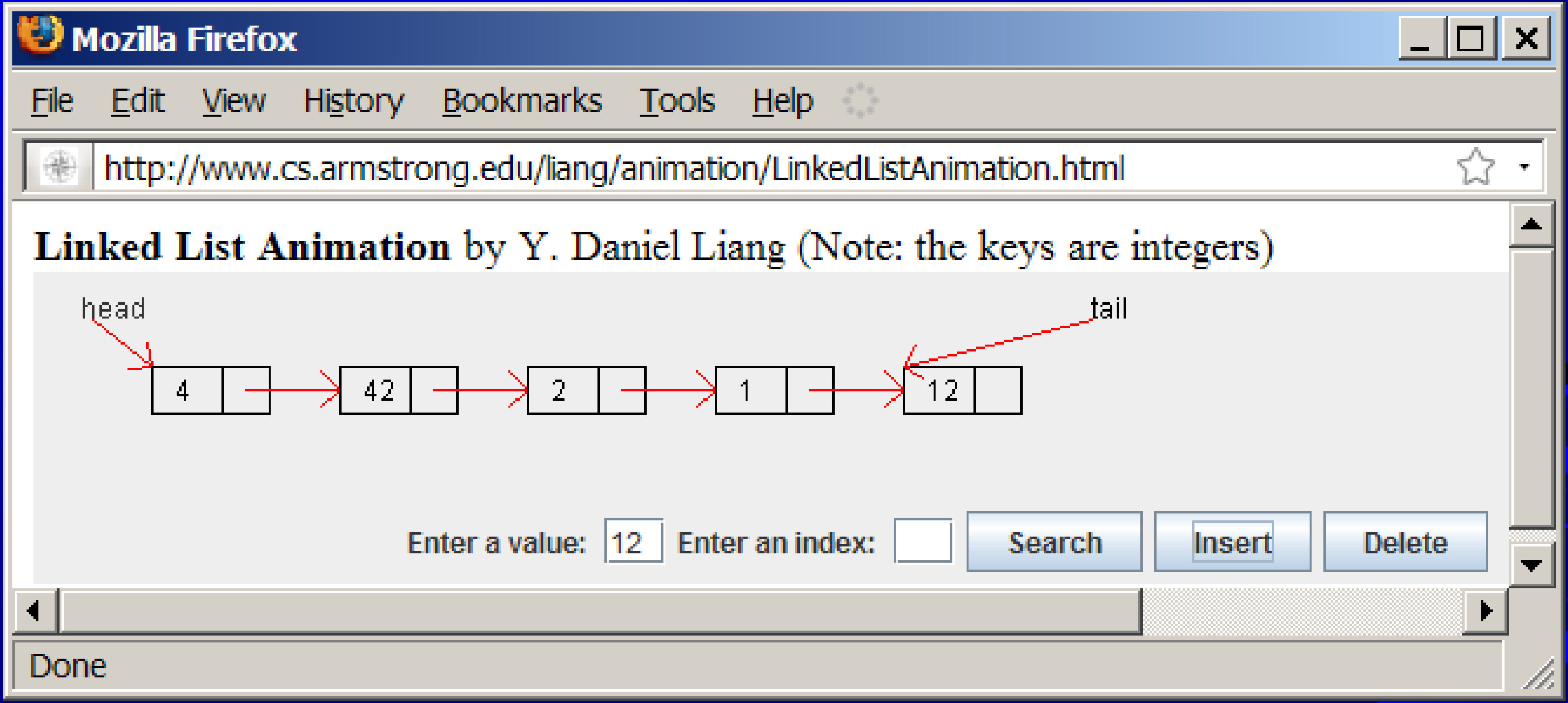
## Lists

A list is a popular data structure to store data in sequential order. For example, a list of students, a list of available rooms, a list of cities, and a list of books, etc. can be stored using lists. The common operations on a list are usually the following:

- Retrieve an element from this list.
- Insert a new element to this list.
- Delete an element from this list.
- Find how many elements are in this list.
- Find if an element is in this list.
- Find if this list is empty.

# Linked List Animation

[www.cs.armstrong.edu/liang/animation/LinkedListAnimation.html](www.cs.armstrong.edu/liang/animation/LinkedListAnimation.html)

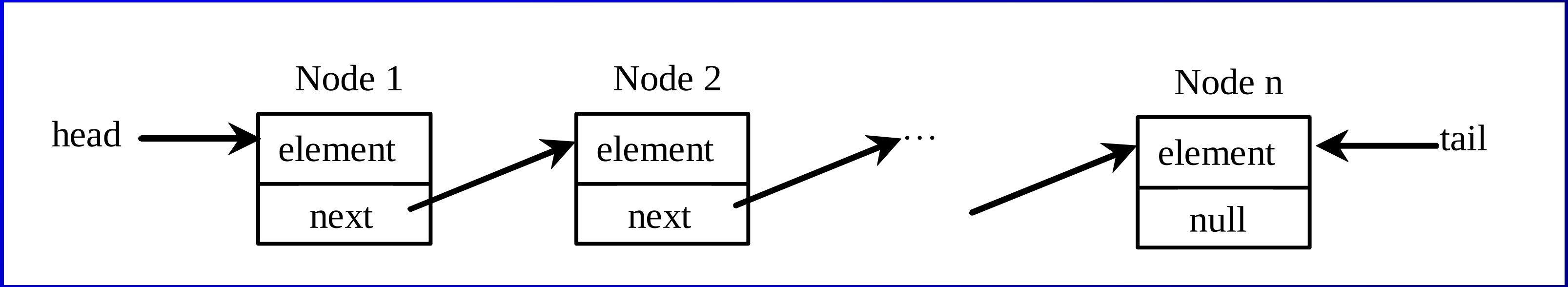

Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.cs.armstrong.edu/liang/animation/LinkedListAnimation.html
head
tail
4
42
2
1
12
Search
Insert
Delete
Done

**Linked List Animation** by Y. Daniel Liang (Note: the keys are integers)

Enter a value: 12 Enter an index:

# Nodes in Linked Lists

A linked list consists of nodes. Each node contains an element, and each node is linked to its next neighbor. Thus a node can be defined as a class, as follows:



```
class Node {
  int element;
  Node next;

  public Node(int o) {
    element = o;
  }
}
```

## Adding Three Nodes

The variable head refers to the first node in the list, and the variable tail refers to the last node in the list. If the list is empty, both are null. For example, you can create three nodes to store three strings in a list, as follows:

Step 1: Declare head and tail:

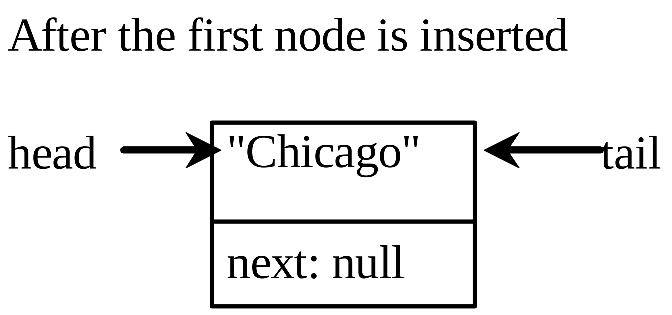
```
Node<String> head = null;
Node<String> tail = null;
```

The list is empty now

## Adding Three Nodes, cont.

Step 2: Create the first node and insert it to the list:

