Chapter 18 Linked Lists, Stacks, Queues, and Priority Queues

Objectives

- To design and implement linked lists (§18.2).
- To explore variations of linked lists (§18.2).
- To define and create iterators for traversing elements in a container (§18.3).
- To generate iterators using generators (§18.4).
- To design and implement stacks (§18.5).
- To design and implement queues (§18.6).
- To design and implement priority queues (§18.7).

What is a Data Structure?

A data structure is a collection of data organized in some fashion. A data structure not only stores data, but also supports the operations for manipulating data in the structure. For example, an array is a data structure that holds a collection of data in sequential order. You can find the size of the array, store, retrieve, and modify data in the array.

Array is simple and easy to use, but it has two limitations:

Limitations of arrays

- Once an array is created, its size cannot be altered.

- Array provides inadequate support for inserting, deleting, sorting, and searching operations.

Object-Oriented Data Structure

In object-oriented thinking, a data structure is an object that stores other objects, referred to as data or elements. So some people refer a data structure as a *container object* or a *collection object*. To define a data structure is essentially to declare a class. The class for a data structure should use data fields to store data and provide methods to support operations such as insertion and deletion. To create a data structure is therefore to create an instance from the class. You can then apply the methods on the instance to manipulate the data structure such as inserting an element to the data structure or deleting an element from the data structure.

Four Classic Data Structures

Four classic dynamic data structures to be introduced in this chapter are lists, stacks, queues, and priority queues. 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. In a priority queue, elements are assigned with priorities. When accessing elements, the element with the highest priority is removed first.

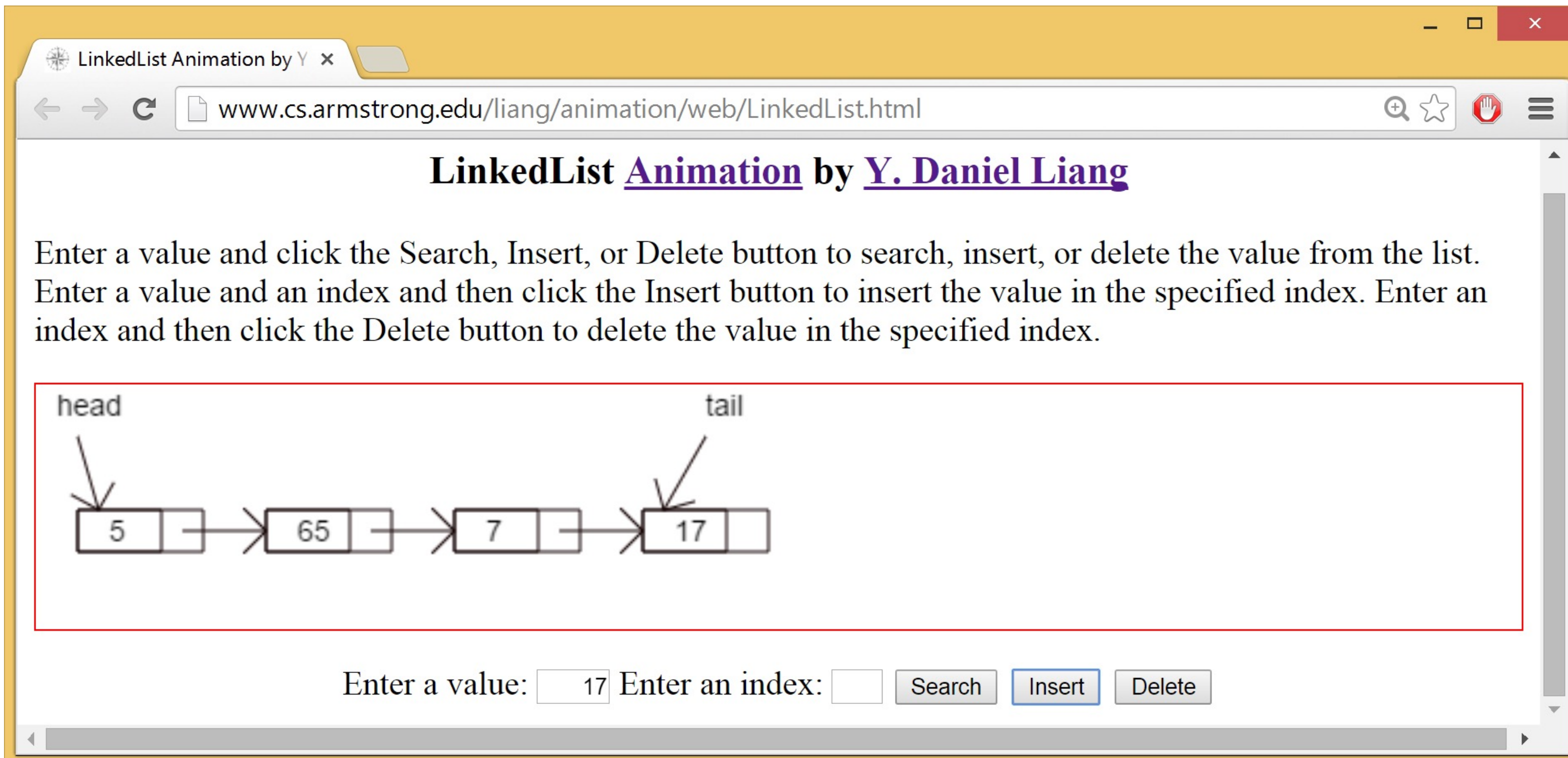
Lists

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.

