CSC220
Chapter 2: Lists and the Collection Interface

Mitsunori Ogihara

University of Miami

September 9, 2010

Outline

Objectives

1. To become familiar with the List interface
2. To understand how to write an array-based implementation of the List interface
3. To study the difference between single-, double-, and circular linked list data structures
4. To learn how to implement the List interface using a linked-list
5. To understand the Iterator interface
6. To learn how to implement the iterator for a linked list
7. To become familiar with the Java Collection framework

The List Interface and ArrayList Class

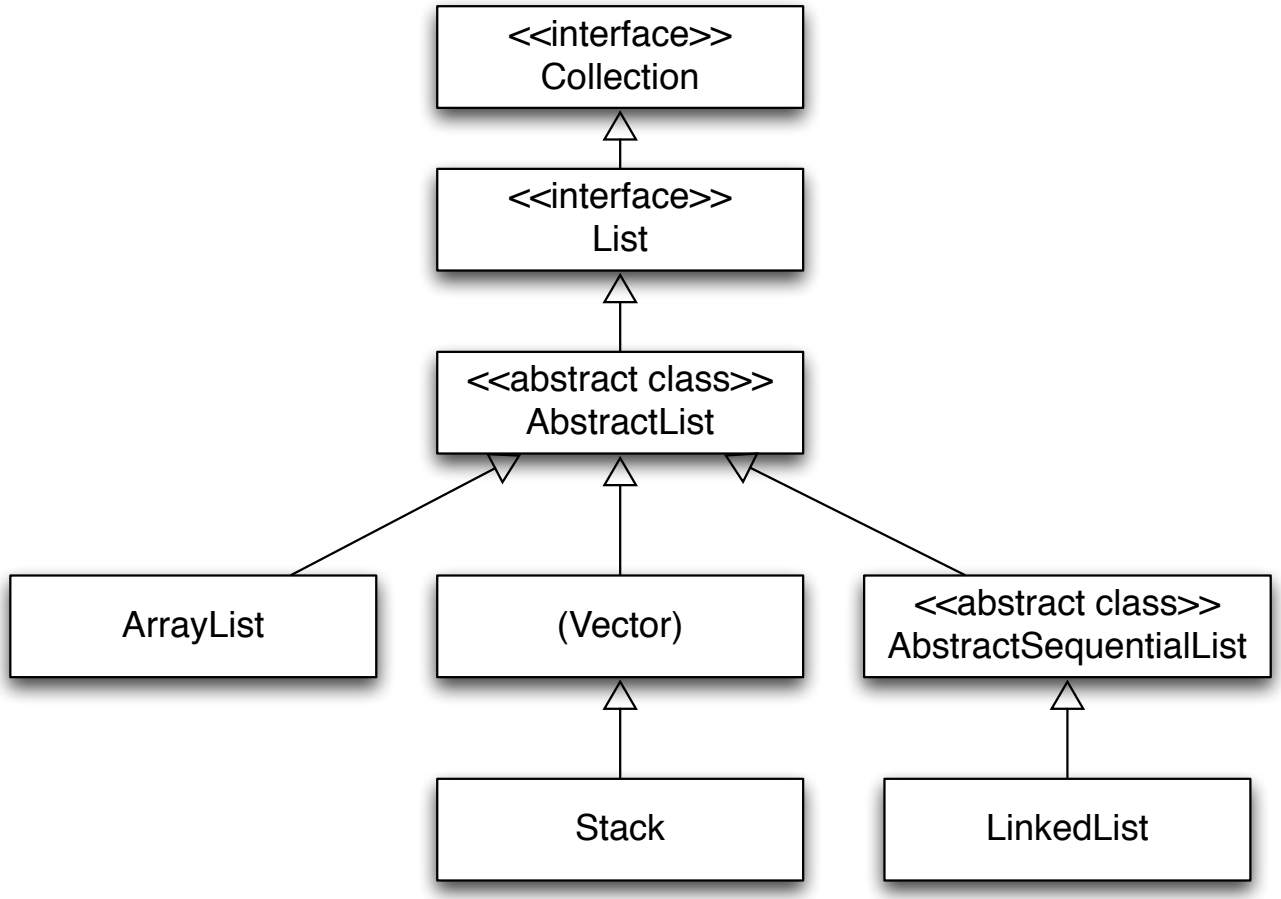
Properties of Arrays:

- Pros: An array is an indexed structure
  - You can select its elements in arbitrary order using a subscript value
  - Elements may be accessed in sequence using a loop that increments the subscript
- Cons:
  - You cannot increase or decrease the length
  - You cannot insert/remove an element without shifting the elements after it

The List Interface and ArrayList Class (cont'd)

- Allowed operations on the List interface include:
  - Checking whether a given data object appears in the list
  - Adding, removing, and replacing an element at a given location
  - Adding and removing an element at the end
  - Obtaining the size
  - Returning an object that allows sequential scanning of the data objects without indexing
- Not all classes implementing the interface perform the allowed operations with the same degree of efficiency
- An array provides the ability to store primitive-type data whereas the List classes all store references to Objects. Autoboxing facilitates this.

The List Interface and ArrayList Class



ArrayList

An implementation of the List interface using array as a method of storage.
The class consists of three fields:

1. The actual array
2. The array size . . . capacity
3. The number of elements stored in the array . . . size

- Insertion and removal are executed by moving all the elements after the point of insertion/removal
- Search is executed by sequentially scanning of the array
- If there is no room for insertion, double the size of the array
  - Create a new double-sized array
  - Move all the elements from the current array to the new one
  - Add the new element

ArrayList is a Generic Class

A generic class is a class defined with some parameters that specify types of data objects that can be dealt with

```
List<String> myList = new
ArrayList<String>();
```

specifies

- myList will be considered to be a List class with String as the type of data to be stored
- myList is actually an ArrayList object with String as the type of data to be stored

Generic Type Declaration in a Class Header

- `public class Foo<E> implements FooInt<E>;`
- `public class Bar<E,F>;`
- `public interface MyInt<E extends Comparable<E>>;`
- `public class MyMy<E extends Comparable<E>> implements MyInt<E>;`

The last two mean that MyInt and MyMy can be used to store data that has the method `compareTo`

ArrayList Internal Representation

An important thing to note about ArrayList is that you cannot create an array of a generic type!

Solution

Use Object as the class and down-cast if necessary.
For example, in the empty constructor, you might want to execute:

```
private int capacity = ARRAY_LIST_INITIAL_SIZE;
private Object[] data = new Object[capacity];
private int size = 0;
```

and then later, for returning an object at index *i* as an E object, use

```
return (E)data[i];
```

ArrayList Operation Headers

E get(int index)	Returns the item at position index
E set(int index, E anEntry)	Replaces the item at the index; returns the previous value
int size()	Returns the number of items in the list
boolean add(E anEntry)	Inserts at the end
void add(int index, E anEntry)	Inserts a reference to anEntry at position index
int indexOf(E target)	Returns the position of the first occurrence of target; returns -1 if target doesn't appear
E remove(int index)	Remove the item at position index and returns the removed item

Advantages and Disadvantages of ArrayList

- set and get require constant time
- add and remove require linear time

More About Addition

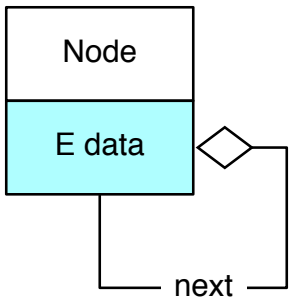
How do we deal with the capacity has been reached?
We will use array size doubling.

List structure

- Linked list overcomes this by providing ability to add or remove items anywhere in the list in constant time, but at the cost of slow indexing
- Each element (node) in a linked list stores information and a link to the next, and optionally previous, node

Basic Component in List

A “node” consisting of a field for storing a data object and a field for referencing to the next node