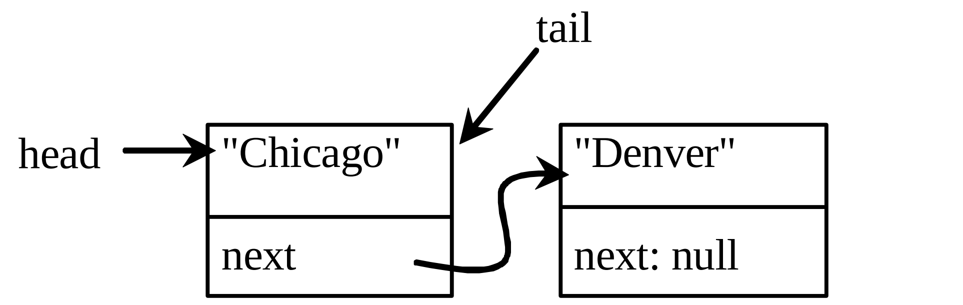
```
head = new Node<String>("Chicago");
tail = head;
```



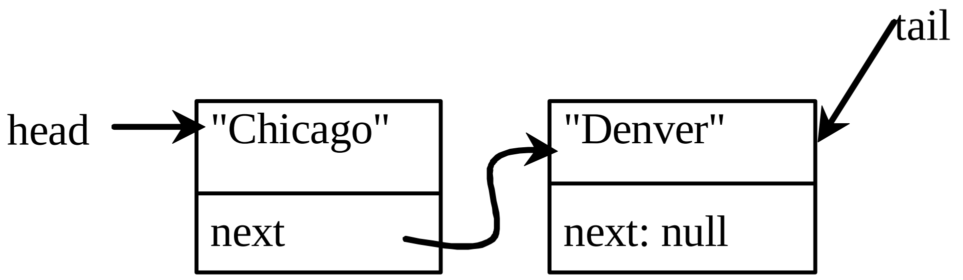
# Adding Three Nodes, cont.

Step 3: Create the second node and insert it to the list:

```
tail.next = new Node<String>("Denver");
```



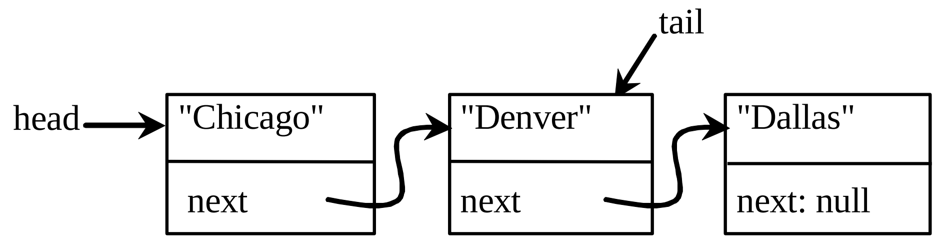
```
tail  = tail.next;
```



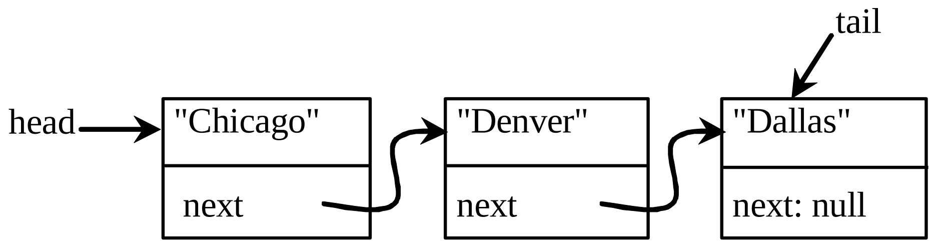
# Adding Three Nodes, cont.

## Step 4: Create the third node and insert it to the list:

```
tail.next =
  new Node<String>("Dallas");
```



```
tail = tail.next;
```

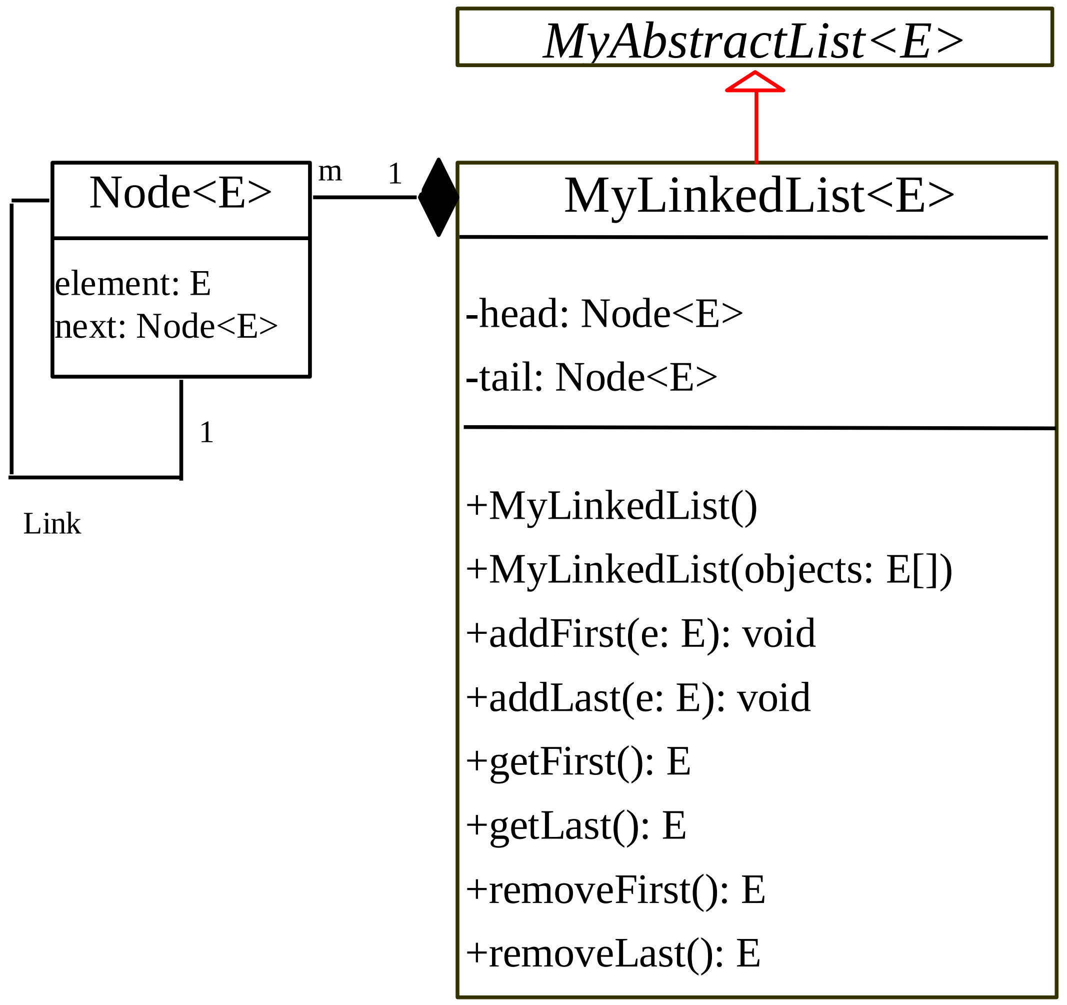


## Traversing All Elements in the List

Each node contains the element and a data field named *next* that points to the next element. If the node is the last in the list, its pointer data field next contains the value null. You can use this property to detect the last node. For example, you may write the following loop to traverse all the nodes in the list.

```
Node<E> current = head;
while (current != null) {
  System.out.println(current.element);
  current = current.next;
}
```