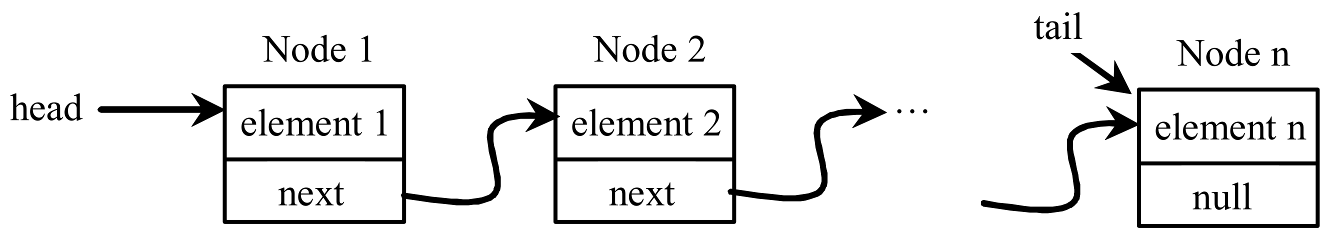
Linked List Animation

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



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:
    def __init__(self, element):
        self.elmenet = element
        self.next = None
```

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 None. For example, you can create three nodes to store three strings in a list, as follows:

Step 1: Declare head and tail:

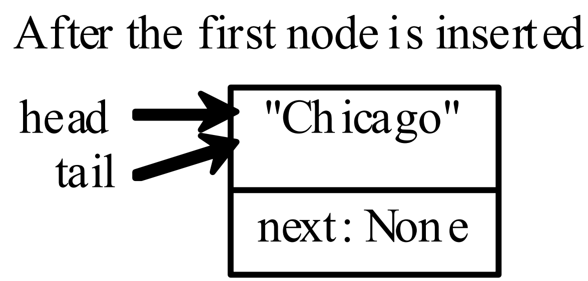
```
head = None
tail = None
```

The list is empty now

Adding Three Nodes, cont.

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

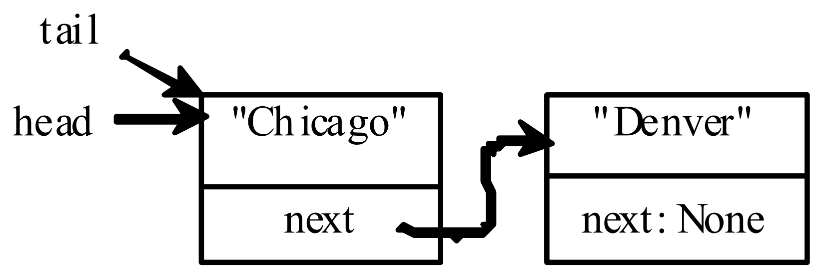
```
head = Node("Chicago")
last = head
```



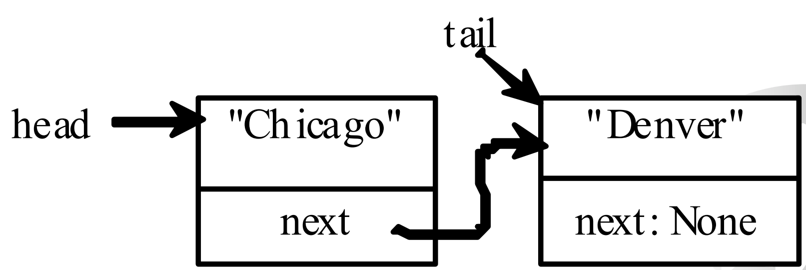
Adding Three Nodes, cont.

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

```
tail.next = Node("Denver")
```



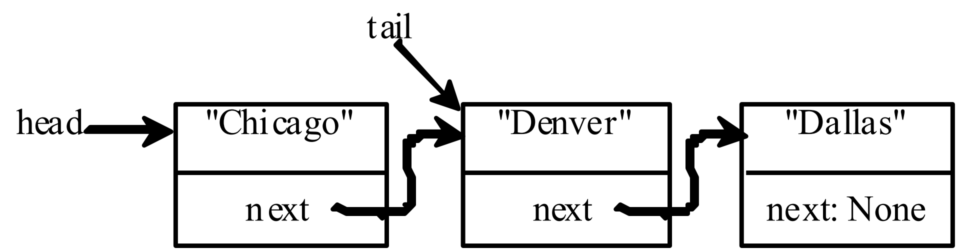

```
tail = tail.next
```



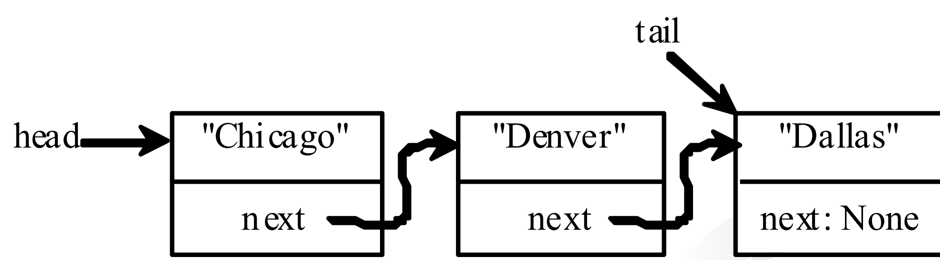
Adding Three Nodes, cont.

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

```
tail.next = Node("Dallas")
```




```
tail = tail.next
```

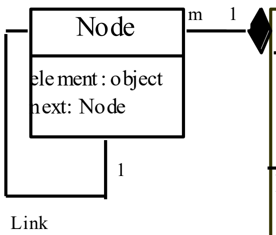


Traversing All Elements in the List

Each node contains the element and a data field named *next* that points to the next element. If the node is the last in the list, its pointer data field next contains the value None. 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.

```
current = head
while current != None:
    print(current.element)
    current = current.next
```

LinkedList

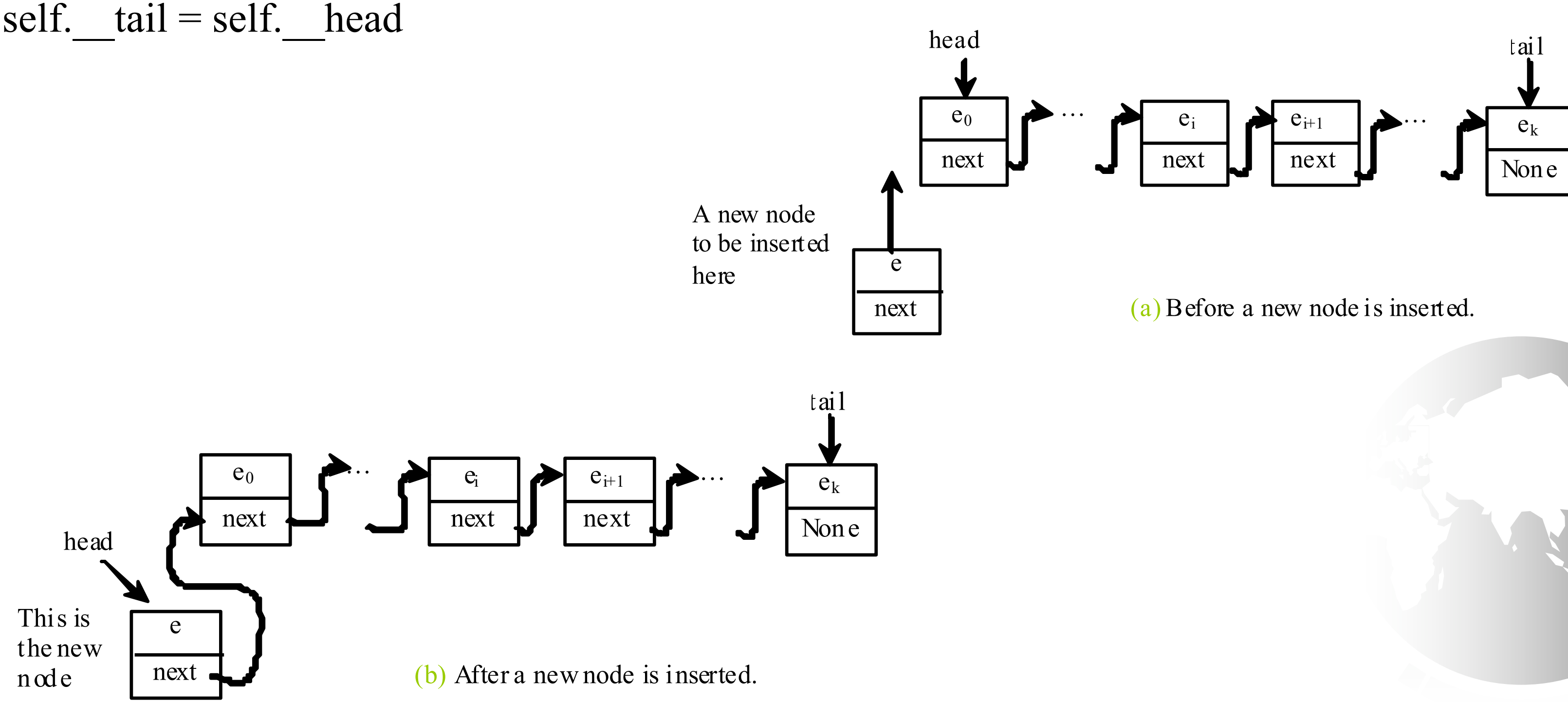


LinkedList	
-head: Node	
-tail: Node	
-size: int	
LinkedList()	Creates an empty linked list.
addFirst(e: object): None	Adds a new element to the head of the list.
addLast(e: object): None	Adds a new element to the tail of the list.
getFirst(): object	Returns the first element in the list.
getLast(): object	Returns the last element in the list.
removeFirst(): object	Removes the first element from the list.
removeLast(): object	Removes the last element from the list.
add(e: object) : None	Same as addLast(e).
insert(index: int, e: object) : None	Adds a new element at the specified index.
clear(): None	Removes all the elements from this list.
contains(e: object): bool	Returns true if this list contains the element.
get(index: int) : object	Returns the element from at the specified index.
indexOf(e: object) : int	Returns the index of the first matching element.
isEmpty(): bool	Returns true if this list contains no elements.
lastIndexOf(e: object) : int	Returns the index of the last matching element.
getSize(): int	Returns the number of elements in this list.
remove(e: object): bool	Removes the element from this list.
removeAt(index: int) : object	Removes the element at the specified index and returns the removed element.
set(index: int, e: object) : object	Sets the element at the specified index and returns the element you are replacing.
__iter__() : Iterator	Returns an iterator for this linked list.

LinkedList TestLinkedList

Implementing addFirst(e)

