Chapter 24 Implementing Lists, Stacks, Queues, and Priority Queues

Objectives

- To design common features of lists in an interface and provide skeleton implementation in an abstract class (§24.2).
- To design and implement a dynamic list using an array (§24.3).
- To design and implement a dynamic list using a linked structure (§24.4).
- To design and implement a stack class using an array list and a queue class using a linked list (§24.5).
- To design and implement a priority queue using a heap (§24.6).

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.

Two Ways to Implement Lists

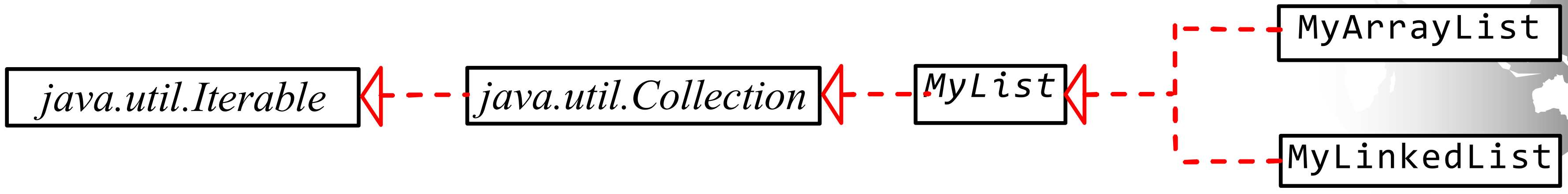
There are two ways to implement a list.

Using arrays. One is to use an array to store the elements. The array is dynamically created. If the capacity of the array is exceeded, create a new larger array and copy all the elements from the current array to the new array.

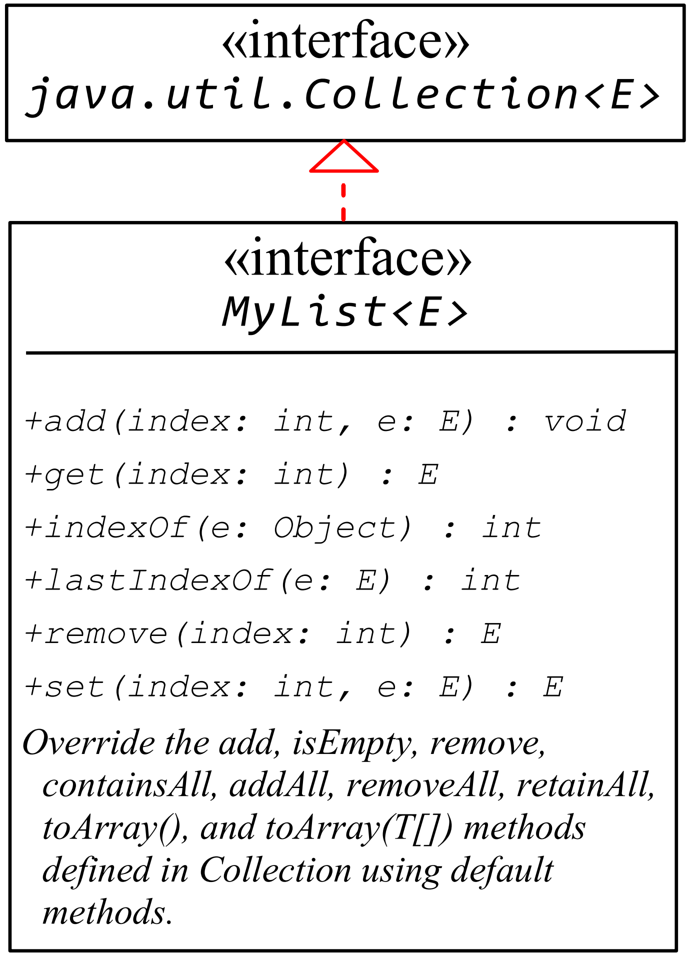
Using linked list. The other approach is to use a linked structure. A linked structure consists of nodes. Each node is dynamically created to hold an element. All the nodes are linked together to form a list.

Design of ArrayList and LinkedList

For convenience, let's name these two classes: MyArrayList and MyLinkedList. These two classes have common operations, but different data fields. The common operations can be generalized in an interface or an abstract class. Prior to Java 8, a popular design strategy is to define common operations in an interface and provide an abstract class for partially implementing the interface. So, the concrete class can simply extend the abstract class without implementing the full interface. Java 8 enables you to define default methods. You can provide default implementation for some of the methods in the interface rather than in an abstract class.



MyList Interface


«interface»
java.util.Collection<E>
«interface»
MyList<E>
+add(index: int, e: E) : void
+get(index: int) : E
+indexOf(e: Object) : int
+lastIndexOf(e: E) : int
+remove(index: int) : E
+set(index: int, e: E) : E
Override the add, isEmpty, remove, containsAll, addAll, removeAll, retainAll, toArray(), and toArray(T[]) methods defined in Collection using default methods.

Inserts a new element at the specified index in this list.

Returns the element from this list at the specified index.

Returns the index of the first matching element in this list.

Returns the index of the last matching element in this list.

Removes the element at the specified index and returns the removed element.

Sets the element at the specified index and returns the element being replaced.

MyList

Array Lists

Array is a fixed-size data structure. Once an array is created, its size cannot be changed. Nevertheless, you can still use array to implement dynamic data structures. The trick is to create a new larger array to replace the current array if the current array cannot hold new elements in the list.

Initially, an array, say data of Object[] type, is created with a default size. When inserting a new element into the array, first ensure there is enough room in the array. If not, create a new array with the size as twice as the current one. Copy the elements from the current array to the new array. The new array now becomes the current array.

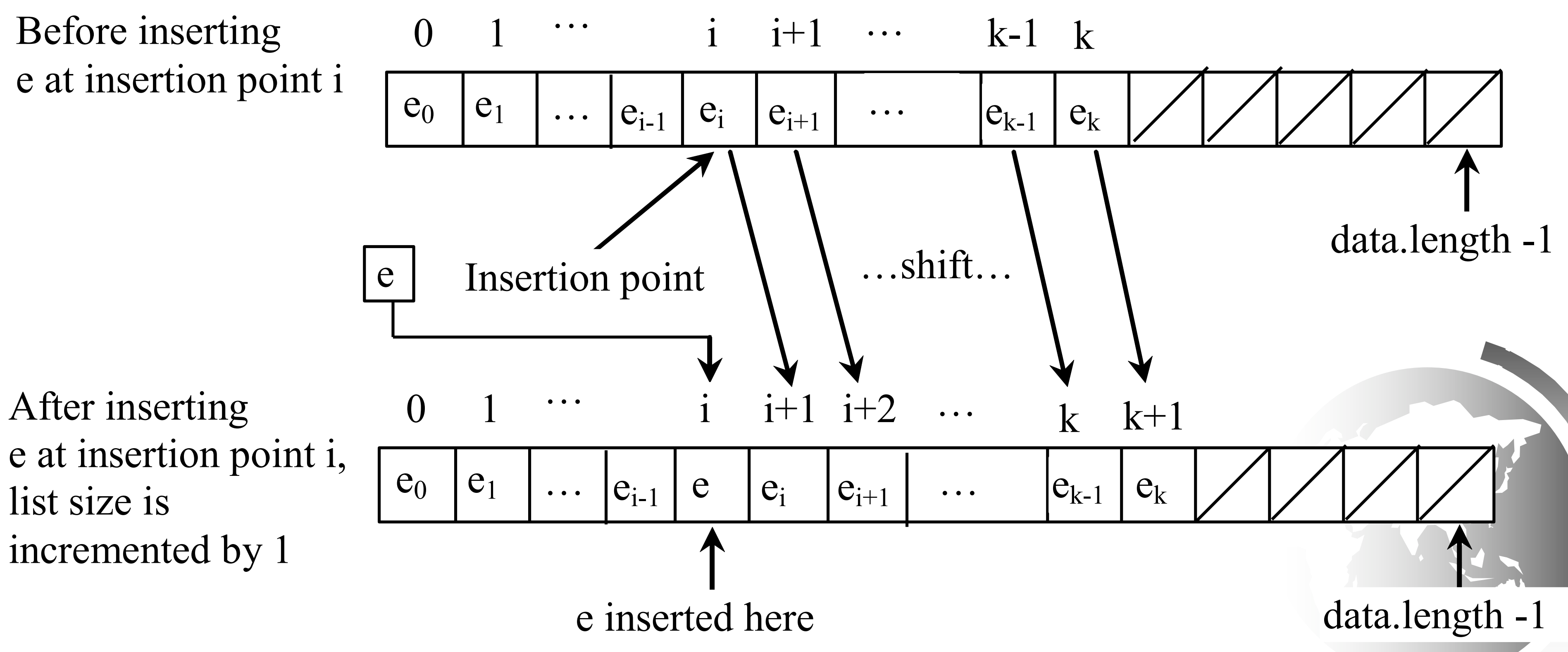
Array List Animation

https://liveexample.pearsoncmg.com/dsanimation/ArrayListeBook.html

Insertion

Before inserting a new element at a specified index, shift all the elements after the index to the right and increase the list size by 1.