# MyLinkedList

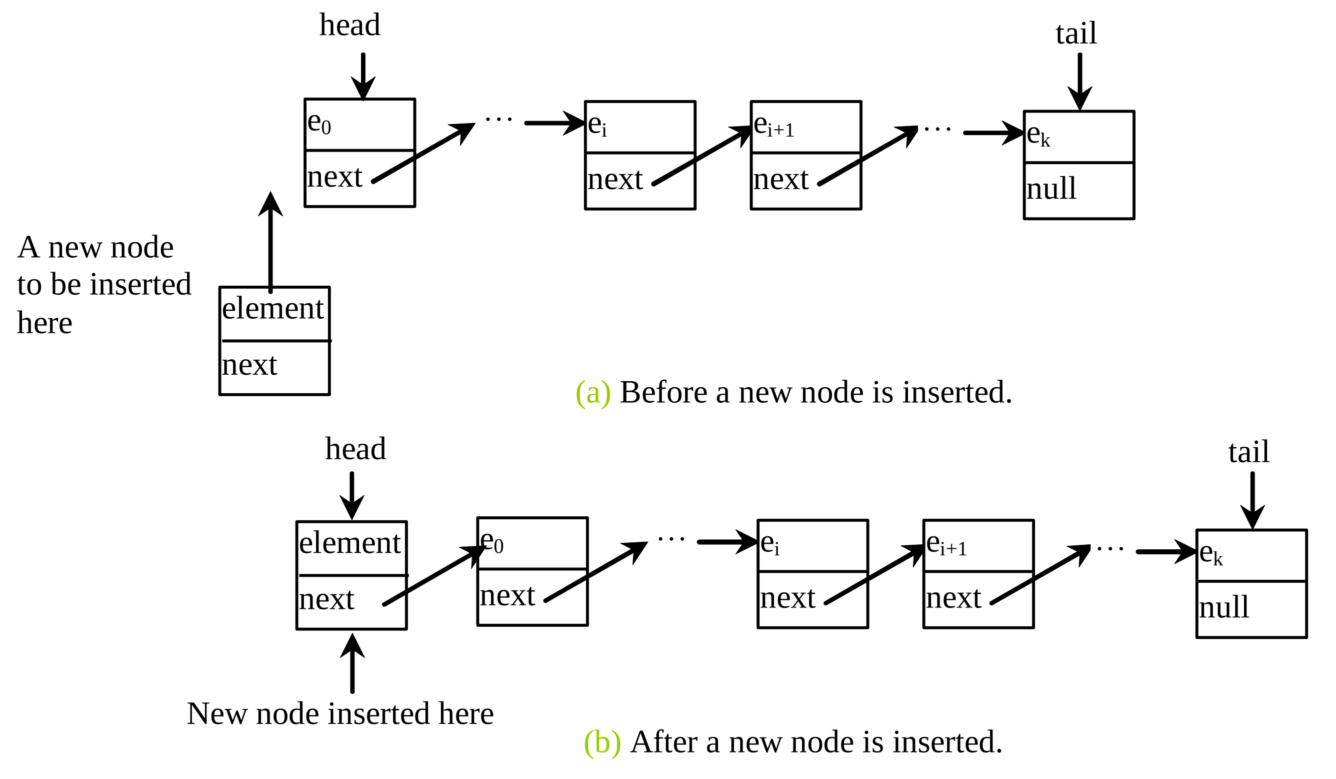


| Method | Description |
|---|---|
| +MyLinkedList() | Creates a default linked list. |
| +MyLinkedList(objects: E[]) | Creates a linked list from an array of objects. |
| +addFirst(e: E): void | Adds the object to the head of the list. |
| +addLast(e: E): void | Adds the object to the tail of the list. |
| +getFirst(): E | Returns the first object in the list. |
| +getLast(): E | Returns the last object in the list. |
| +removeFirst(): E | Removes the first object from the list. |
| +removeLast(): E | Removes the last object from the list. |

MyLinkedList

# Implementing addFirst(E o)

```
public void addFirst(E o) {
  Node<E> newNode = new Node<E>(o);
  newNode.next = head;
  head = newNode;
  size++;
  if (tail == null)
    tail = head;
}
```

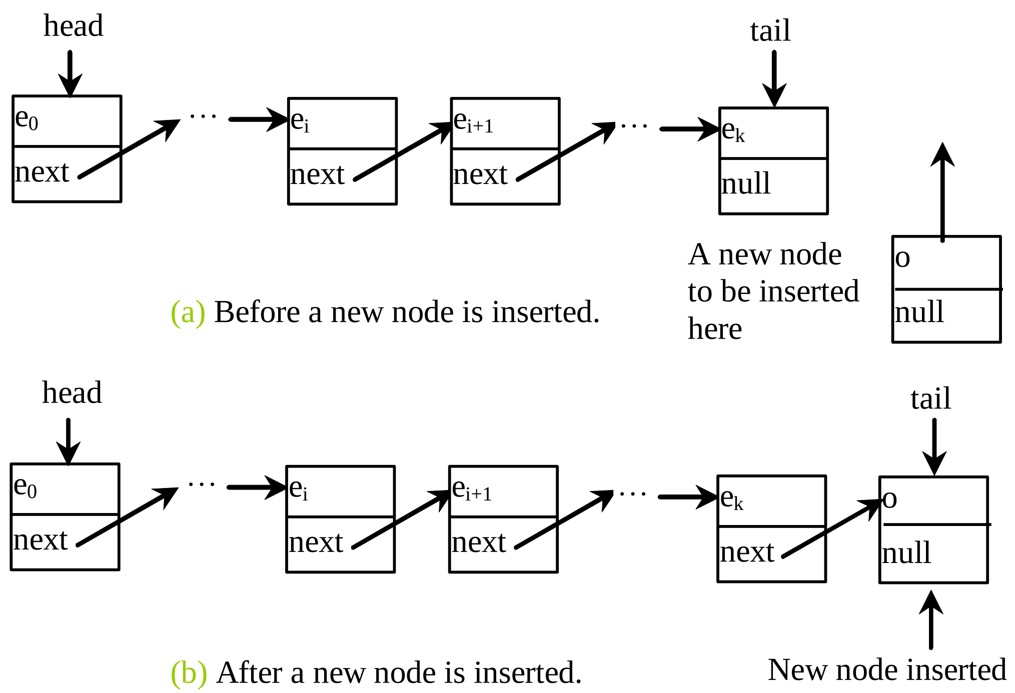


(a) Before a new node is inserted.

(b) After a new node is inserted.

# Implementing addLast(E o)

```
public void addLast(E o) {
  if (tail == null) {
    head = tail = new Node<E>(element);
  }
  else {
    tail.next = new Node(element);
    tail = tail.next;
  }
  size++;
}
```

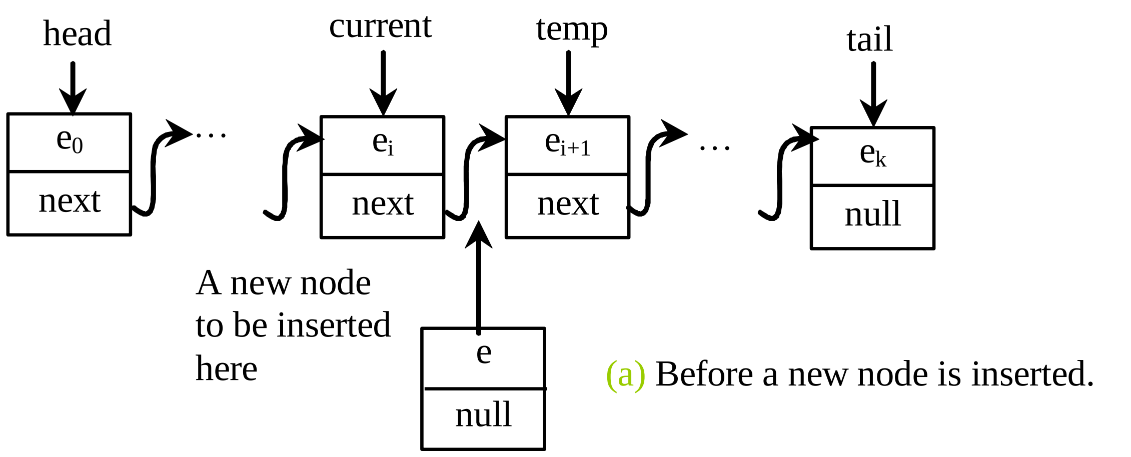


(a) Before a new node is inserted.