```
def addFirst(self, e):
    newNode = Node(e) # Create a new node
    newNode.next = self.__head # link the new
node with the head
    self.__head = newNode # head points to the
new node
    self.__size += 1 # Increase list size
    if self.__tail == None: # the new node is the
only node in list
        self.__tail = self.__head
```

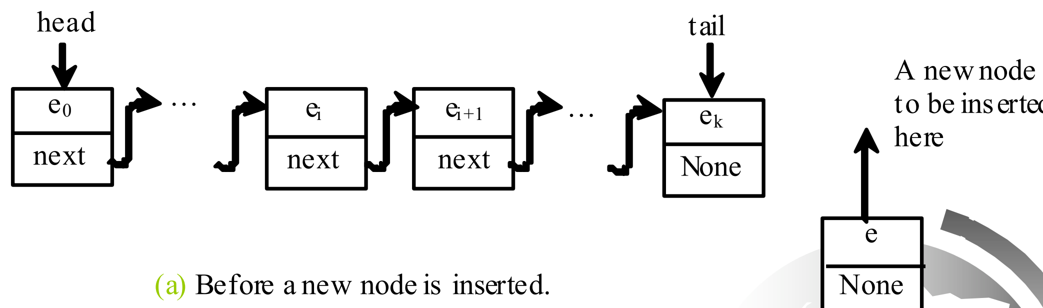


Implementing addLast(e)

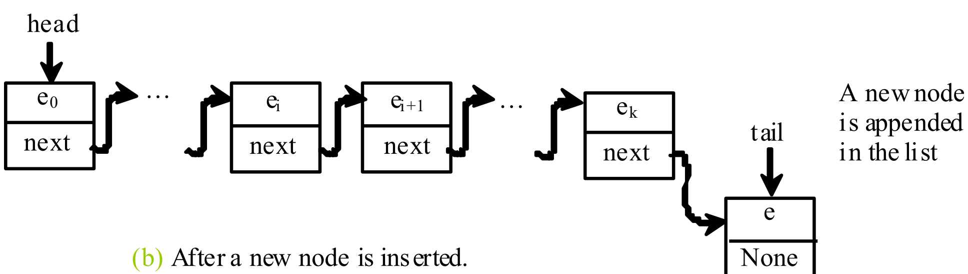
```
def addLast(self, e):
    newNode = Node(e) # Create a new node for e

    if self.__tail == None:
        self.__head = self.__tail = newNode # The only node in list
    else:
        self.__tail.next = newNode # Link the new with the last node
        self.__tail = self.__tail.next # tail now points to the last node

    self.__size += 1 # Increase size
```



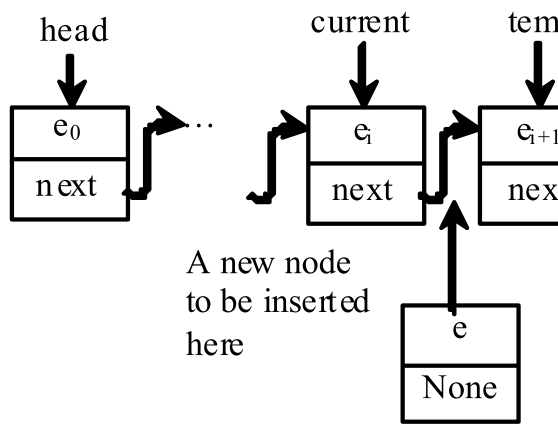
(a) Before a new node is inserted.



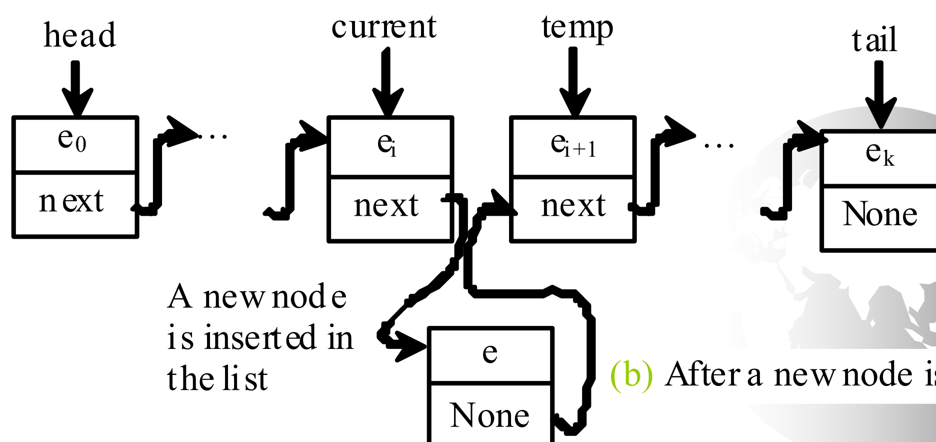
(b) After a new node is inserted.

Implementing add(index, e)