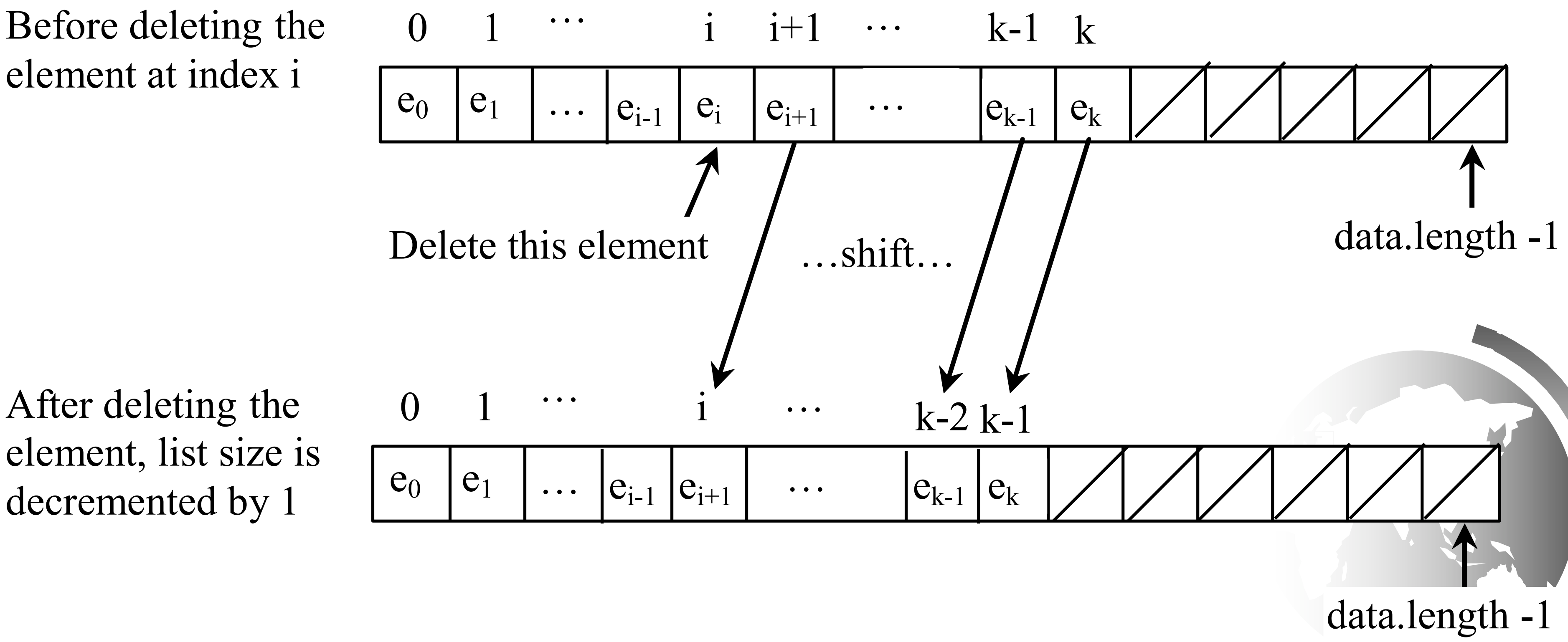
Add Method Animation



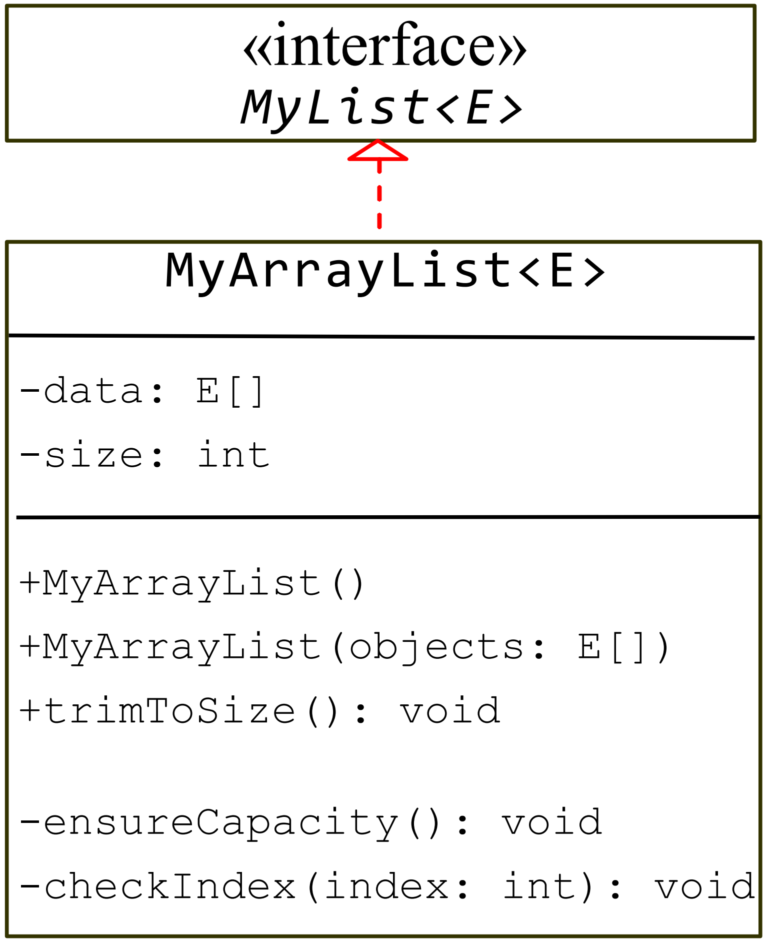
Deletion

To remove an element at a specified index, shift all the elements after the index to the left by one position and decrease the list size by 1.

Remove Method Animation



Implementing MyArrayList



```
-data: E[]
-size: int

+MyArrayList()
+MyArrayList(objects: E[])
+trimToSize(): void

-ensureCapacity(): void
-checkIndex(index: int): void
```

Array for storing elements in this array list.

The number of elements in the array list.

Creates a default array list.

Creates an array list from an array of objects.

Trims the capacity of this array list to the list's current size.

Doubles the current array size if needed.

Throws an exception if the index is out of bounds in the list.