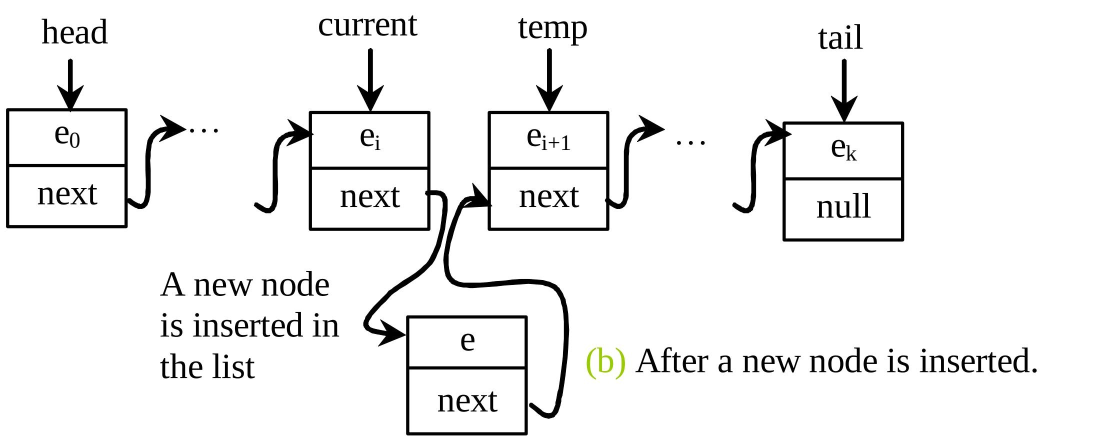
(b) After a new node is inserted.

# Implementing add(int index, E o)

```
public void add(int index, E o) {
  if (index == 0) addFirst(o);
  else if (index >= size) addLast(o);
  else {
    Node<E> current = head;
    for (int i = 1; i < index; i++)
      current = current.next;
    Node<E> temp = current.next;
    current.next = new Node<E>(o
    (current.next).next = temp;
    size++;
  }
}
```



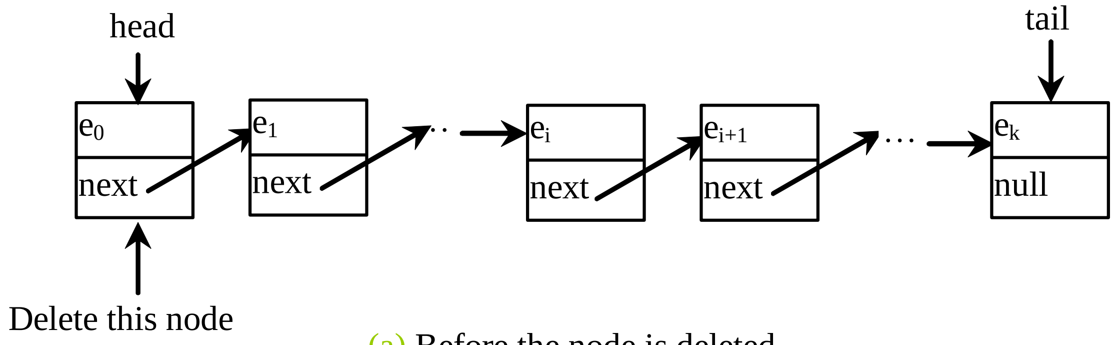
(a) Before a new node is inserted.



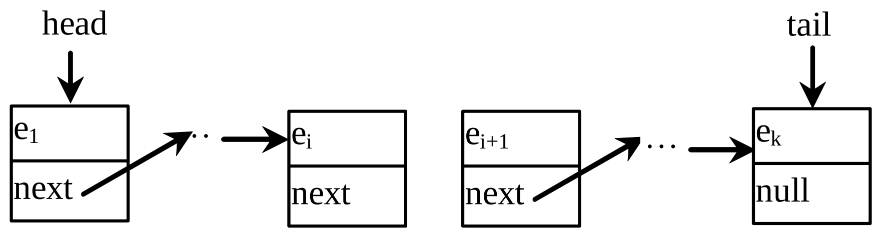
(b) After a new node is inserted.

# Implementing removeFirst()

```
public E removeFirst() {
  if (size == 0) return null;
  else {
    Node<E> temp = head;
    head = head.next;
    size--;
    if (head == null) tail = null;
    return temp.element;
  }
}
```



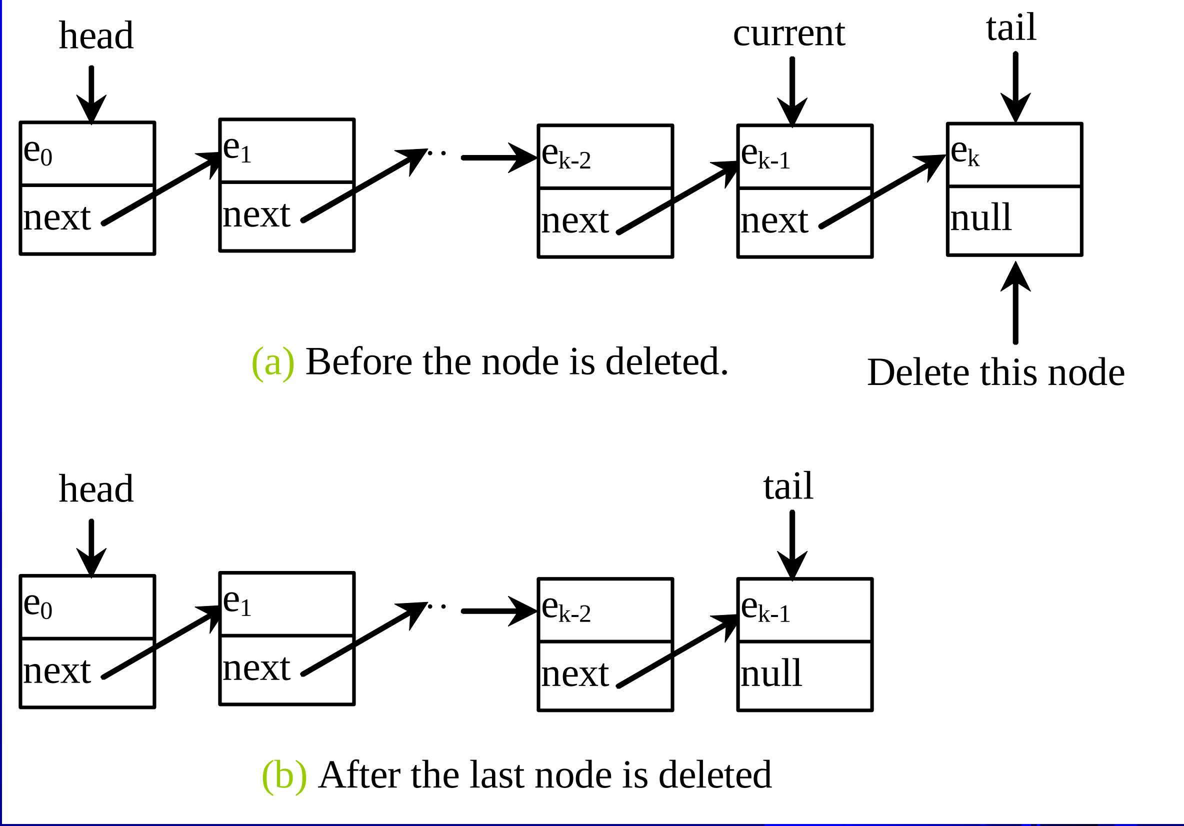
(a) Before the node is deleted.