```
def insert(self, index, e):
    if index == 0:
        self.addFirst(e) # Insert first
    elif index >= size:
        self.addLast(e) # Insert last
    else: # Insert in the middle
        current = head
        for i in range(1, index):
            current = current.next
        temp = current.next
        current.next = Node(e)
        (current.next).next = temp
        self.__size += 1
```



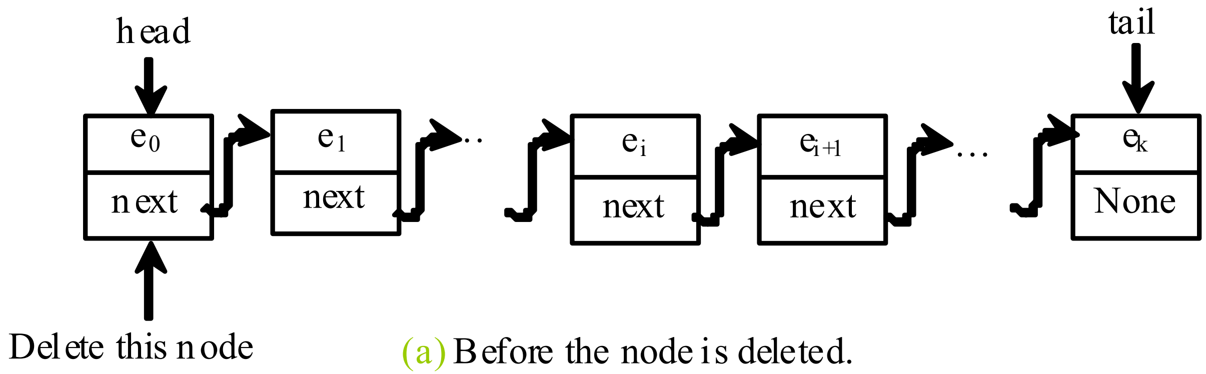
(a) Before a new node is inserted.



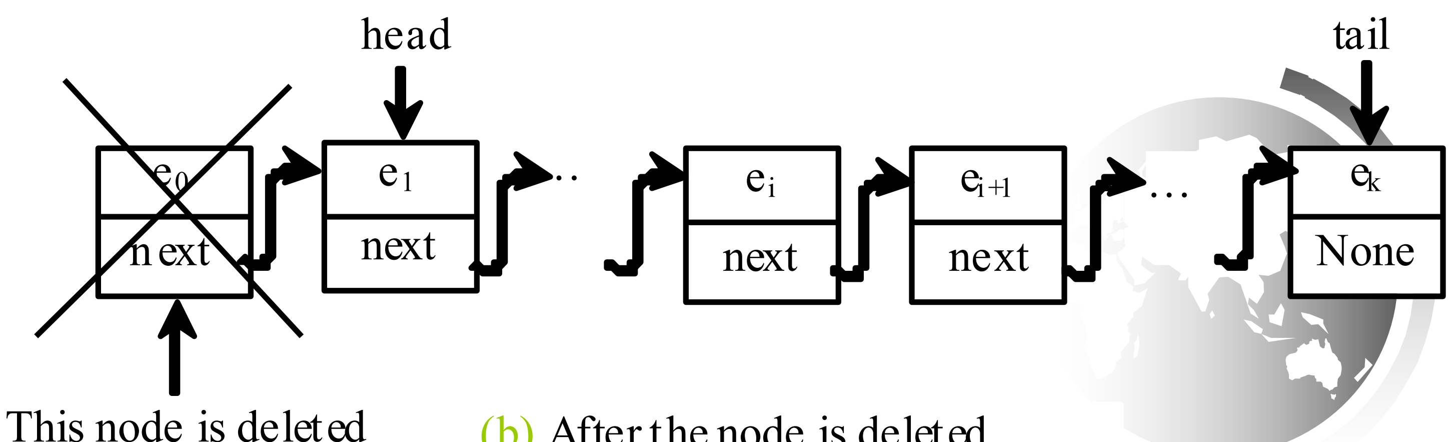
(b) After a new node is inserted.

Implementing removeFirst()

```
def removeFirst(self):
    if self.__size == 0:
        return None
    else:
        temp = self.__head
        self.__head =
        self.__head.next self.__size -= 1
    if self.__head == None:
       self.__tail = None return temp.element
```



(a) Before the node is deleted.



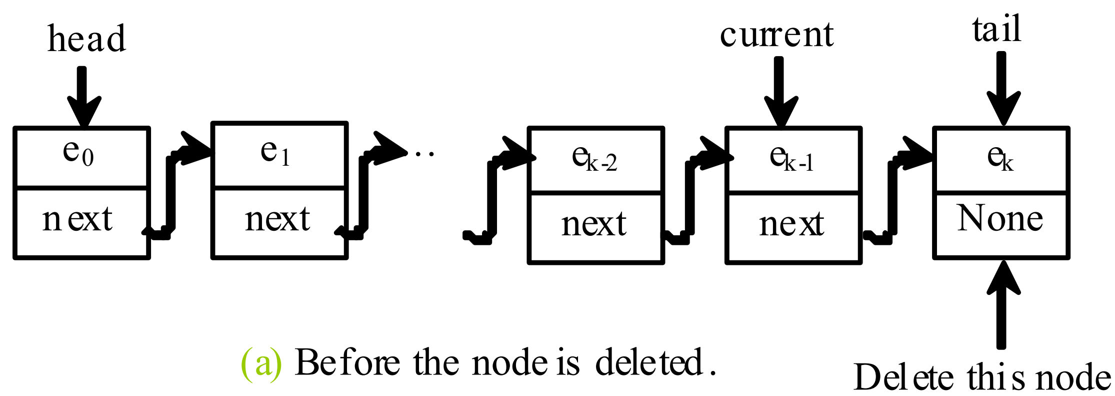
(b) After the node is deleted.

Implementing removeLast()

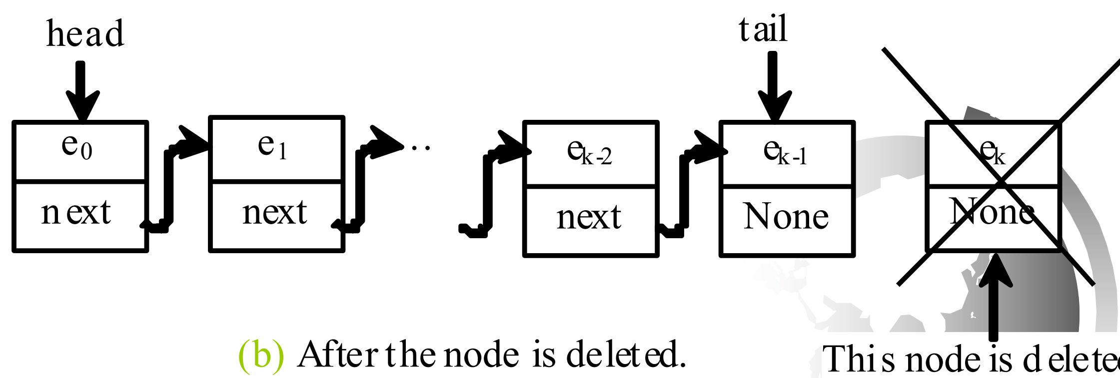
```
def removeLast(self):
    if self.__size == 0:
        return None
    elif self.__size == 1:
        temp = self.__head
        self.__head = self.__tail = None
        self.__size = 0
        return temp.element
    else:
        current = self.__head