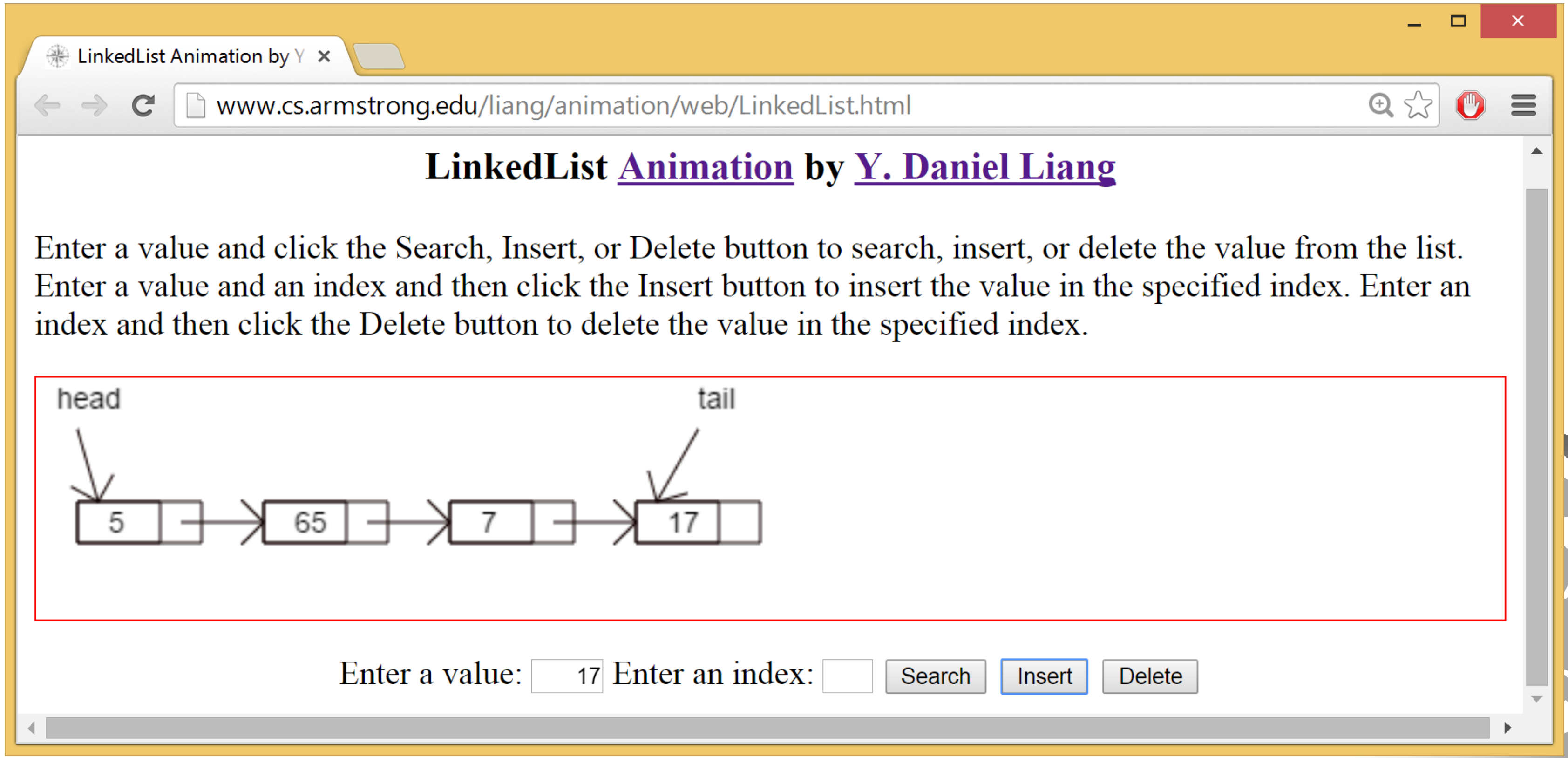
MyArrayList TestMyArrayList

Linked Lists

Since MyArrayList is implemented using an array, the methods get(int index) and set(int index, Object o) for accessing and modifying an element through an index and the add(Object o) for adding an element at the end of the list are efficient. However, the methods add(int index, Object o) and remove(int index) are inefficient because it requires shifting potentially a large number of elements. You can use a linked structure to implement a list to improve efficiency for adding and removing an element anywhere in a list.

Linked List Animation

https://liveexample.pearsoncmg.com/dsanimation/LinkedListeBook.html



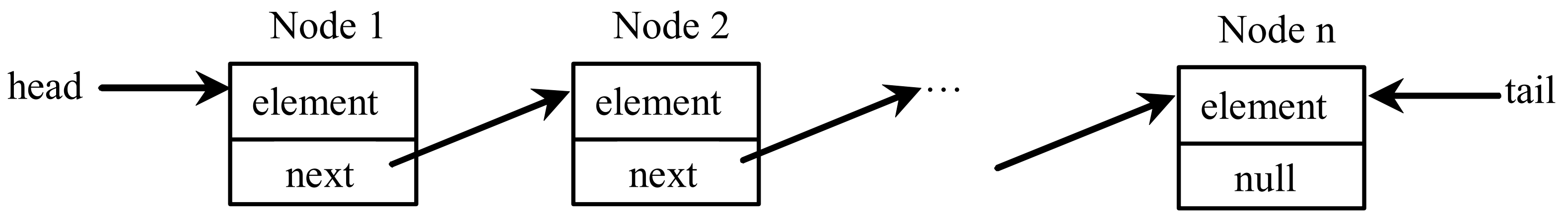
LinkedList Animation by Y. Daniel Liang

Enter a value and click the Search, Insert, or Delete button to search, insert, or delete the value from the list. Enter a value and an index and then click the Insert button to insert the value in the specified index. Enter an index and then click the Delete button to delete the value in the specified index.

LinkedList Animation by Y
www.cs.armstrong.edu/liang/animation/web/LinkedList.html
head
tail
5
65
7
17
Enter a value: 17
Enter an index:
Search
Insert
Delete

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<E> {
  E element;
  Node<E> next;