(b) After the first node is deleted

```
public E removeLast() {
 if (size == 0) return null;
 else if (size == 1)
 {
  Node<E> temp = head;
  head = tail = null;
  size = 0;
  return temp.element;
 }
 else
 {
  Node<E> current = head;
  for (int i = 0; i < size - 2; i++)
   current = current.next;
  Node temp = tail;
  tail = current;
  tail.next = null;
  size--;
  return temp.element;
 }
}
```

# Implementing removeLast()

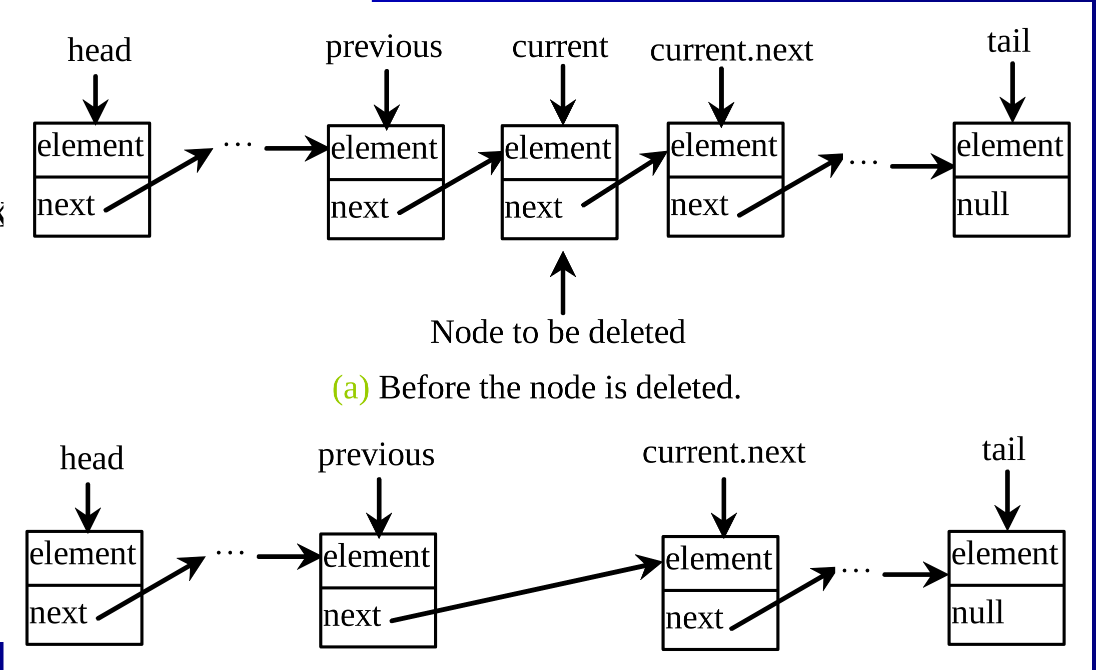


(a) Before the node is deleted.

(b) After the last node is deleted

# Implementing remove(int index)

```
public E remove(int index) {
  if (index < 0 || index >= size) return null;
  else if (index == 0) return removeFirst();
  else if (index == size - 1) return removeLast();
  else {
    Node<E> previous = head;
    for (int i = 1; i < index; i++) {
      previous = previous.next;
    }
    Node<E> current = previous.nex
    previous.next = current.next;
    size--;
    return current.element;
  }
}
```

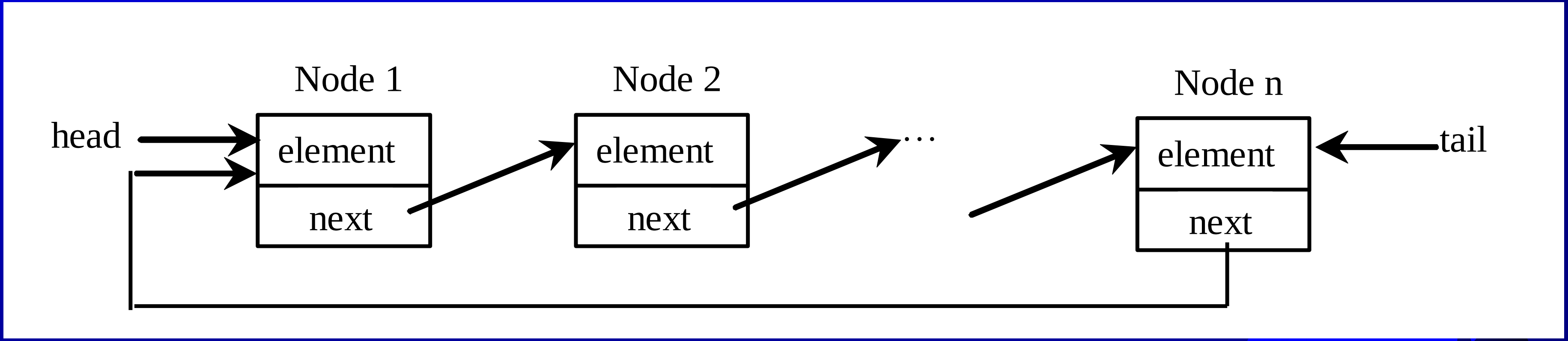


(a) Before the node is deleted.

(b) After the node is deleted.

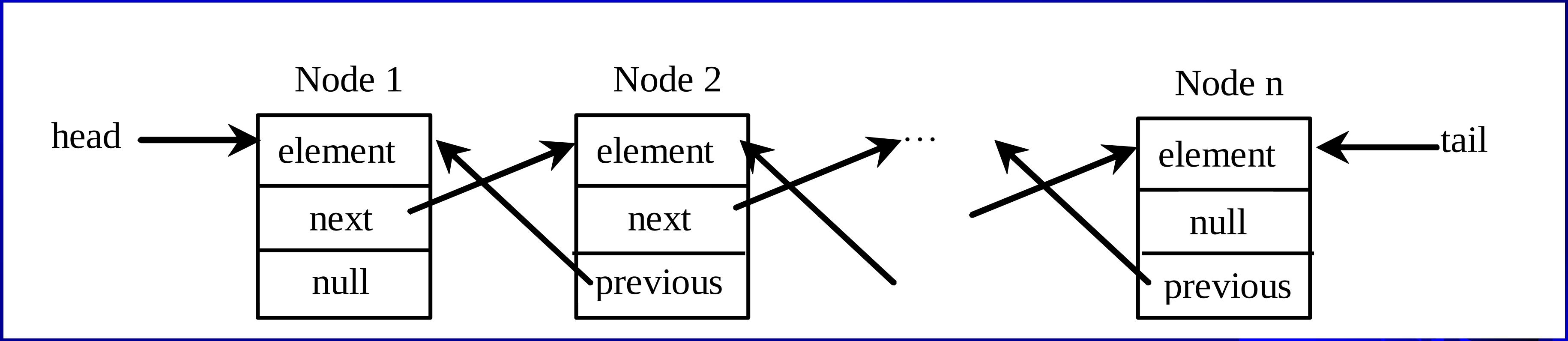
# Circular Linked Lists

- A *circular, singly linked list* is like a singly linked list, except that the pointer of the last node points back to the first node.