        for i in range(self.__size - 2):
            current = current.next

        temp = self.__tail
        self.__tail = current
        self.__tail.next = None
        self.__size -= 1
        return temp.element
```



(a) Before the node is deleted.



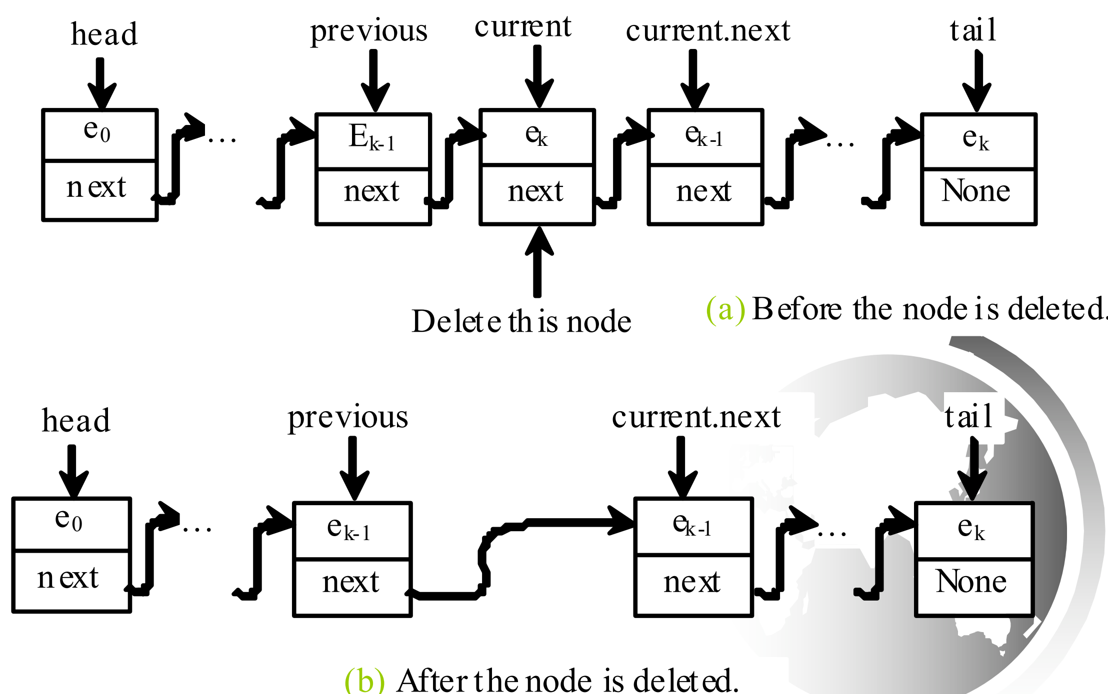
(b) After the node is deleted.

Implementing removeAt(index)

```
def removeAt(self, index):
    if index < 0 or index >= self.__size:
        return None # Out of range
    elif index == 0:
        return self.removeFirst() # Remove first
    elif index == self.__size - 1:
        return self.removeLast() # Remove last
    else:
        previous = self.__head

        for i in range(1, index):
            previous = previous.next

        current = previous.next
        previous.next = current.next
        self.__size -= 1
        return current.element
```



Time Complexity for list and LinkedList

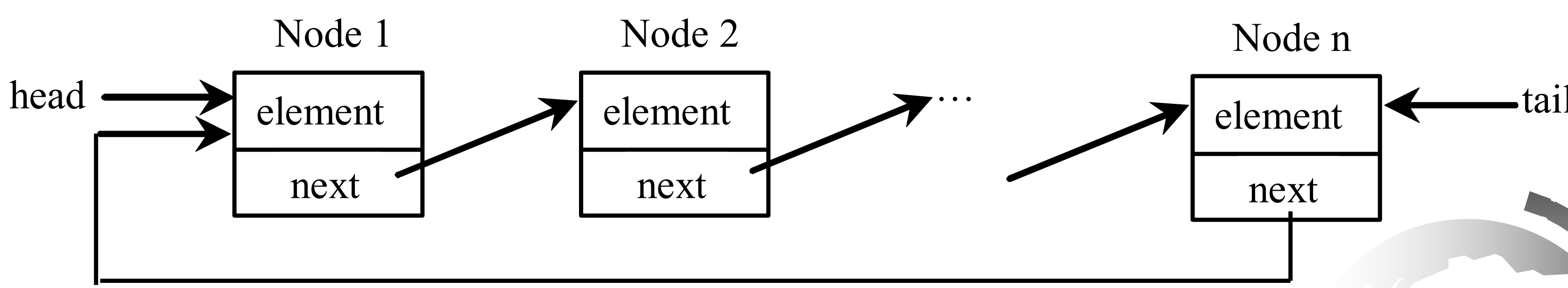
Methods for list/Complexity		Methods for LinkedList/Complexity	
`append(e: E)`	$O(1)$	`add(e: E)`	$O(1)$
`insert(index: int, e: E)`	$O(n)$	`insert(index: int, e: E)`	$O(n)$
N/A		`clear()`	$O(1)$
`e in myList`	$O(n)$	`contains(e: E)`	$O(n)$
`list[index]`	$O(1)$	`get(index: int)`	$O(n)$
`index(e: E)`	$O(n)$	`indexOf(e: E)`	$O(n)$
`len(x) == 0?`	$O(1)$	`isEmpty()`	$O(1)$
N/A		`lastIndexOf(e: E)`	$O(n)$
`remove(e: E)`	$O(n)$	`remove(e: E)`	$O(n)$
`len(x)`	$O(1)$	`getSize()`	$O(1)$
`del x[index]`	$O(n)$	`removeAt(index: int)`	$O(n)$
`x[index] = e`	$O(n)$	`set(index: int, e: E)`	$O(n)$
`insert(0, e)`	$O(n)$	`addFirst(e: E)`	$O(1)$
`del x[0]`	$O(n)$	`removeFirst()`	$O(1)$

Comparing list with LinkedList

LinkedListPerformance

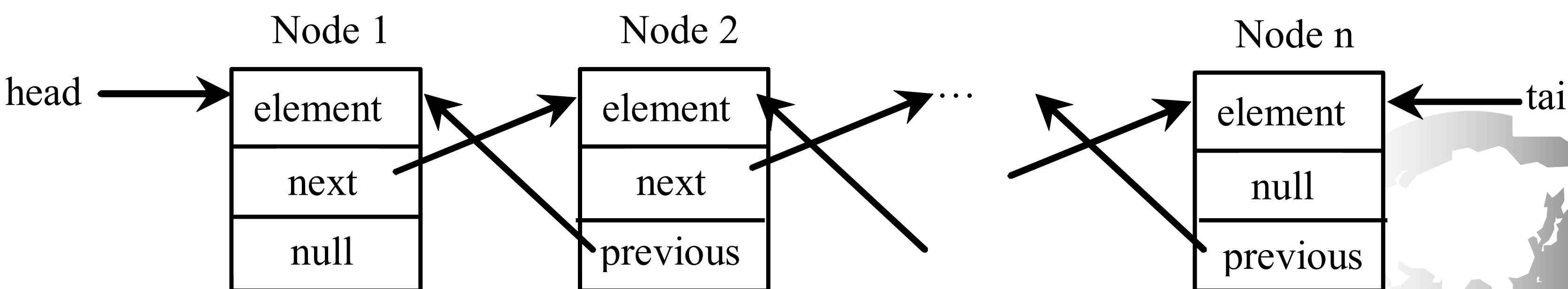
Circular Linked Lists

- 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.



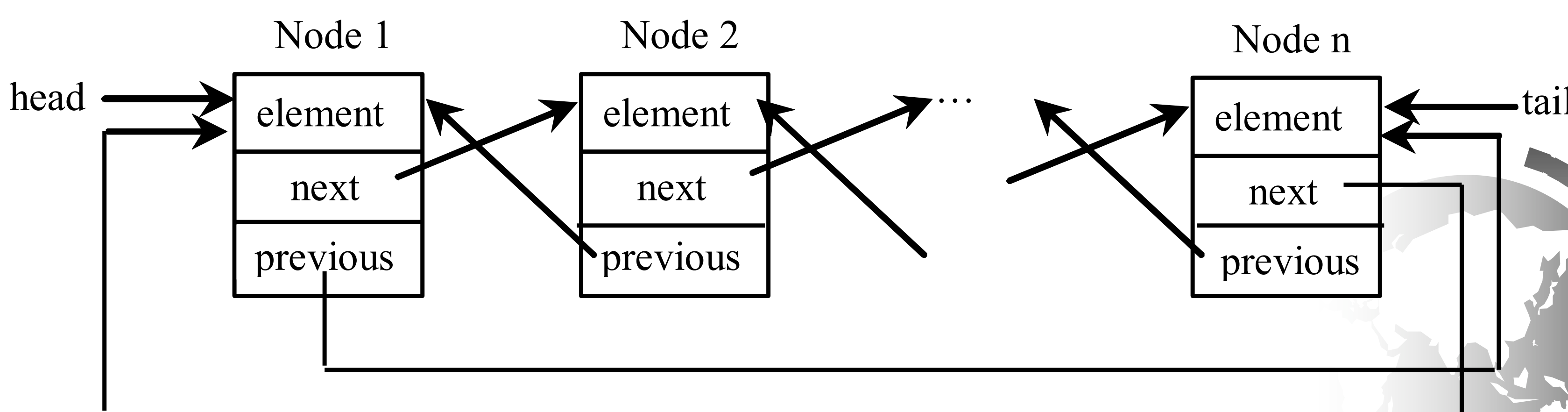
Doubly Linked Lists

- 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.



Iterators

An *iterator* is an object that provides a uniformed way for traversing the elements in a container object. Recall that you can use a for loop to traverse the elements in a list, a tuple, a set, a dictionary, and a string. For example, the following code displays all the elements in set1 that are greater than 3.