  public Node(E 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;
```

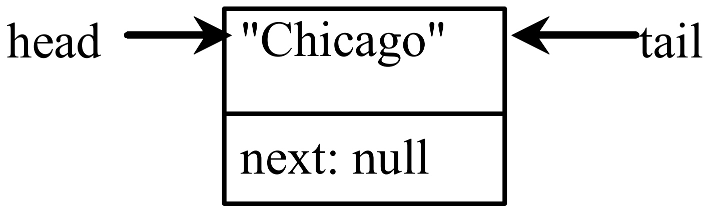
The list is empty now

Adding Three Nodes, cont.

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

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

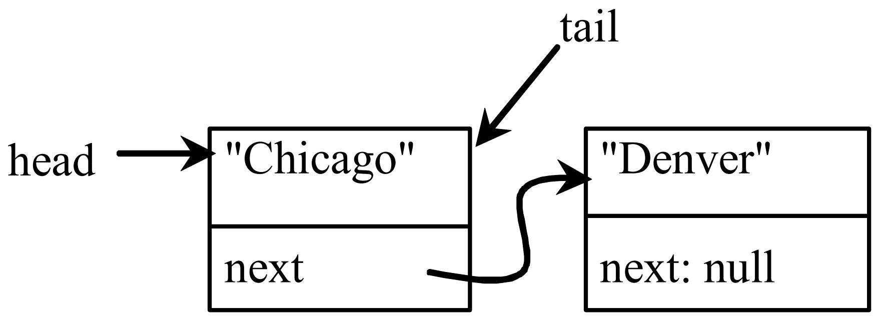
After the first node is inserted



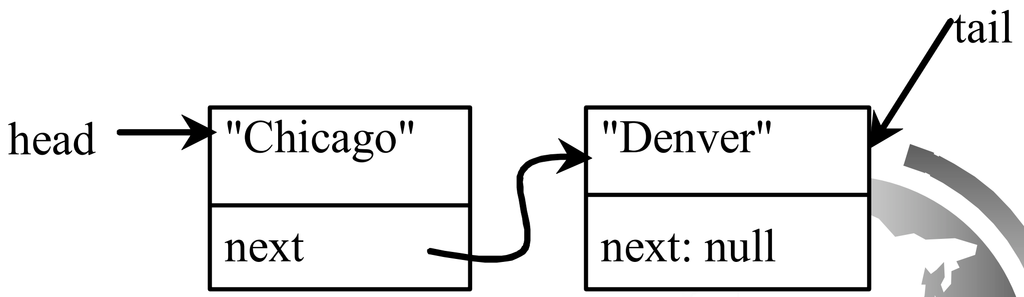
Adding Three Nodes, cont.

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

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



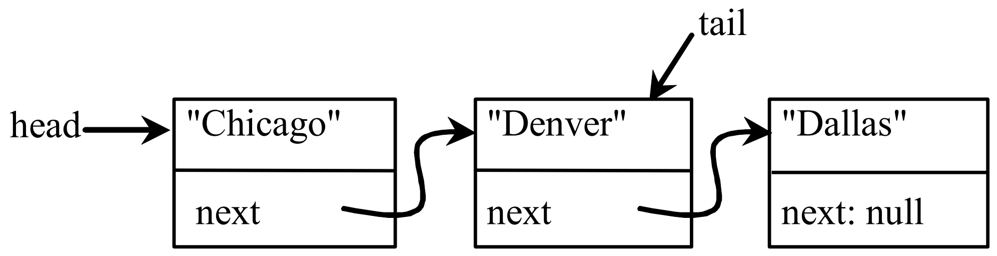
```
tail  = tail.next;
```



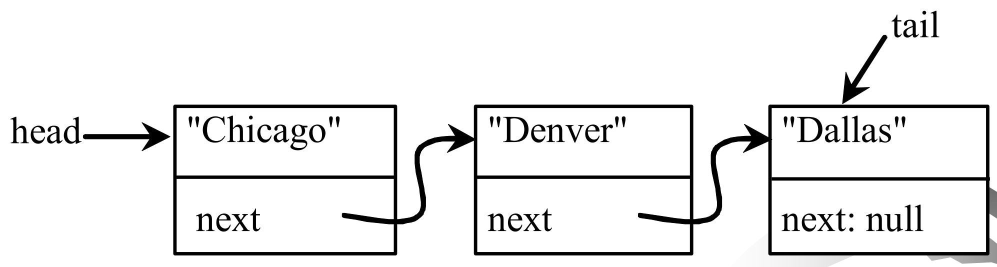
Adding Three Nodes, cont.

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

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



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

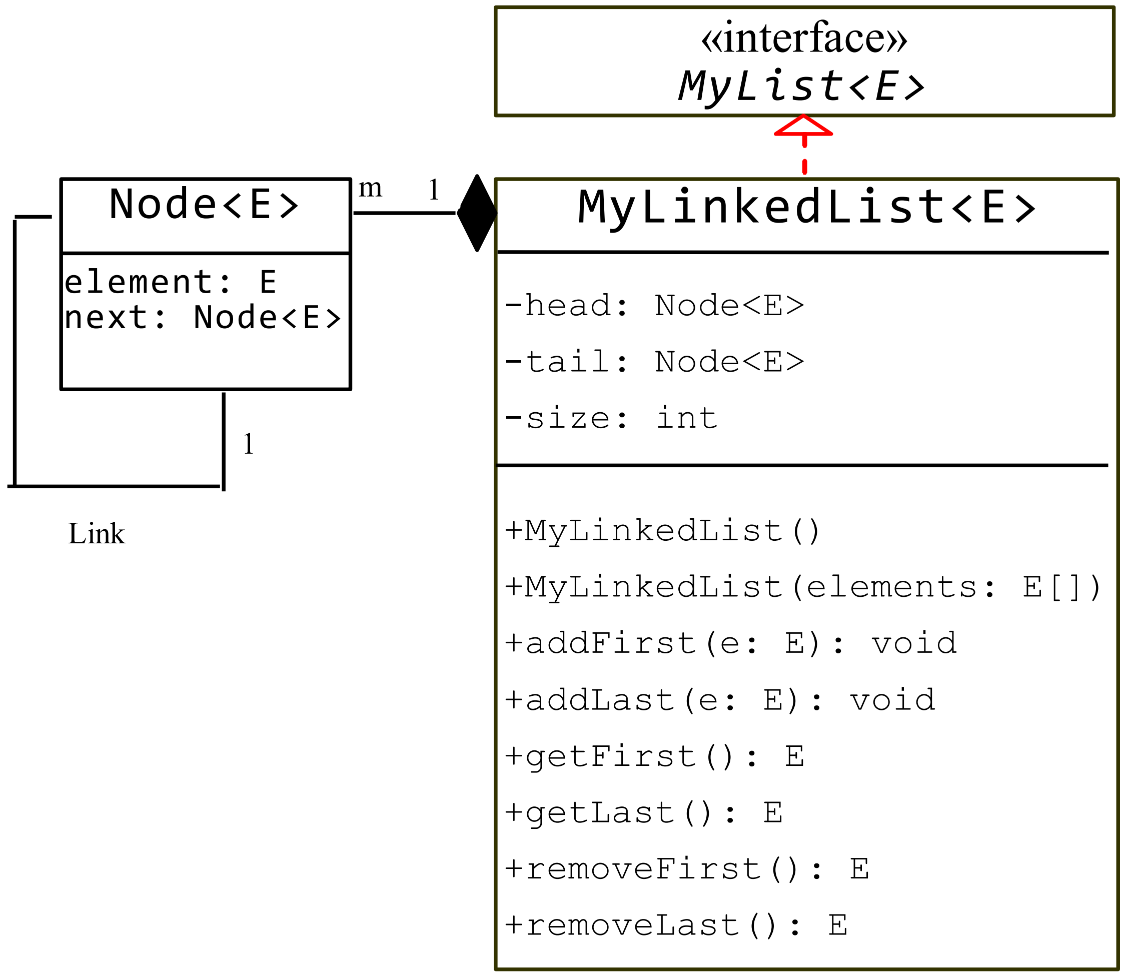


Traversing All Elements in the List

Each node contains the element and a data field named *next* that points to the next node. 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



The head of the list.

The tail of the list.

The number of elements in the list.

Creates a default linked list.

Creates a linked list from an array of elements.

Adds an element to the head of the list.

Adds an element to the tail of the list.

Returns the first element in the list.

Returns the last element in the list.

Removes the first element from the list.

Removes the last element from the list.

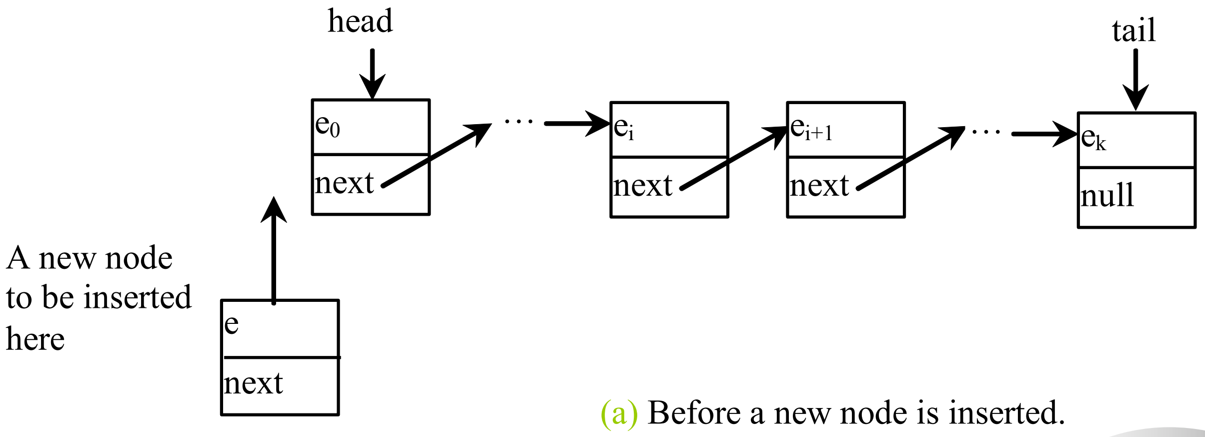
MyLinkedList

TestMyLinkedList

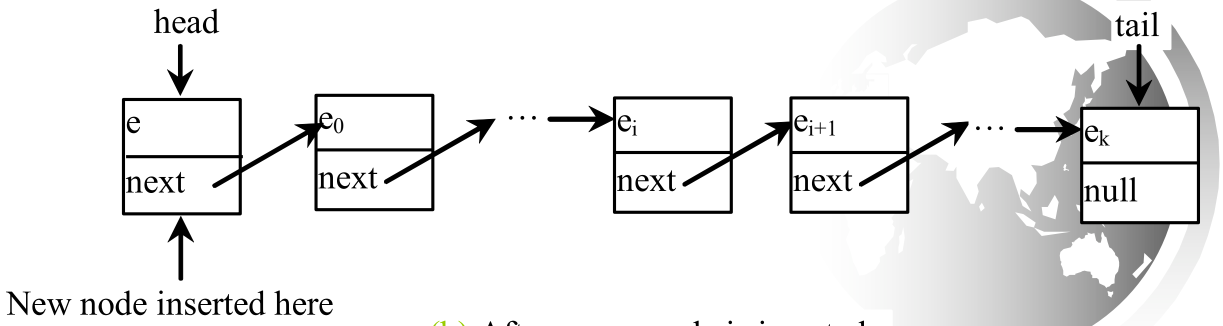
Implementing addFirst(E e)

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

addFirst Animation



(a) Before a new node is inserted.

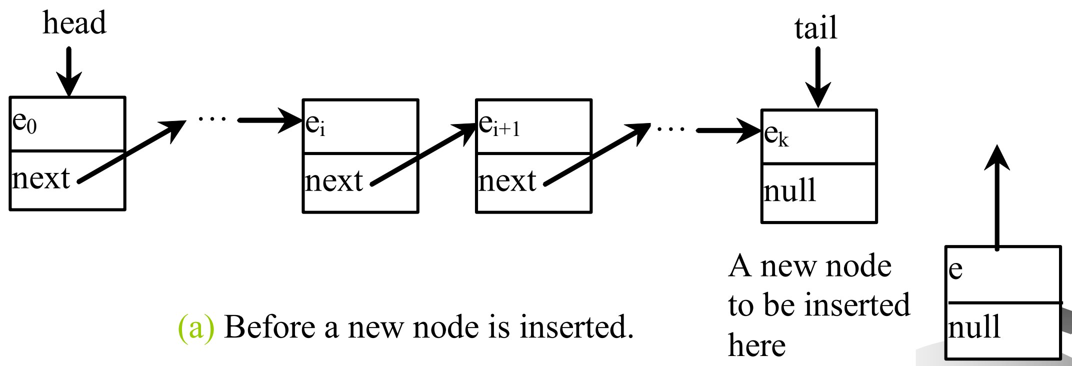


(b) After a new node is inserted.

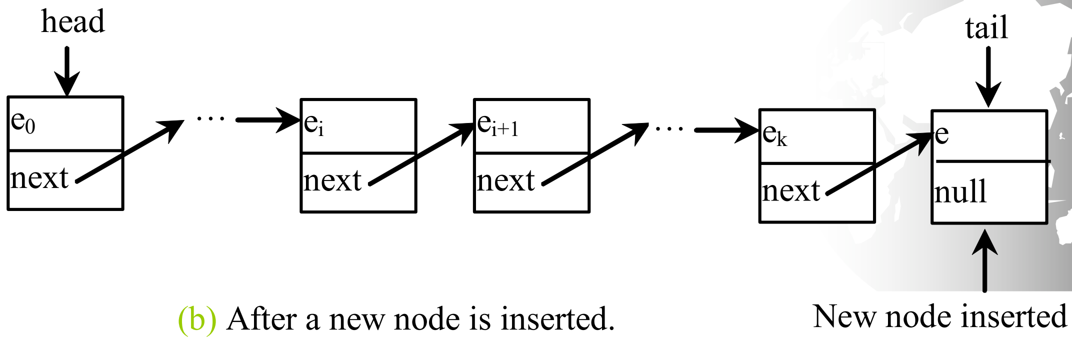
Implementing addLast(E e)

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

addLast Animation



(a) Before a new node is inserted.

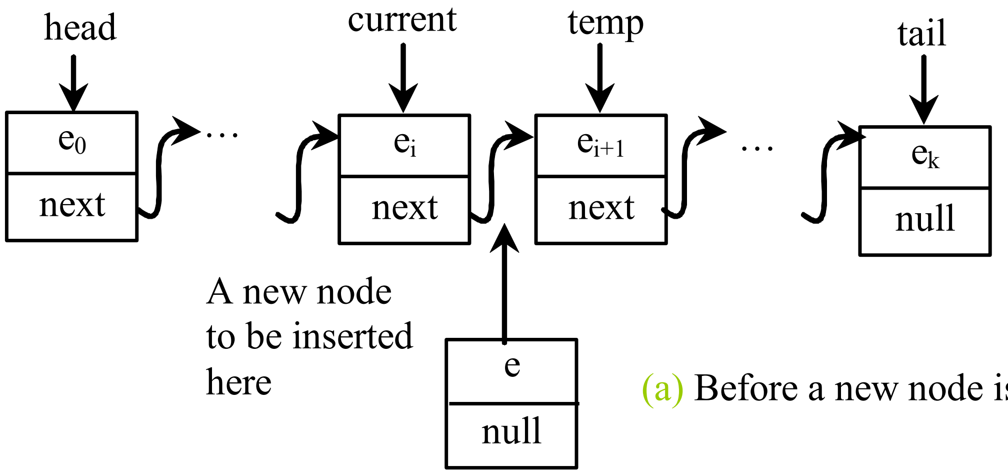


(b) After a new node is inserted.

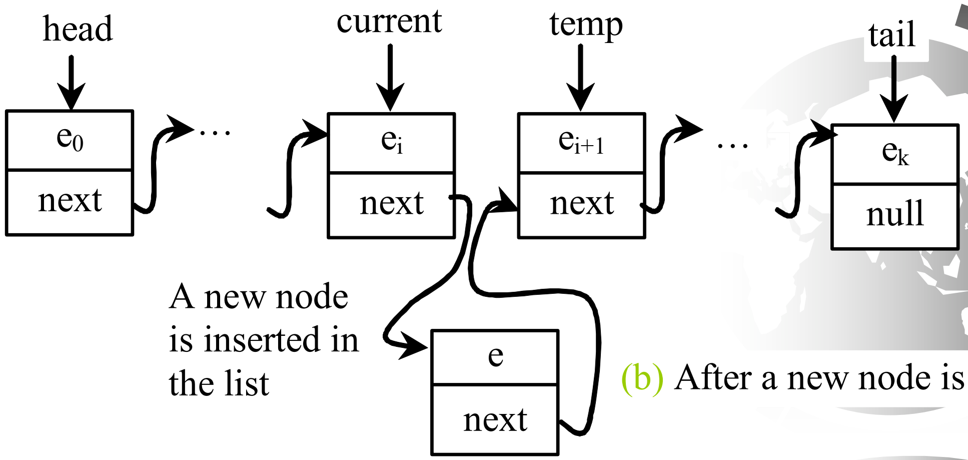
Implementing add(int index, E e)