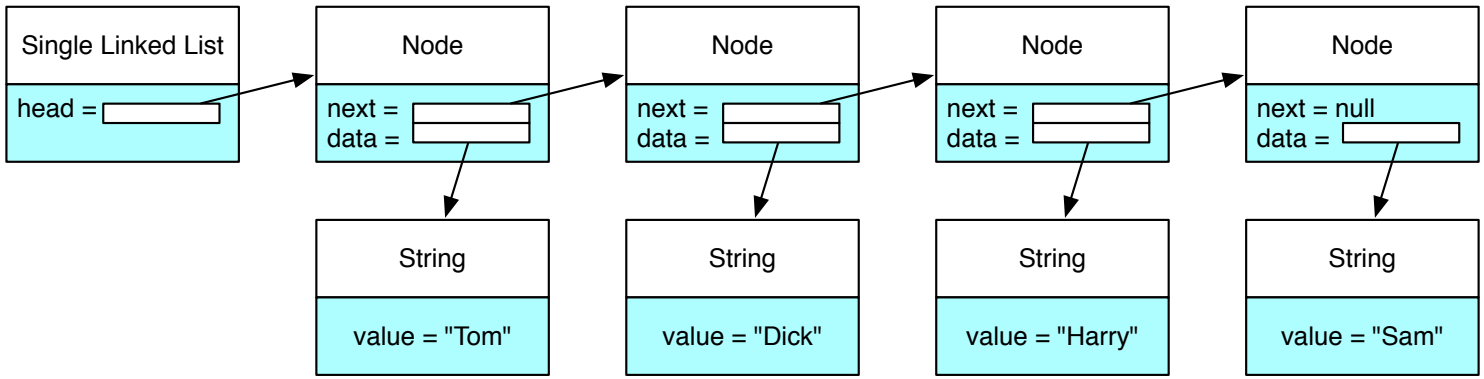


An Inner Node Class

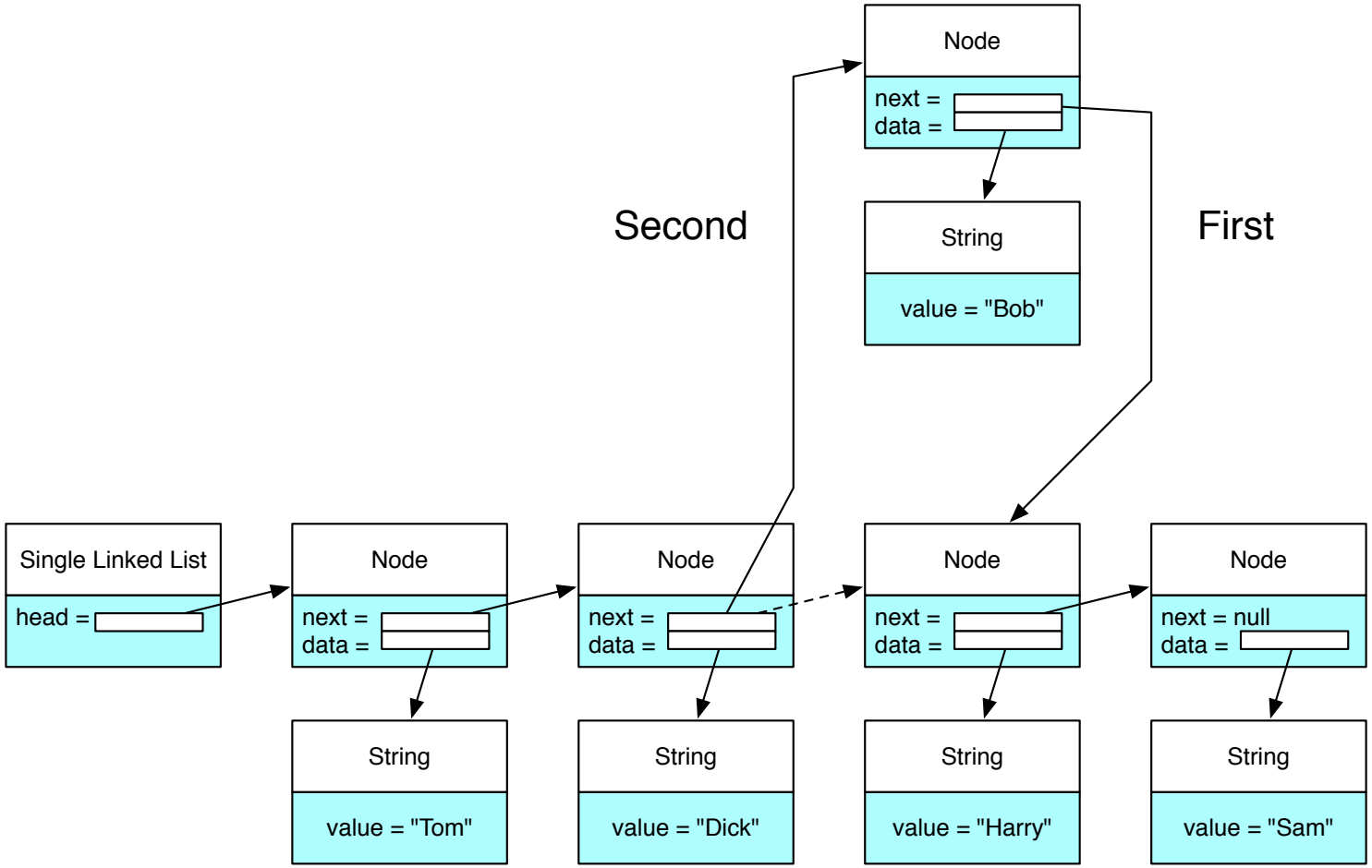
```
public MOList<E> {
   public class Node<E> {
      E data;
      Node<E> next;
      Node(E item) {
         /* constructor */
      }
      /* Other methods */
   }
   Node<E> head;
   int size;
   MOList() {
      /* constructor */
   }
   /* Other methods */
}
```