```
set1 = {4, 5, 1, 9}
for e in set1:
    if e > 3:
        print(e, end = ' ')
```

__iter__ Method

Can you use a for loop to traverse the elements in a linked list? To enable the traversal using a for loop in a container object, the container class must implement the __iter__() method that returns an iterator as shown in lines 112-114 in Listing 18.2, LinkedList.py.

```
def __iter__(self):
    return LinkedListIterator(self.__head)
```

__next__ Method

An iterator class must contains the __next__() method that returns the next element in the container object as shown in lines in lines 122-132 in Listing 18.2, LinkedList.py.

LinkedList

TestIterator

FibonacciNumberIterator

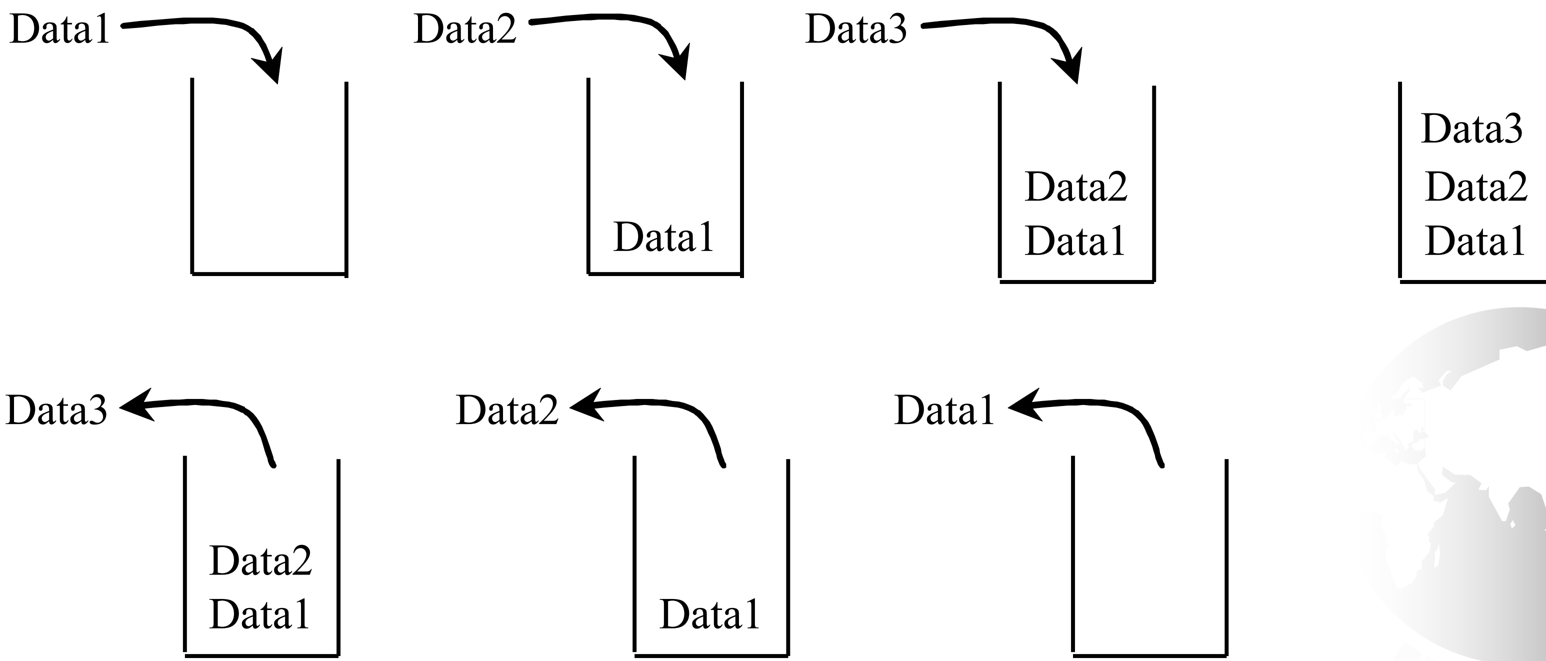
Generators

Generators are special Python functions for generating iterators. They are written like regular functions but use the yield statement to return data. To see how generators work, we rewrite Listing 18.5 FibnacciNumberIterator.py using a generator in Listing 18.6.