```
public void add(int index, E e) {
  if (index == 0) addFirst(e);
  else if (index >= size) addLast(e);
  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);
    (current.next).next = temp;
    size++;
  }
}
```

Add(index, e) Animation



(a) Before a new node is inserted.

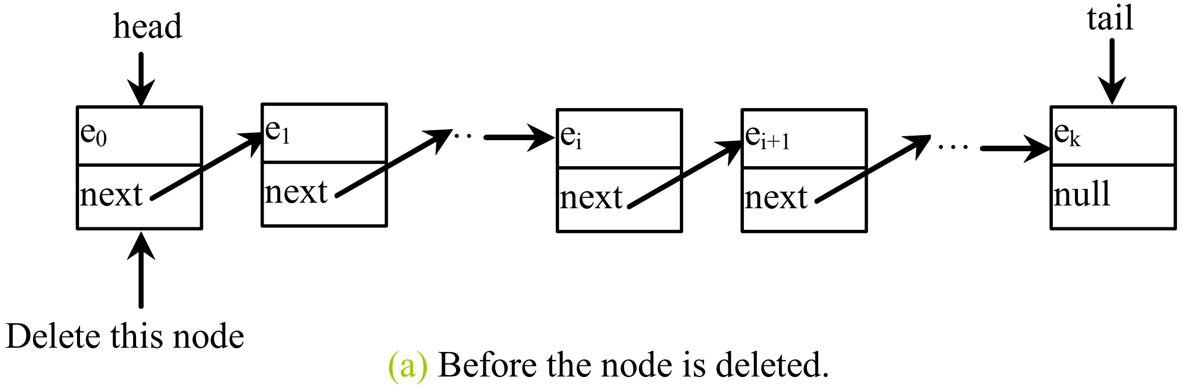


(b) After a new node is inserted.

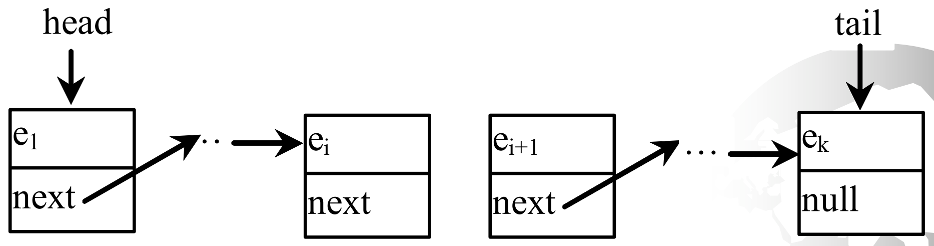
Implementing removeFirst()

```
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;
  }
}
```

removeFirst Animation



(a) Before the node is deleted.

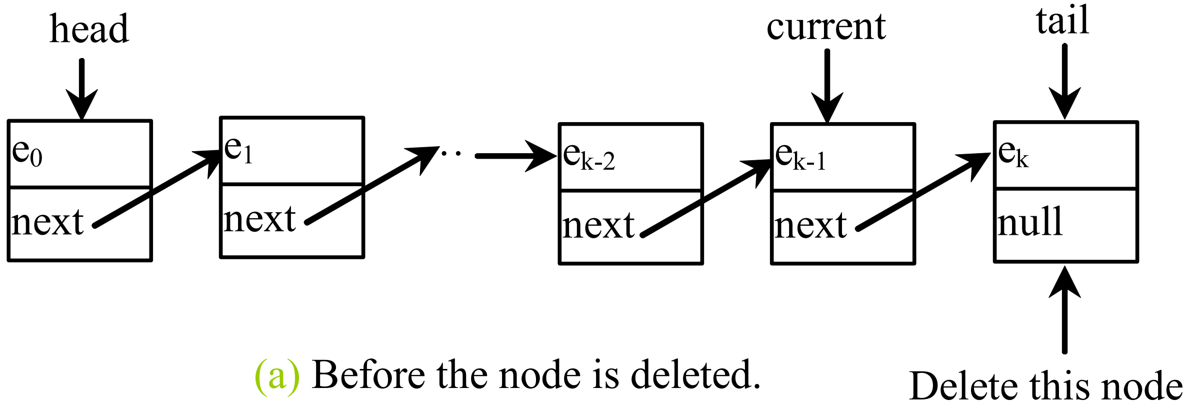


(b) After the first node is deleted

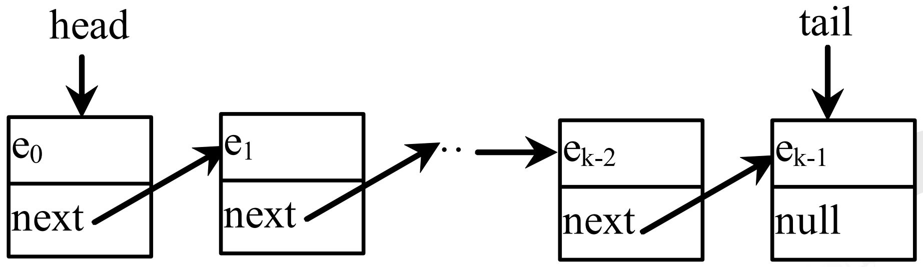
Implementing removeLast()