Following the “next” Reference to Traverse Data Objects

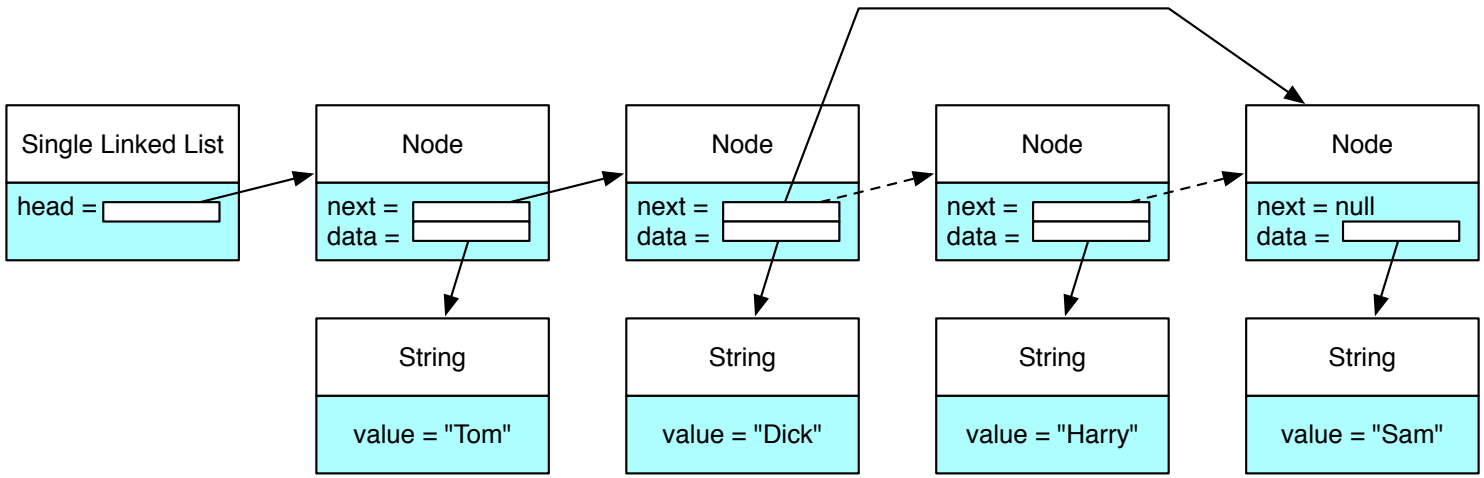
If x is a node object, then x.next is the next object of x, and x.next.next is the next, next object of x.



Insertion


Node
next =
data =
Second
String
value = "Bob"
First
Single Linked List
head =
Node
next =
data =
Node
next =
data =
Node
next =
data =
Node
next = null
data =
String
value = "Tom"
String
value = "Dick"
String
value = "Harry"
String
value = "Sam"

Removal


Single Linked List
head =
Node
next =
data =
Node
next =
data =
Node
next =
data =
Node
next = null
data =
String
value = "Tom"
String
value = "Dick"
String
value = "Harry"
String
value = "Sam"

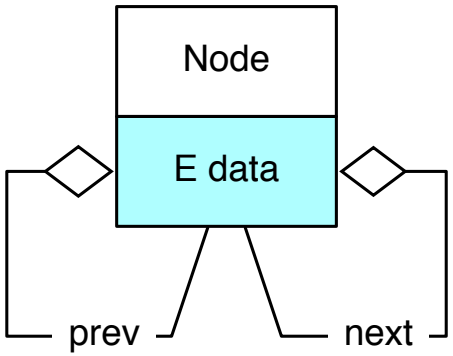
Double-Linked List

A double-linked list is a list that overcomes some of the limitations of a single-linked list:

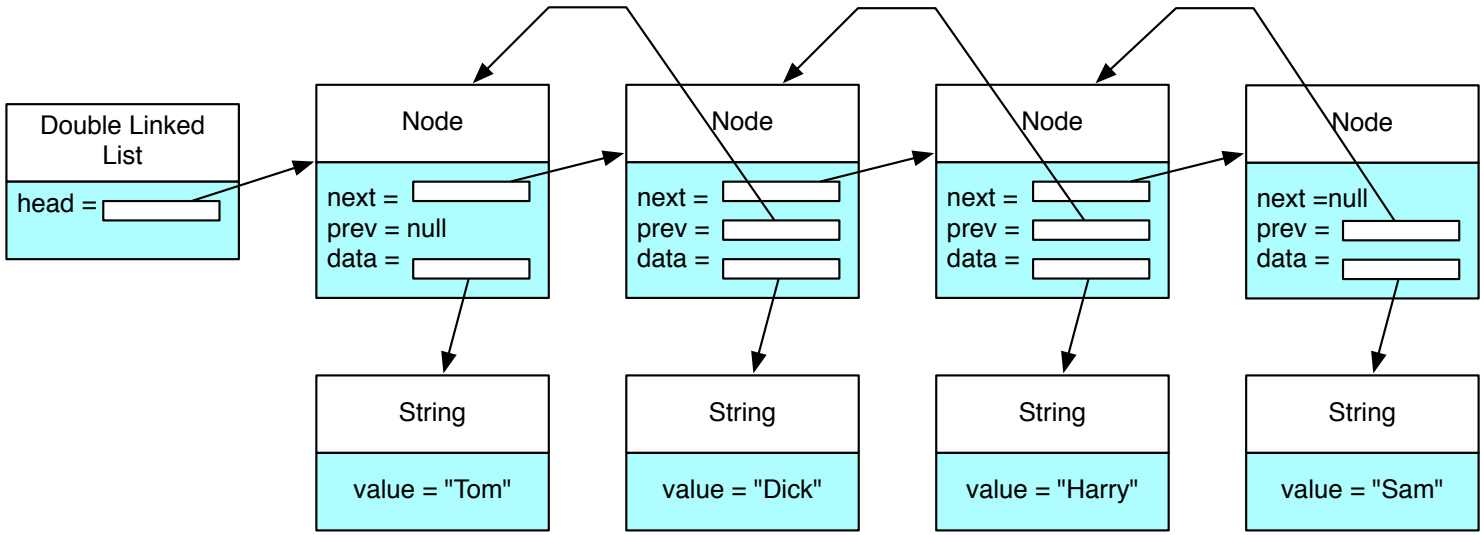
- Easy to insert a node after a referenced node, but hard to insert a node before a referenced node
- Can remove a node only if a reference to the predecessor is available
- Can traverse the list only in the forward direction