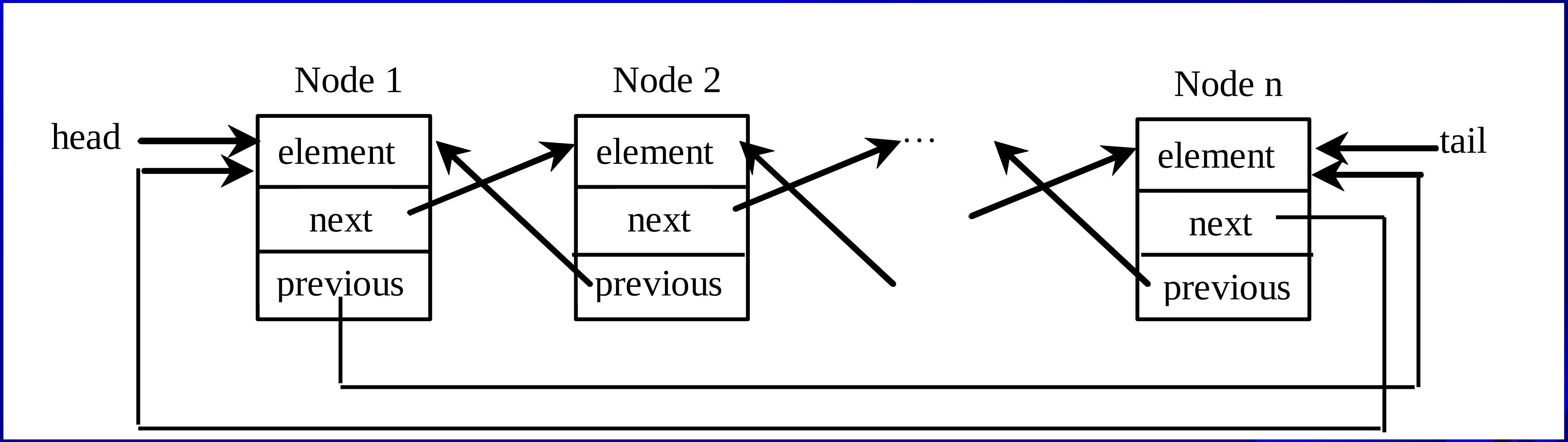
# Doubly Linked Lists

- A *doubly linked list* contains the nodes with two pointers. One points to the next node and the other points to the previous node. These two pointers are conveniently called *a forward pointer* and *a backward pointer*. So, a doubly linked list can be traversed forward and backward.



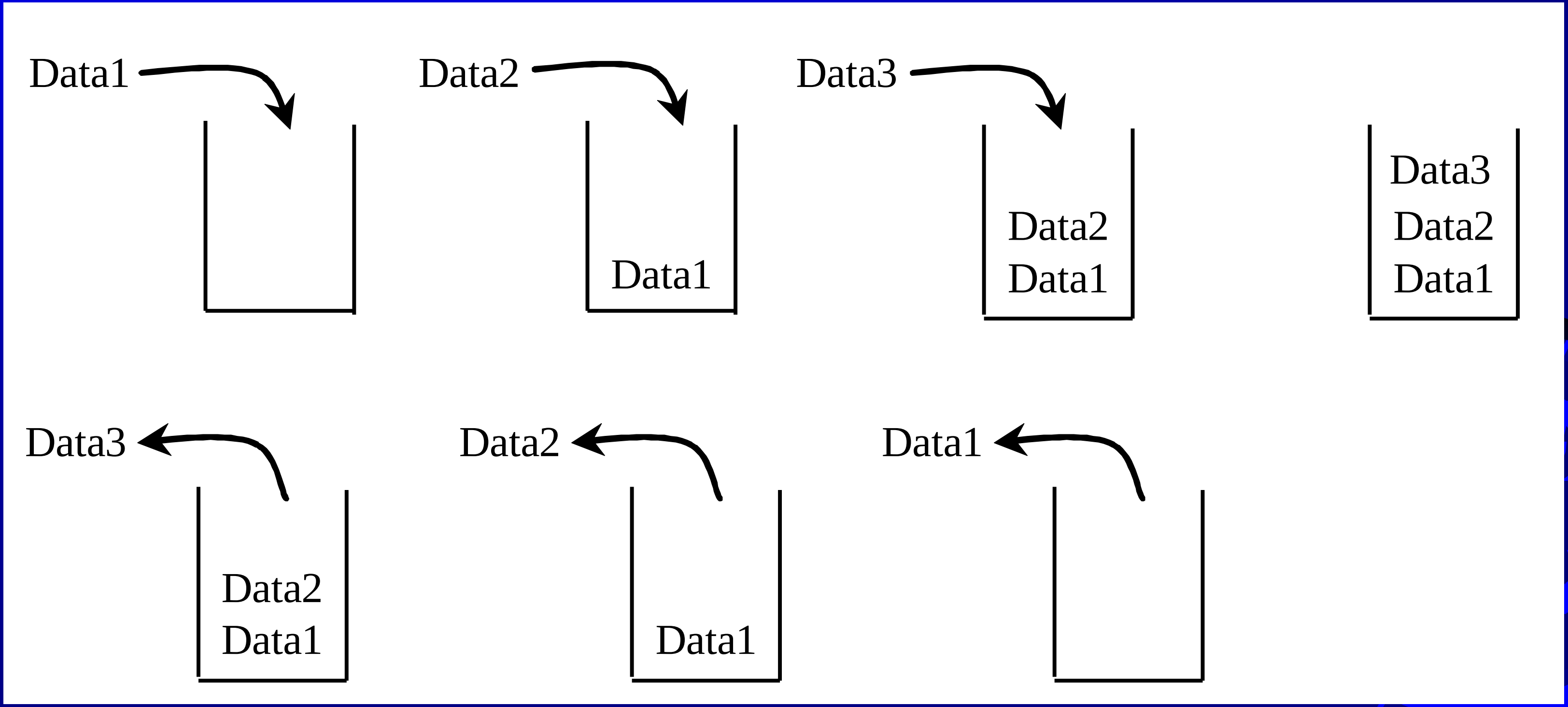
# Circular Doubly Linked Lists

- A *circular, doubly linked list* is doubly linked list, except that the forward pointer of the last node points to the first node and the backward pointer of the first pointer points to the last node.



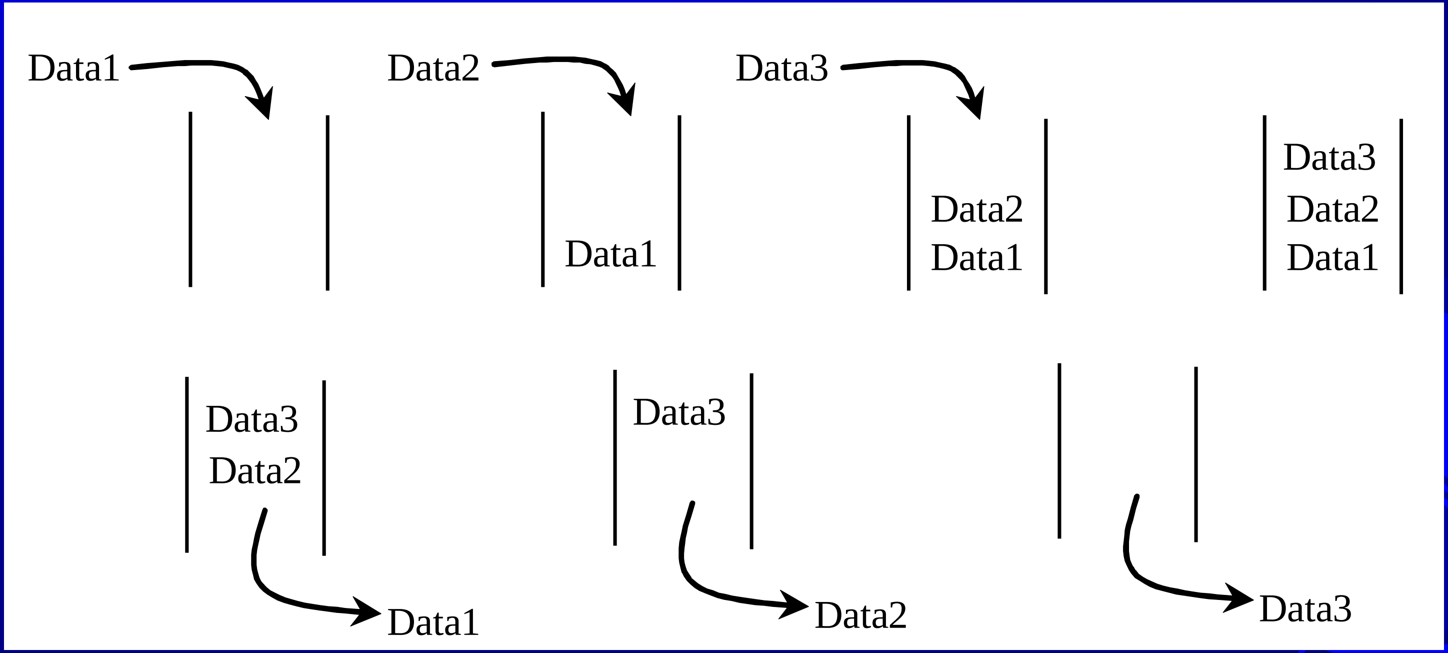
# Stacks

A stack can be viewed as a special type of list, where the elements are accessed, inserted, and deleted only from the end, called the top, of the stack.