```
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;
  }
}
```

addLast Animation



(a) Before the node is deleted.

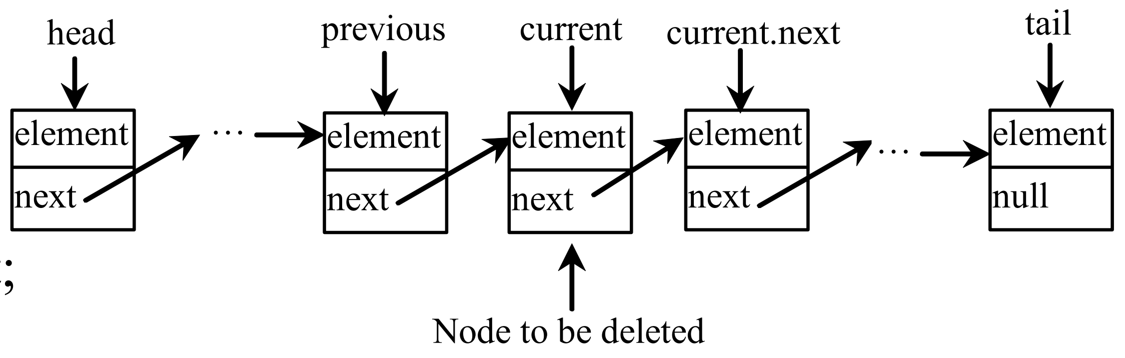


(b) After the last node is deleted

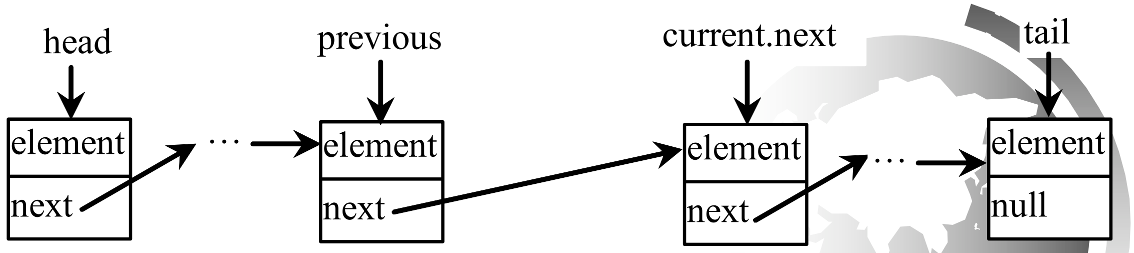
Implementing remove(int index)