Double-linked Node

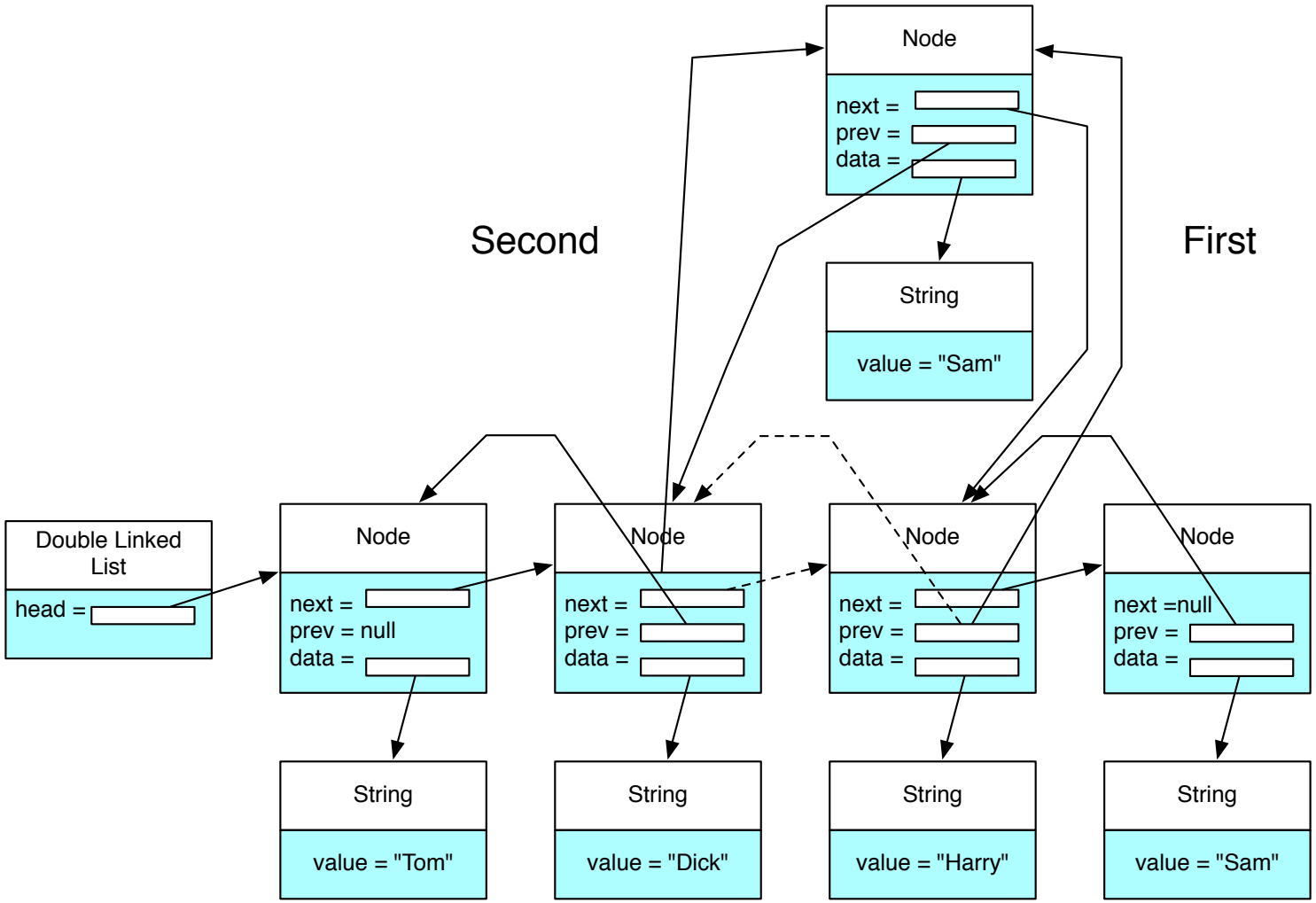
The basic unit of information storage is a node with two links, one pointing to the next node and the other pointing to the previous node