FibonacciNumberGenerator

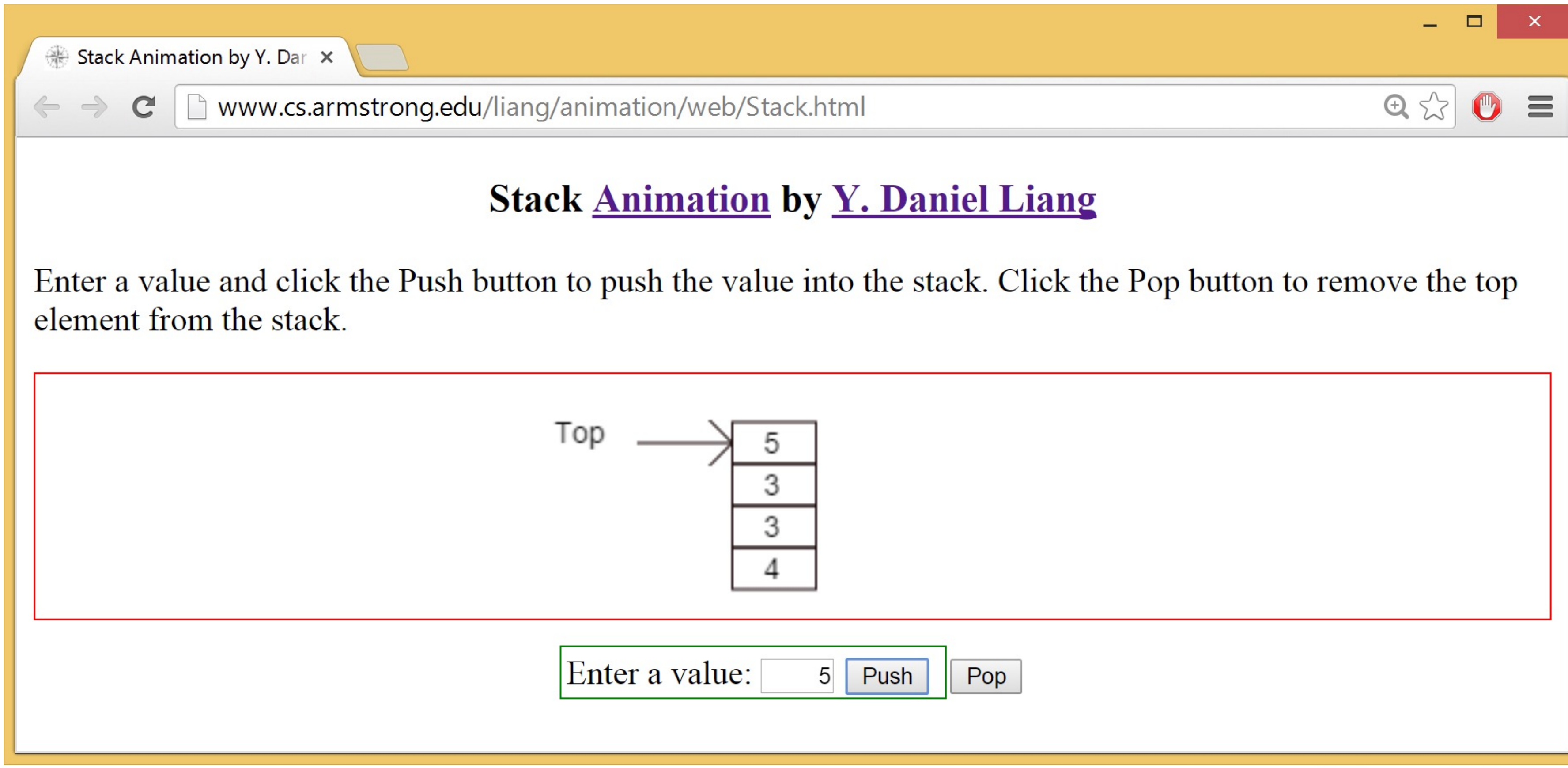
Stacks

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.



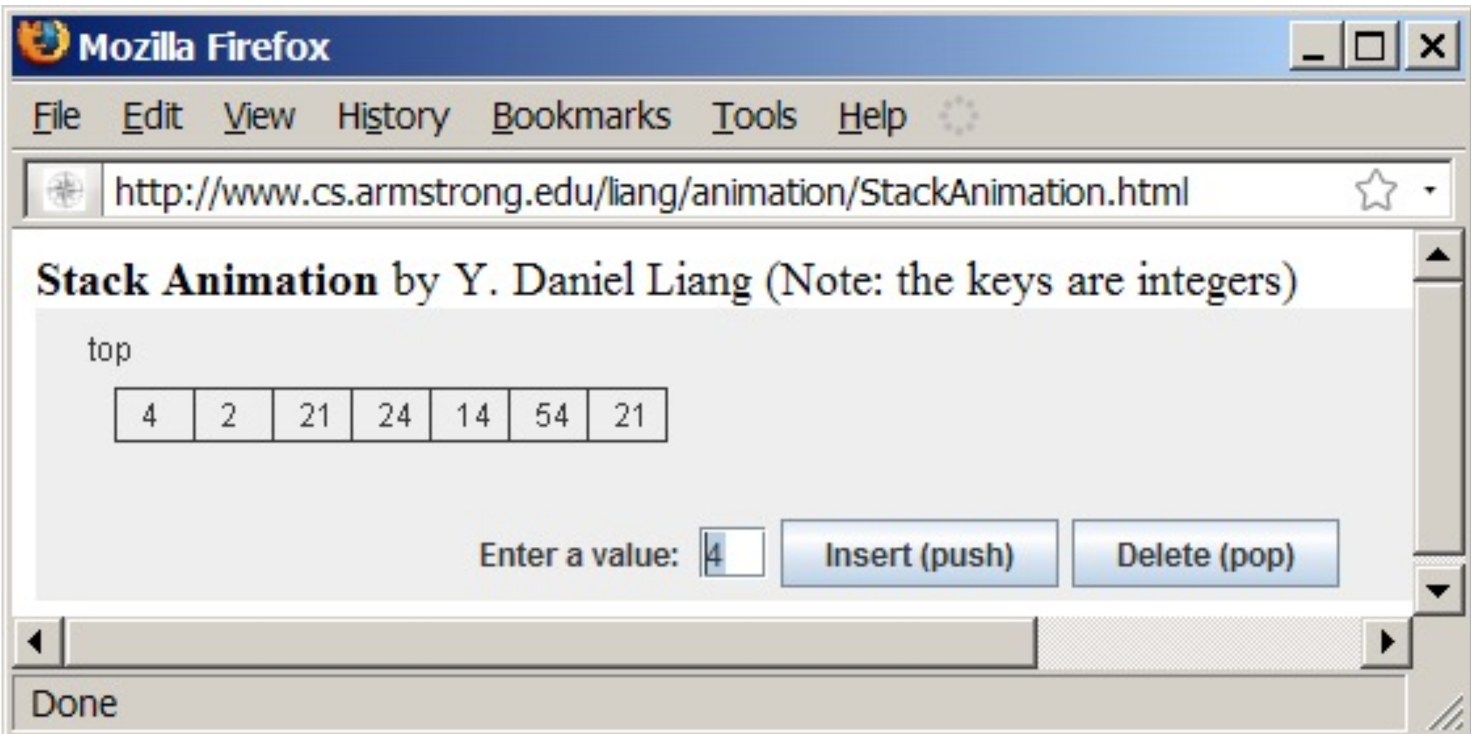
Stack Animation

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



Stack Animation

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



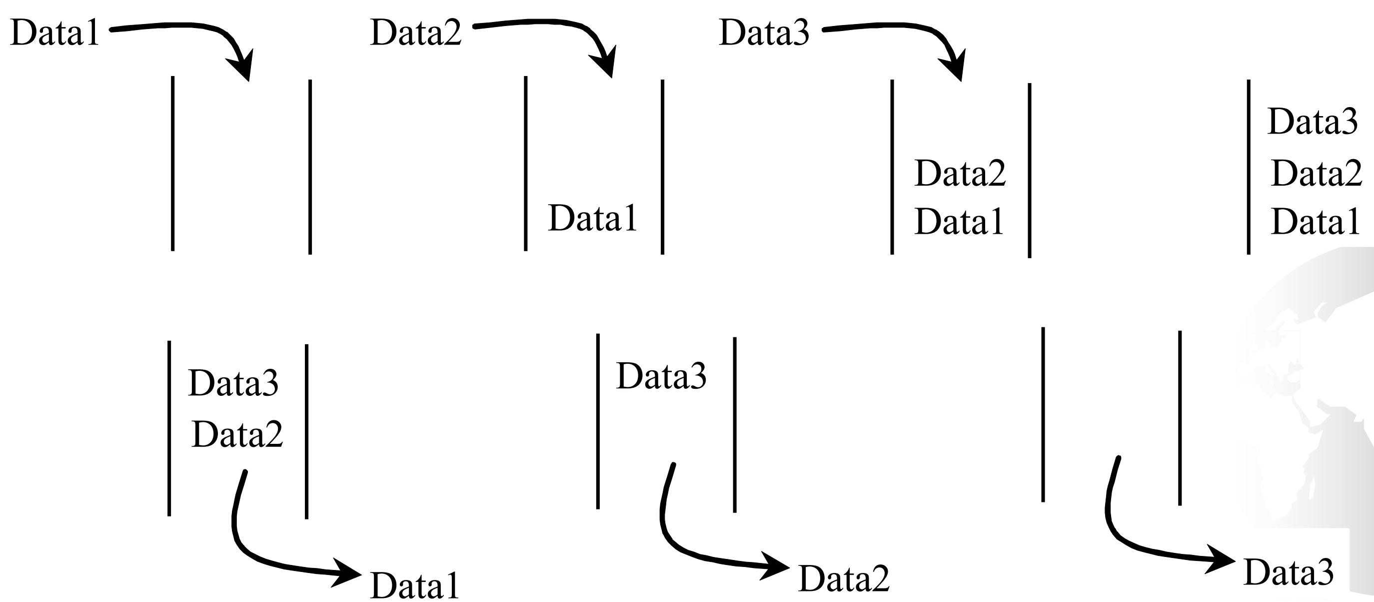
Stack

Stack	
-elements: list	A list to store the elements in the stack.
Stack()	Constructs an empty stack.
isEmpty(): bool	Returns true if the stack is empty.
peek(): object	Returns the element at the top of the stack without removing it from the stack.
push(value: object): None	Stores an element into the top of the stack.
pop():object	Removes the element at the top of the stack and returns it.
getSize(): int	Returns the number of elements in the stack.

Stack TestStack