```
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.next;
    previous.next = current.next;
    size--;
    return current.element;
  }
}
```

remove Animation



(a) Before the node is deleted.



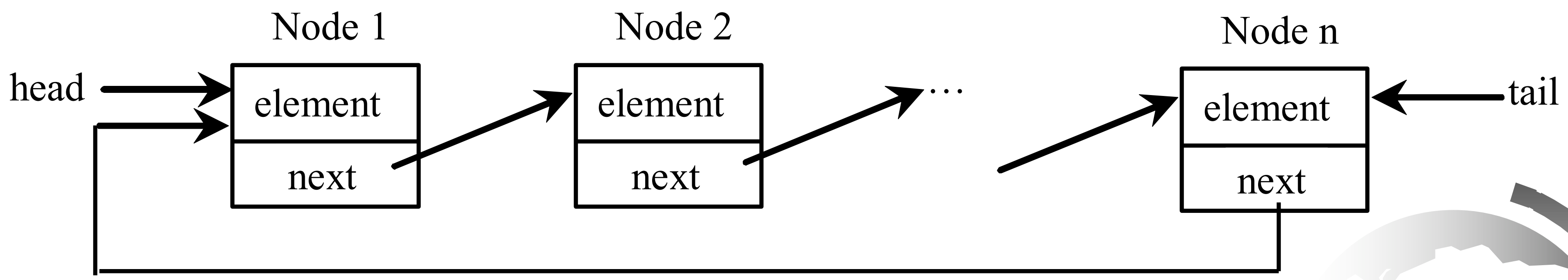
(b) After the node is deleted.

Time Complexity for ArrayList and LinkedList

Methods	MyArrayList/ArrayList	MyLinkedList/LinkedList
`add(e: E)`	$O(1)$	$O(1)$
`add(index: int, e: E)`	$O(n)$	$O(n)$
`clear()`	$O(1)$	$O(1)$
`contains(e: E)`	$O(n)$	$O(n)$
`get(index: int)`	$O(1)$	$O(n)$
`indexOf(e: E)`	$O(n)$	$O(n)$
`isEmpty()`	$O(1)$	$O(1)$
`lastIndexOf(e: E)`	$O(n)$	$O(n)$
`remove(e: E)`	$O(n)$	$O(n)$
`size()`	$O(1)$	$O(1)$
`remove(index: int)`	$O(n)$	$O(n)$
`set(index: int, e: E)`	$O(n)$	$O(n)$
`addFirst(e: E)`	$O(n)$	$O(1)$
`removeFirst()`	$O(n)$	$O(1)$

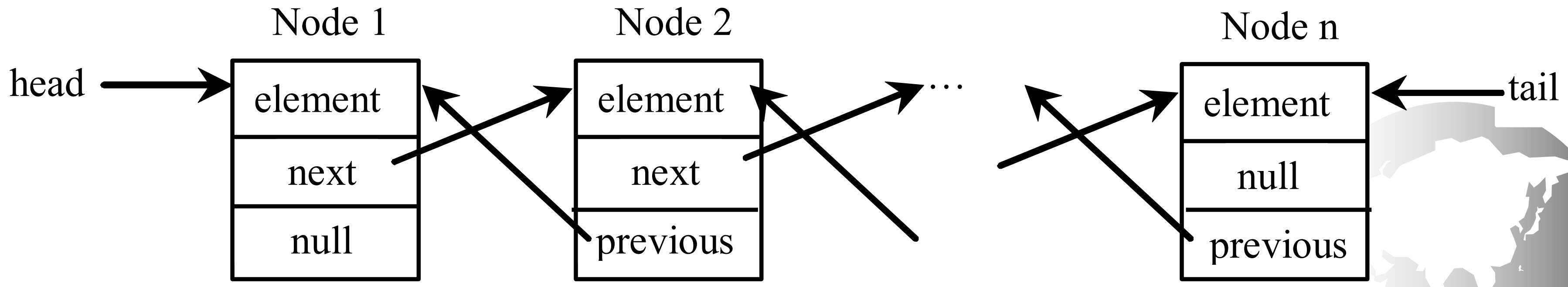
Circular Linked Lists

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.



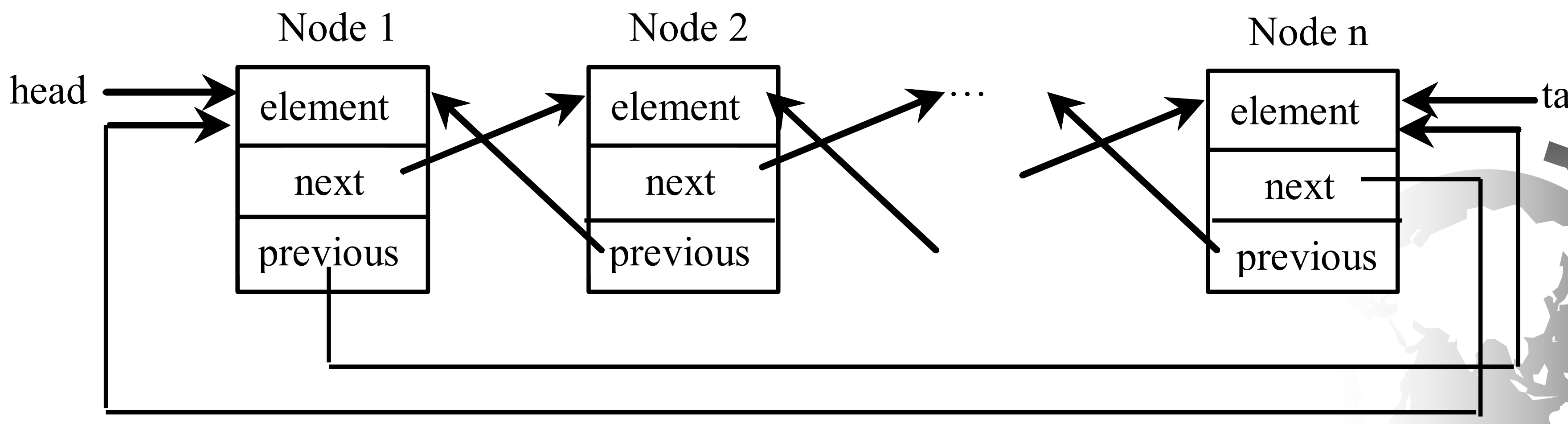
Doubly Linked Lists

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.



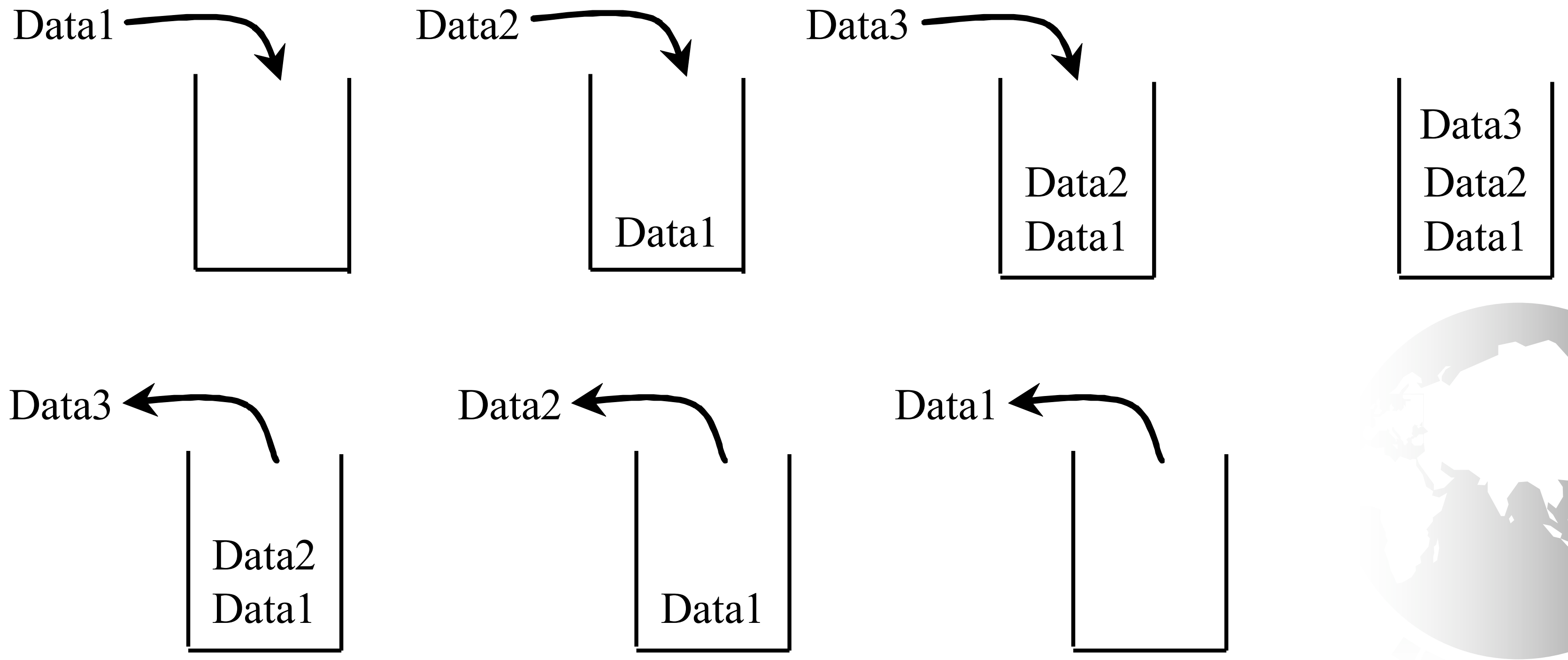
Circular Doubly Linked Lists

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.



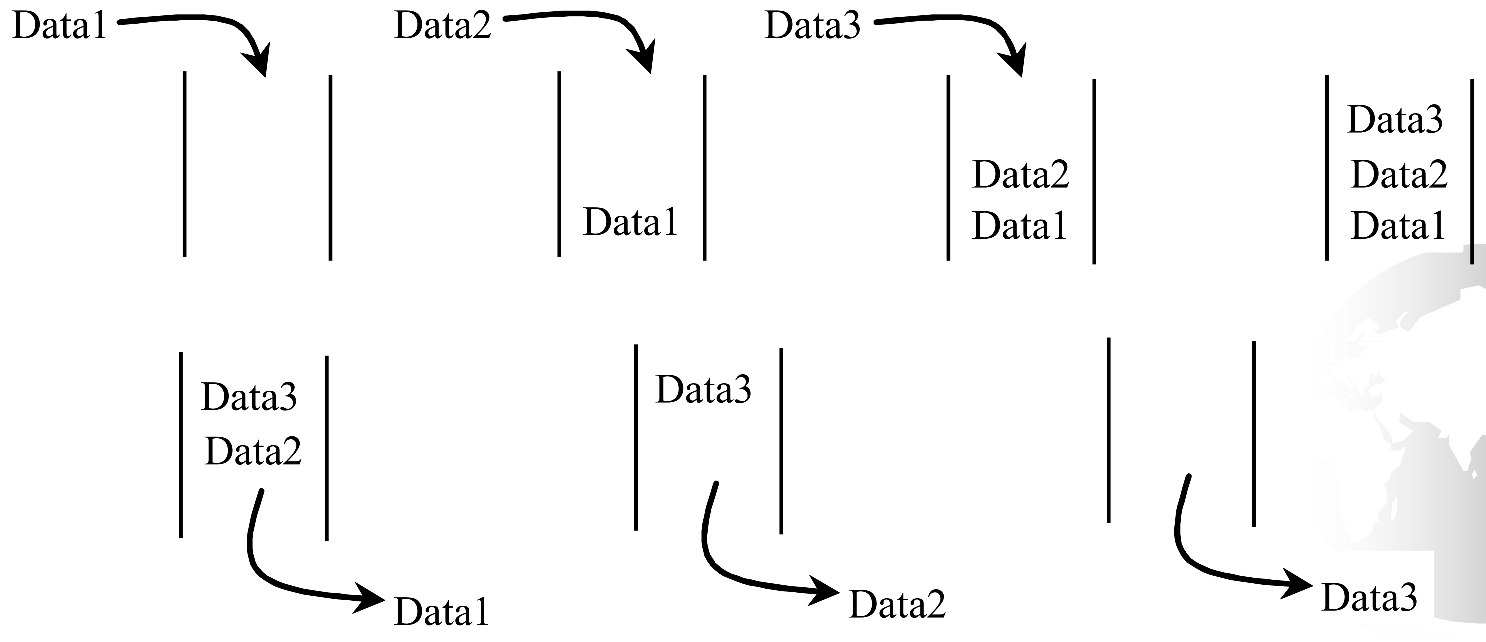
Stacks

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.



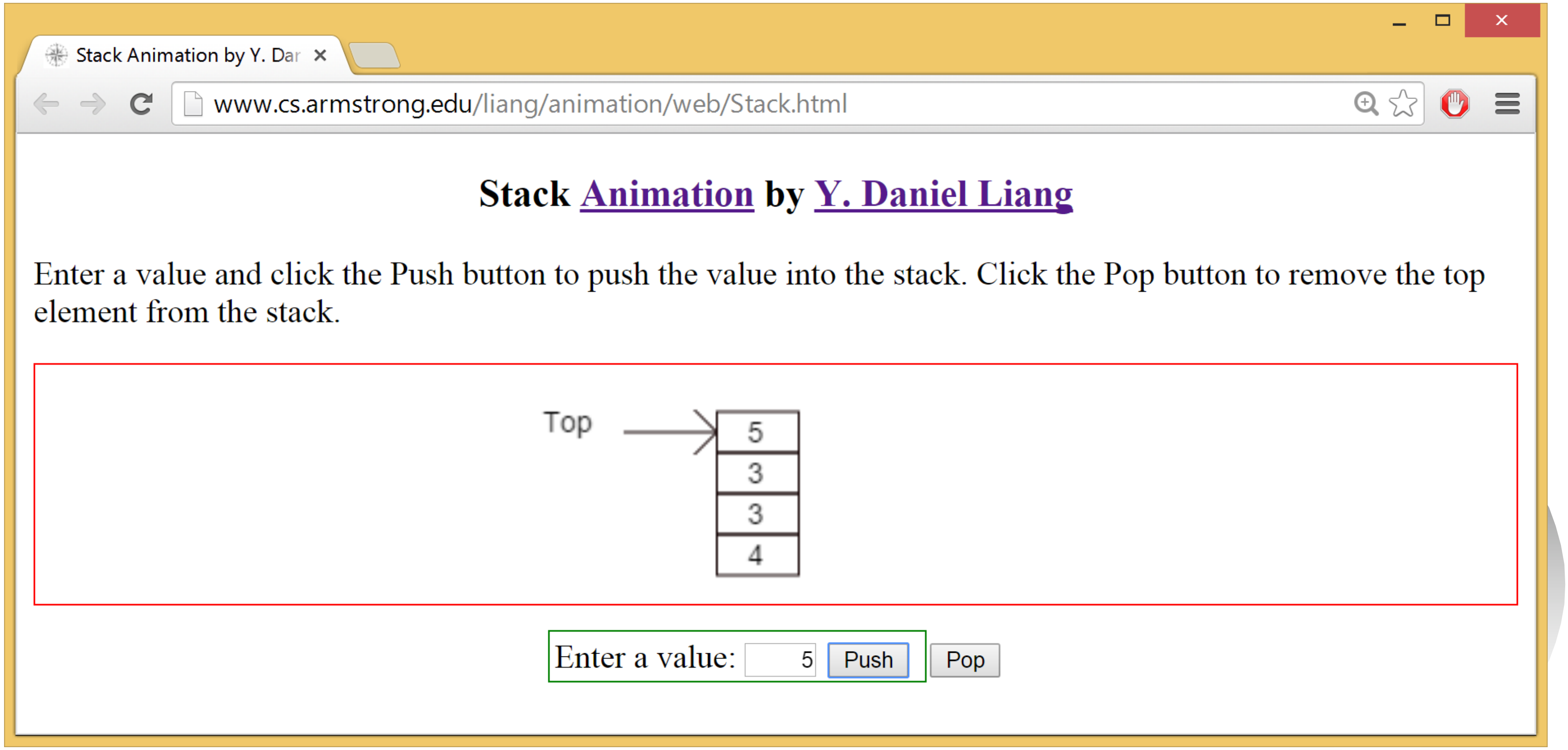
Queues

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

https://liveexample.pearsoncmg.com/dsanimation/StackeBook.html


Stack Animation by Y. Dar

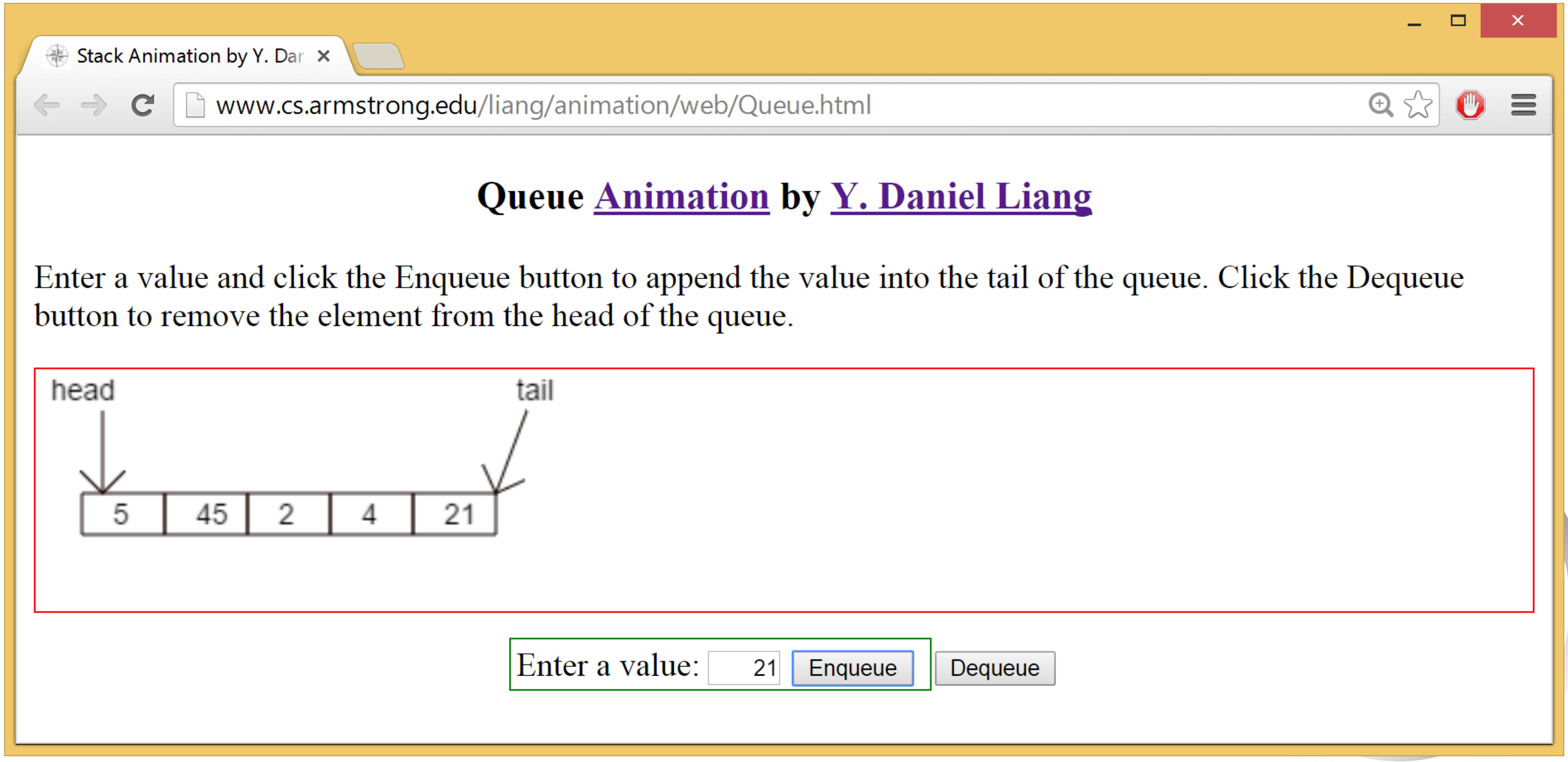
www.cs.armstrong.edu/liang/animation/web/Stack.html

Stack Animation by Y. Daniel Liang

Enter a value and click the Push button to push the value into the stack. Click the Pop button to remove the top element from the stack.

Queue Animation