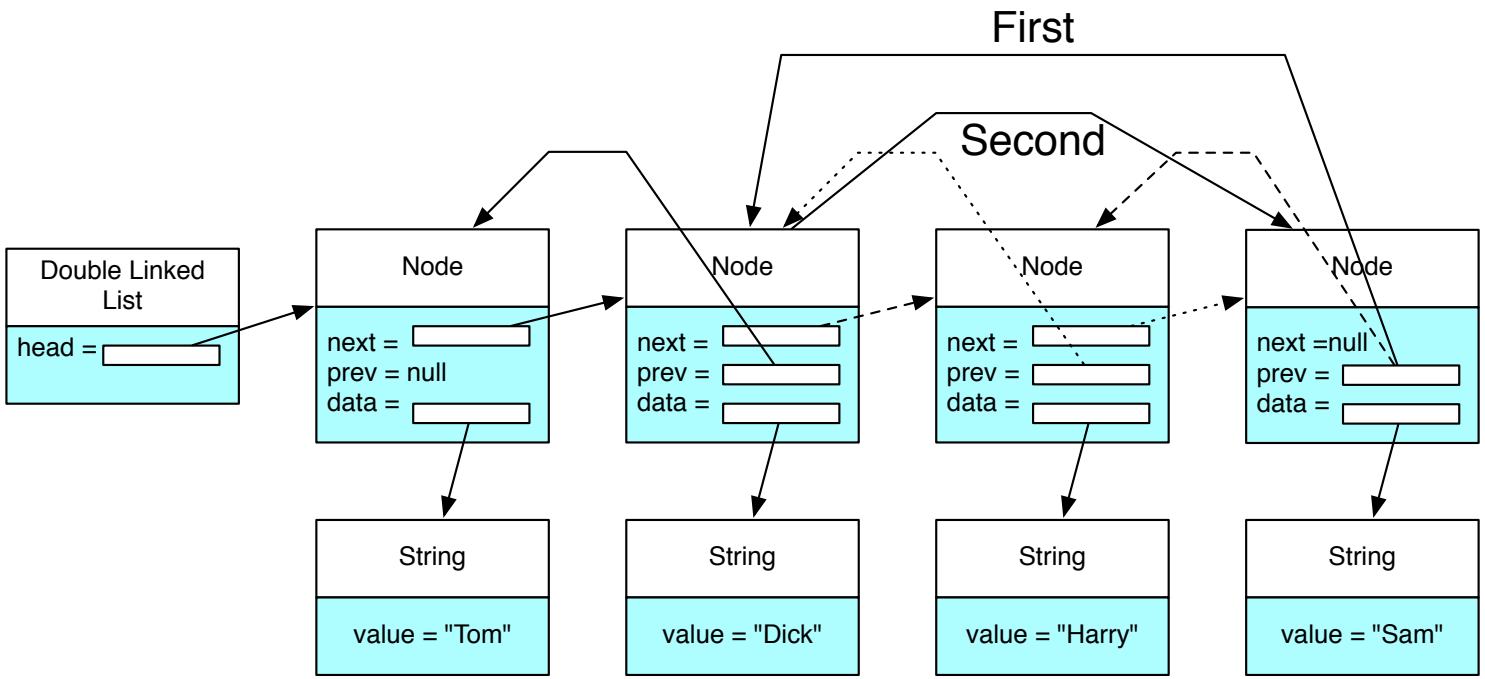
Double Linked List Example



Insertion


Node
next =
prev =
data =
Second
First
String
value = "Sam"
Double Linked List
head =
Node
next =
prev = null
data =
Node
next =
prev =
data =
Node
next =
prev =
data =
Node
next =null
prev =
data =
String
value = "Tom"
String
value = "Dick"
String
value = "Harry"
String
value = "Sam"

Removal

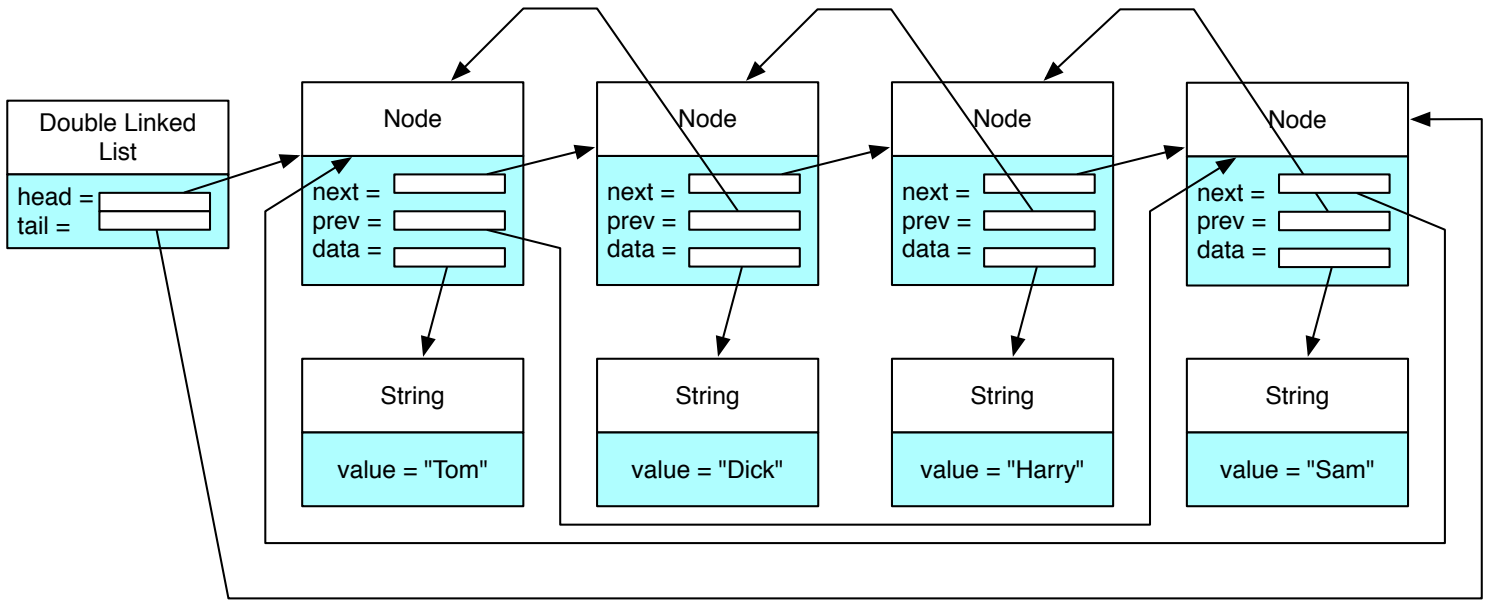

First
Second
Double Linked List
head =
Node
next =
prev = null
data =
Node
next =
prev =
data =
Node
next =
prev =
data =
Node
next =null
prev =
data =
String
value = "Tom"
String
value = "Dick"
String
value = "Harry"
String
value = "Sam"