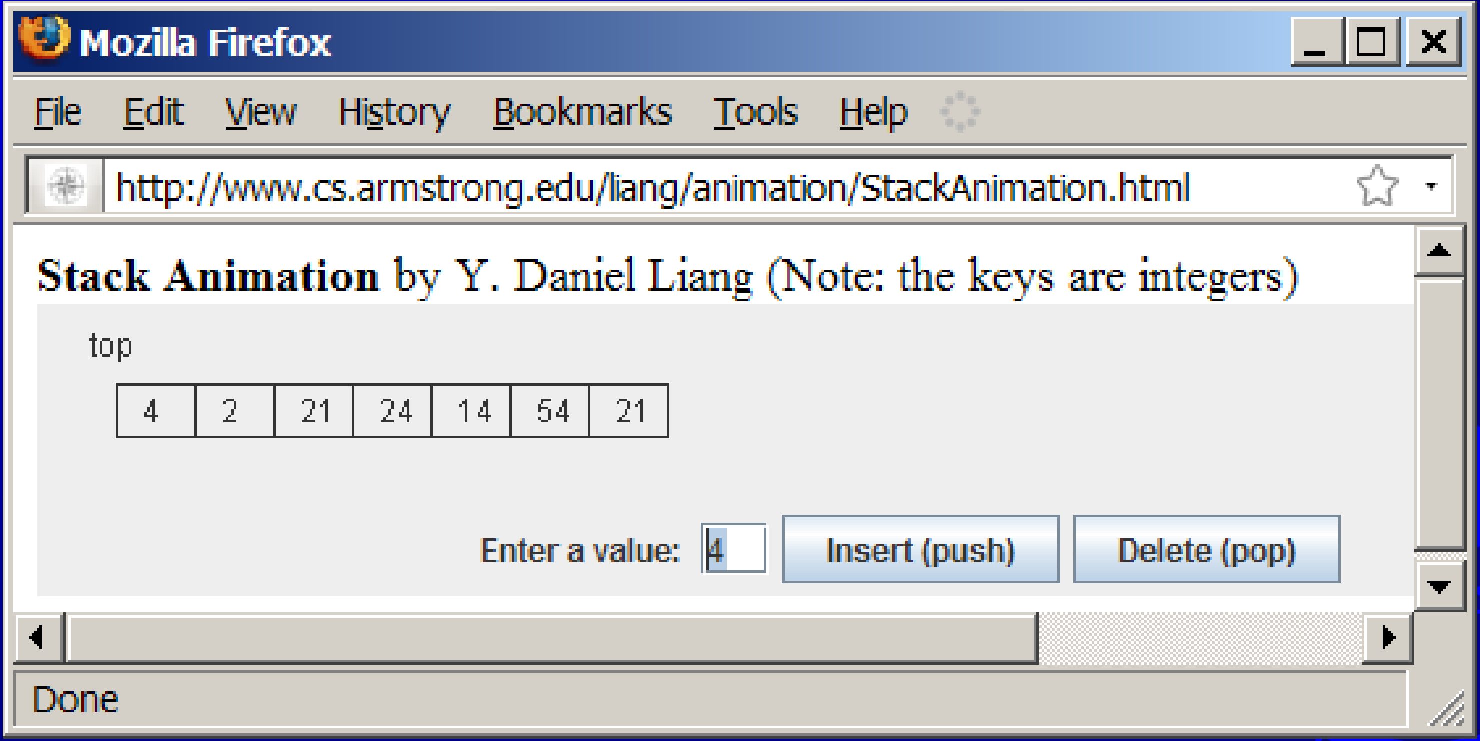
# Queues

A queue represents a waiting list. A queue can be viewed as a special type of list, where the elements are inserted into the end (tail) of the queue, and are accessed and deleted from the beginning (head) of the queue.



## Stack Animation

[www.cs.armstrong.edu/liang/animation/StackAnimation.html](www.cs.armstrong.edu/liang/animation/StackAnimation.html)

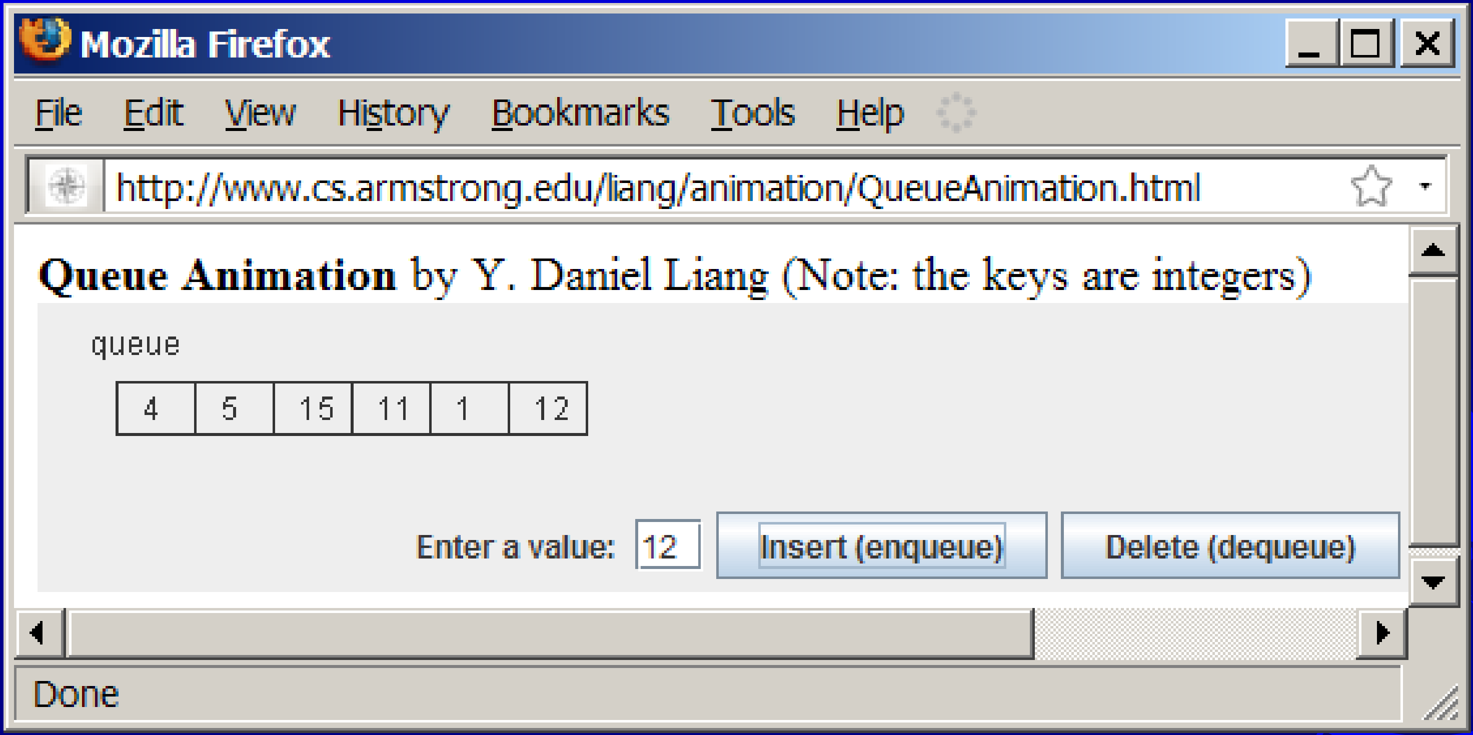

Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.cs.armstrong.edu/liang/animation/StackAnimation.html
top
Enter a value: 4
Insert (push)
Delete (pop)
Done