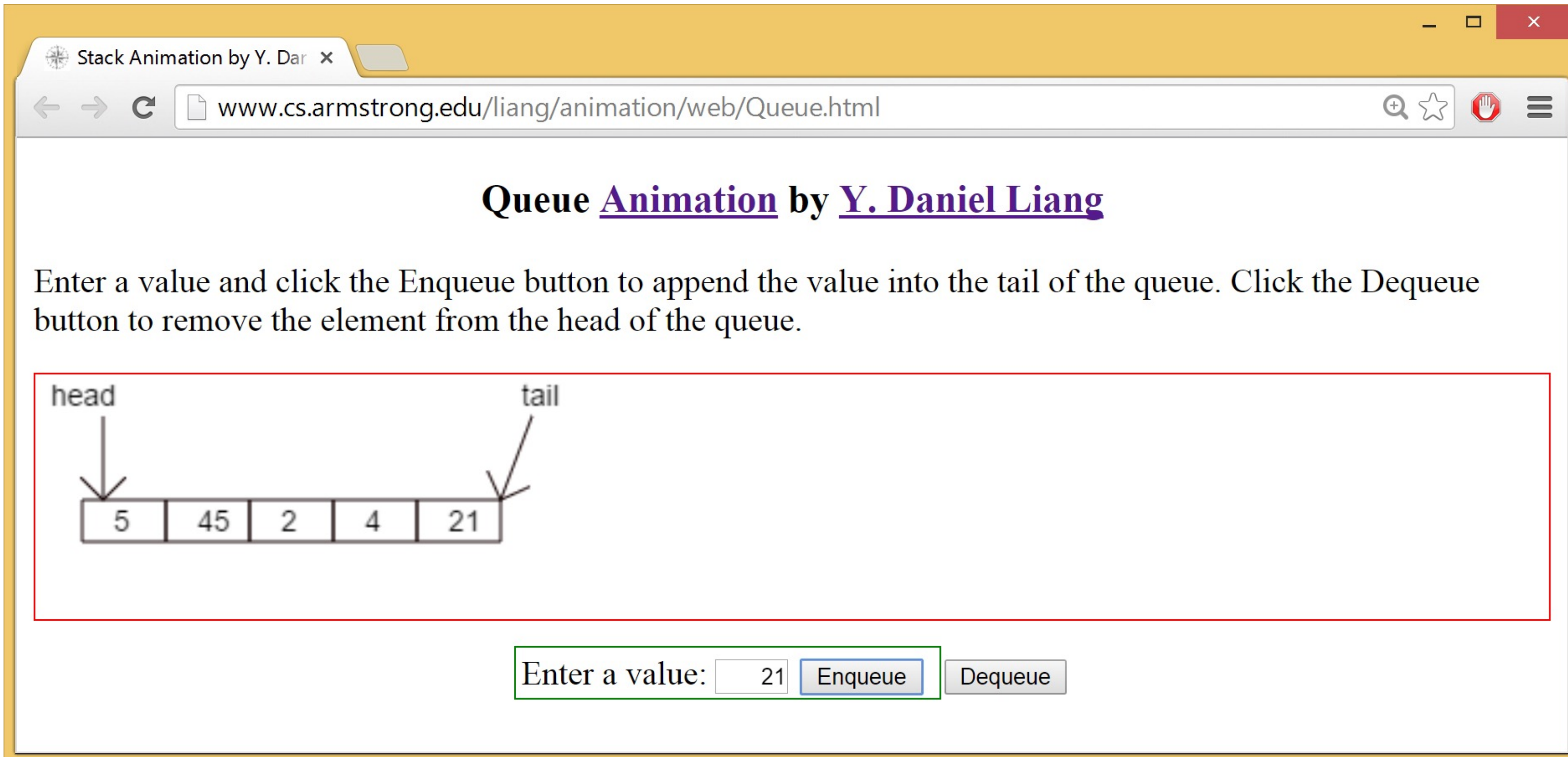
Queues

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.



Queue Animation

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



Queue

Queue	
-elements LinkedList	Stores queus elements in a list.
Queue()	Creates an empty queue.
enqueue(e: object): None	Adds an element to this queue.
dequeue(): object	Removes an element from this queue.
getSize(): int	Returns the number of elements from this queue.
isEmpty(): bool	Returns true if the queue is empty.
__str__(): str	Returns a string representation of the queue.

Queue   TestQueue

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.

PriorityQueue	
-heap: Heap	Elements are stored in a heap.
enqueue(element: object): None	Adds an element to this queue.
dequeue(): objecct	Removes an element from this queue.
getSize(): int	Returns the number of elements from this que

PriorityQueue   TestPriorityQueue