**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# **Introduction**

A **Map** is a type of collection that associates a key with a value. The mapping of keys to values can be accomplished using different underlying data structures. In this three-part assignment, you will be working with two such data structures and then using them to implement a **Map**. One is called a **Linked List** and the other is called a **Tree**.

In part I, you will be implementing a Linked List.

Since the key-value pairs stored in the map could be of any type, we will be using [Generics](https://www.geeksforgeeks.org/generics-in-java/) to implement all our classes. For example, we could have < “Andrew”, 1> as a key-value pair (which is of type <String, Integer>) and we could also have < 432, 34> as a key-value pair (which is of type <Integer, Integer>). Hence, it is important that we provide a way to reuse the same code with different inputs. **Generics** enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods. Much like the more familiar *formal parameters* used in method declarations, type parameters provide a way for you to re-use the same code with different inputs. The difference is that the inputs to formal parameters are values, while the inputs to type parameters are types.

In this part of the assignment, you will be implementing a ***doubly linked list***.

**Note:** You are **NOT** allowed to use any built-in Java data structures for any part of this assignment.

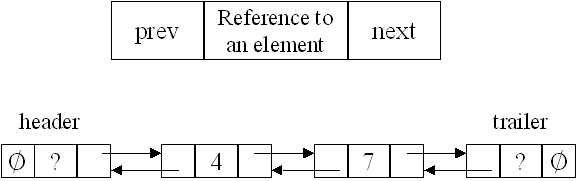
## **What is a Doubly Linked List?**

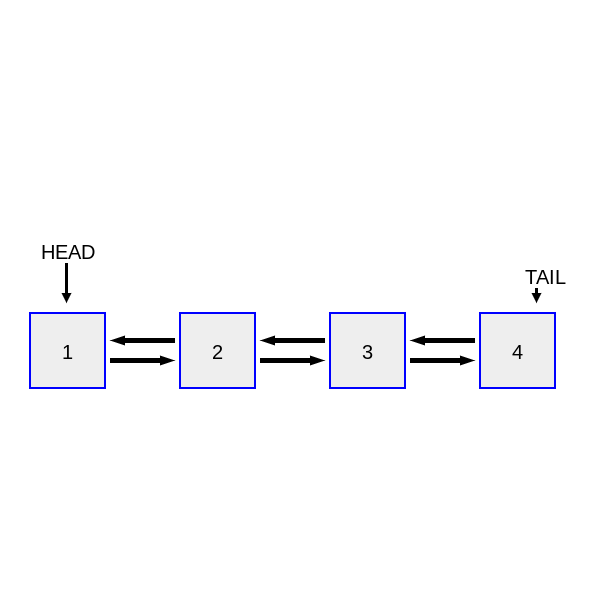
A Linked List is a linear data structure where each element is a separate object. Linked list elements are not stored at contiguous locations; the elements are linked using pointers. A linked list can either be singly linked or doubly linked. In the case of a **singly linked list**, each node in the list points to the node which is next to it on the list. However, there is no way to know which node comes *before* a node in the list because the singly linked lists do not have pointers to previous nodes in the list. **Doubly linked lists** overcome this limitation by storing two pointers---one for the previous node in the list and the other for the next node in the list.

There are two special nodes in the list called the head and tail nodes. The **head** points to the first node in the list whereas the **tail** points to the last node in the list. Each node also stores some data value apart from the pointers.

# **Implementation**

The node and list structure that you will be implementing is this:





# **Program and Starter Code**

You are being provided with starter code which is an abstract class representing a **List** and an interactive **TestDriver**, <https://cs.unh.edu/~cs416/public/3P>. It contains the signatures of the methods that you need to implement. It also contains an [inner class](https://www.geeksforgeeks.org/inner-class-java/) to represent a **Node** in the **LinkedList**. You have to implement the following methods for the **LinkedList** class:

**public LinkedList( )**

The constructor for LinkedList. It should initialize any necessary fields.

**public Node getHead( )**

Returns the current head of this list. Mostly used for testing purposes.

**public Node getTail( )**

Returns the current tail of this list. Mostly used for testing purposes.

**public boolean add( T value )**

Appends the specified value to the end of this list. Should not add duplicate items to the list. Returns true if added, false otherwise.

**public void add( int index, T value )**

Adds the specified value at the given index of this list. Should not add duplicate items to the list.

**public String toString()**

Returns a string representation of this LinkedList. It should be in this format:

[item1, item2, item3]

**public void clear( )**

Removes all of the elements from this list.

**public T get( int index )**

Returns the element at the specified position in this list. Should return null if the index is invalid.

**public T get( Object o )**

Returns the data that matches the object given. It should use the equals method of the list node's data when comparing to o. Should return null if the object does not match any data in the list.

**public boolean contains( Object o )**

Returns true if this list contains the specified element, false otherwise.

**public boolean isEmpty( )**

Returns true if this list contains no elements, false otherwise.

**public T remove( int index )**

Removes the element at the specified position in this list. Returns the element from the list or null if index is invalid.

**public boolean remove( Object o )**

Removes the first occurrence of the specified element from this list, if it is present. If this list does not contain the element, it is unchanged. Returns true if this list contained the specified element, false otherwise.

**public int size( )**

Returns the number of elements in this list.

# Interactive Test Driver

Provides a user interface for creating and manipulating a LinkedList of strings. Includes a command for each of the methods above. **It should exit when a user enters the command x.**

| **Command** | **Method** |
| --- | --- |
| a | add( T value ) |
| A | add( int index, T value ) |
| c | contains( Object o ) |
| C | clear() |
| g | get( int index ) |
| s | size() |
| e | isEmpty() |
| r | remove( Object o ) |
| R | remove( int index ) |

Example transcript:

| a the  [the]  ------------------  a fox  [the, fox]  ------------------  A 2 jumped  [the, fox, jumped]  ------------------  A 1 quick  [the, quick, fox, jumped]  ------------------  s  4  ------------------  e  false  ------------------  g 1  quick  ------------------  R 3  [the, quick, fox]  ------------------  r fast  [the, quick, fox]  ------------------  r the  [quick, fox]  ------------------  c fox  true  ------------------  c the  false  ------------------  C  []  ------------------  x |
| --- |