https://liveexample.pearsoncmg.com/dsanimation/QueueeBook.html



Stack Animation by Y. Dar
www.cs.armstrong.edu/liang/animation/web/Queue.html

Queue Animation by Y. Daniel Liang

Enter a value and click the Enqueue button to append the value into the tail of the queue. Click the Dequeue button to remove the element from the head of the queue.

head
tail
5
45
2
4
21
Enter a value: 21
Enqueue
Dequeue

Implementing Stacks and Queues

Using an array list to implement Stack

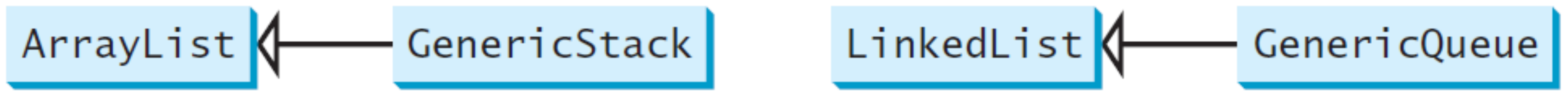
Use a linked list to implement Queue

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

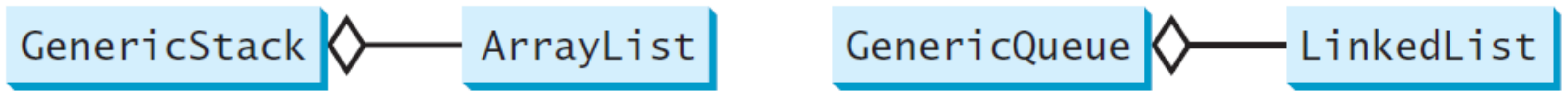
There are two ways to design the stack and queue classes:

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



(a) Using inheritance

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



(b) Using composition

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

GenericStack

GenericStack<E>	
-list: java.util.ArrayList<E>	An array list to store elements.
+GenericStack()	Creates an empty stack.
+getSize(): int	Returns the number of elements in this stack.
+peek(): E	Returns the top element in this stack.
+pop(): E	Returns and removes the top element in this stack.
+push(o: E): void	Adds a new element to the top of this stack.
+isEmpty(): boolean	Returns true if the stack is empty.

GenericQueue

GenericQueue<E>	
-list: LinkedList<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.

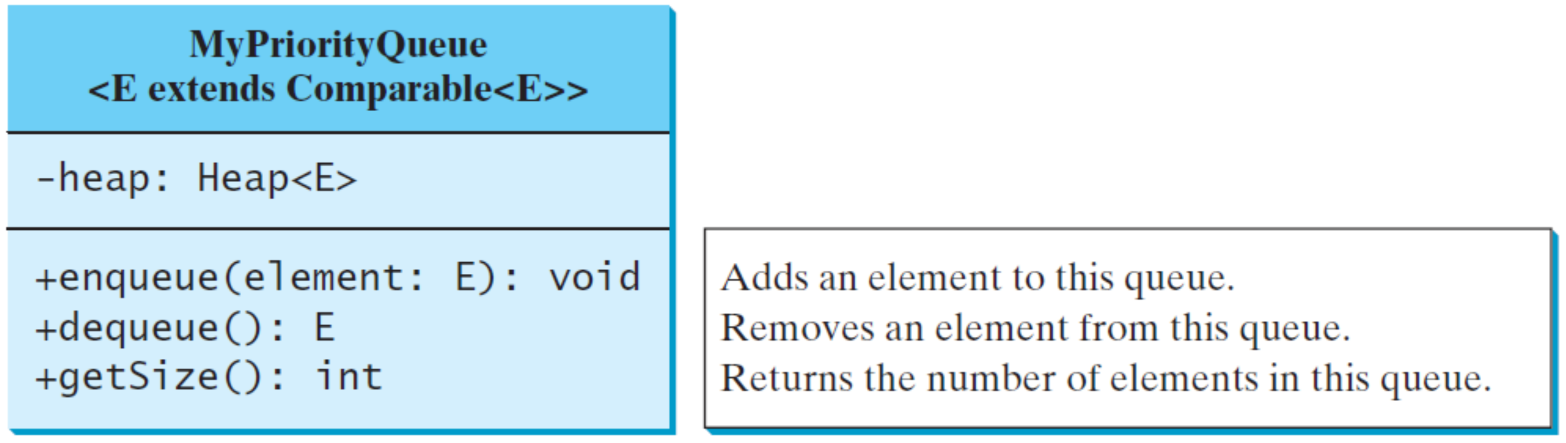
Example: Using Stacks and Queues

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

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 extends Comparable<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 in this queue.

MyPriorityQueue

TestPriorityQueue