**Stack Animation** by Y. Daniel Liang (Note: the keys are integers)

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| 4 | 2 | 21 | 24 | 14 | 54 | 21 |

# Queue Animation

www.cs.armstrong.edu/liang/animation/QueueAnimation.html


Mozilla Firefox

http://www.cs.armstrong.edu/liang/animation/QueueAnimation.html

**Queue Animation** by Y. Daniel Liang (Note: the keys are integers)

queue

| | | | | | |
|---|---|---|---|---|---|
| 4 | 5 | 15 | 11 | 1 | 12 |

Enter a value: 12

Done

## Implementing Stacks and Queues

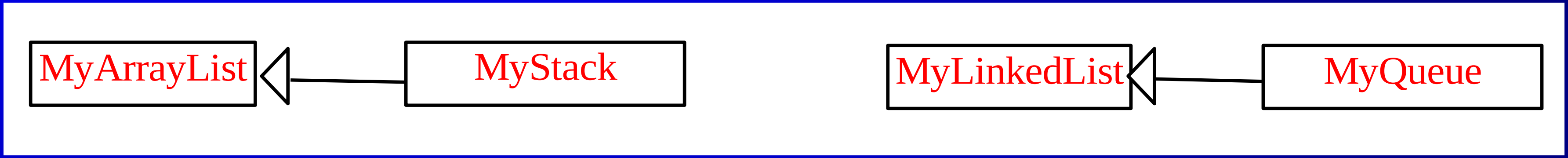
- Using an array list to implement Stack
- Use a linked list to implement Queue

Since the insertion and deletion operations on a stack are made only at the end of the stack, using an array list to implement a stack is more efficient than a linked list. Since deletions are made at the beginning of the list, it is more efficient to implement a queue using a linked list than an array list. This section implements a stack class using an array list and a queue using a linked list.

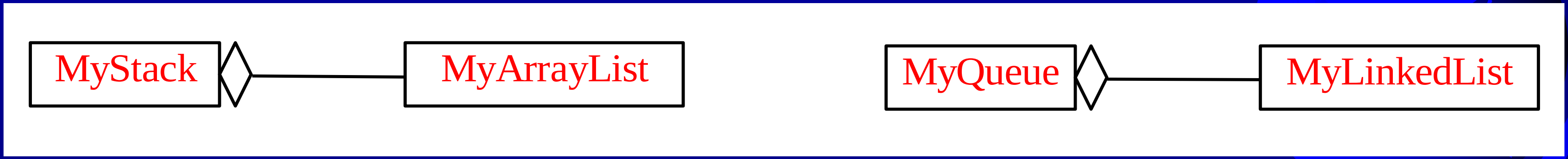
# Design of the Stack and Queue Classes

There are two ways to design the stack and queue classes:

– Using inheritance: You can declare the stack class by extending the array list class, and the queue class by extending the linked list class.



– Using composition: You can declare an array list as a data field in the stack class, and a linked list as a data field in the queue class.



## Composition is Better

Both designs are fine, but using composition is better because it enables you to define a complete new stack class and queue class without inheriting the unnecessary and inappropriate methods from the array list and linked list.

# MyStack and MyQueue

MyStack

| MyStack | |
|---|---|
| -list: MyArrayList | |
| +isEmpty(): boolean | Returns true if this stack is empty. |
| +getSize(): int | Returns the number of elements in this stack. |
| +peek(): Object | Returns the top element in this stack. |
| +pop(): Object | Returns and removes the top element in this stack. |
| +push(o: Object): Object | Adds a new element to the top of this stack. |
| +search(o: Object): int | Returns the position of the specified element in this stack. |

MyQueue

| MyQueue<E> | |
|---|---|
| -list: MyLinkedList<E> | |
| +enqueue(e: E): void | Adds an element to this queue. |
| +dequeue(): E | Removes an element from this queue. |
| +getSize(): int | Returns the number of elements from this queue. |

## Example: Using Stacks and Queues

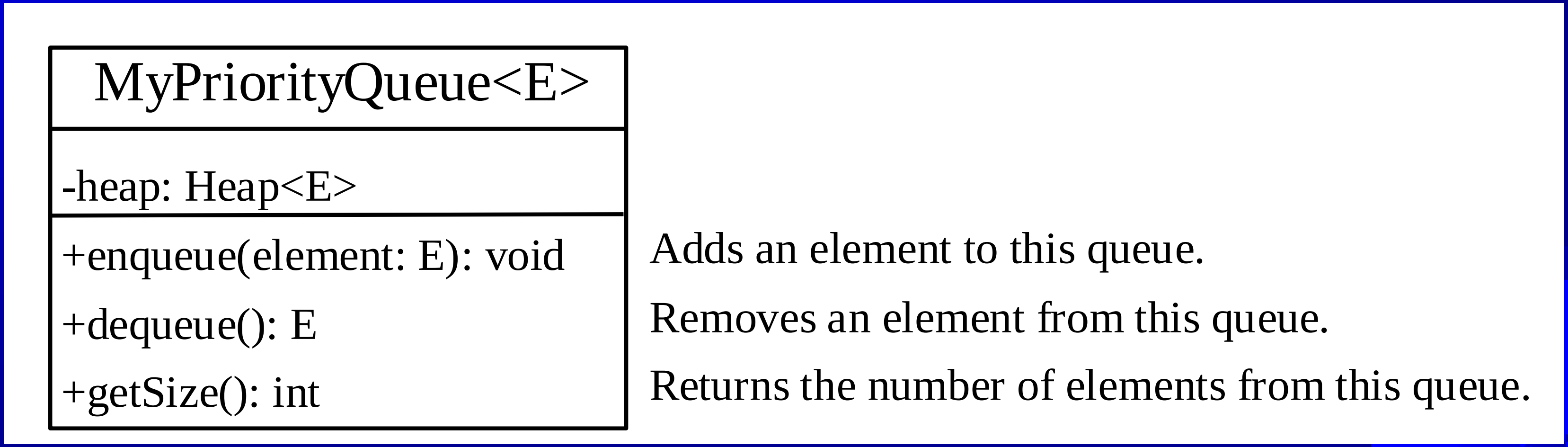
Write a program that creates a stack using MyStack and a queue using MyQueue. It then uses the push (enqueu) method to add strings to the stack (queue) and the pop (dequeue) method to remove strings from the stack (queue).

TestStackQueue

Run

## Priority Queue

A regular queue is a first-in and first-out data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue. In a priority queue, elements are assigned with priorities. When accessing elements, the element with the highest priority is removed first. A priority queue has a largest-in, first-out behavior. For example, the emergency room in a hospital assigns patients with priority numbers; the patient with the highest priority is treated first.


MyPriorityQueue<E>
-heap: Heap<E>
+enqueue(element: E): void
+dequeue(): E
+getSize(): int
Adds an element to this queue.
Removes an element from this queue.
Returns the number of elements from this queue.

MyPriorityQueue TestPriorityQueue Run