Circular List

This is a double-linked list with:

- the “prev” data field of head pointing to the tail;
- the “next” data field of tail pointing to the head

Need to be careful with that for every node both “prev” and “tail” are defined



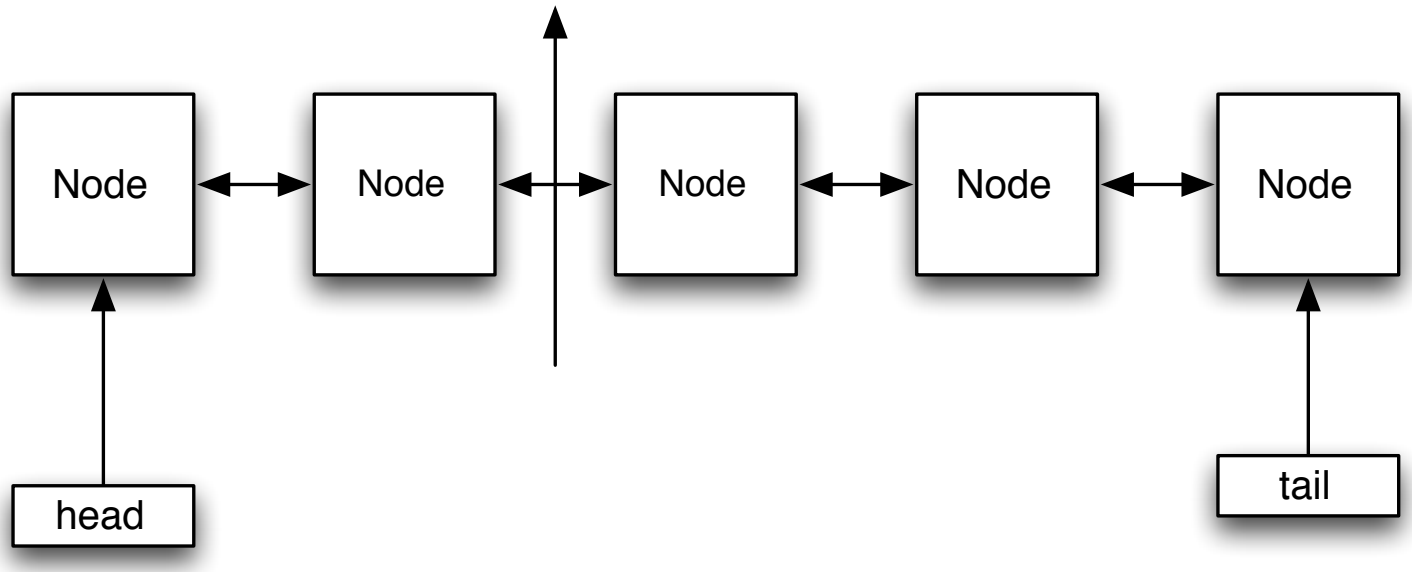
Iterator

`Iterator<E>` is an interface that enables sequential scanning of objects of type *E*
Three methods are required:

- `boolean hasNext()`: answers whether there is an item to be returned;
- `E next()`: returns an item and prepares to return the subsequent item;
- `void remove()`: removes the item that has just been returned; produces an error if there is no such element

Pictorial description

In some sense, the reference to the node containing the “next” data object sits between that node and the previous one



How to Implement Iterator

- A class that implements `Iterable<E>` must to have a method `iterator()` that returns an object that implements `Iterator<E>`
- If a class `MyClass` implements `Iterable<E>`, then the following code enables execution with a sequential scanning of the data in `MyClass`

```
for (E foo : MyClass) {
    /* loop body */;
}
```

ListIterator

Iterator with more methods, in particular, with backward moves

- `boolean hasPrevious()`: returns whether there is a previous element
- `E previous()`: returns the previous element
- `void add(E obj)`: inserts the data obj immediately before the data to be returned by `next()`
- `void set(E obj)`: replaces the last returned data with obj
- `int nextIndex()`: returns the index of the item to be returned by `next()`
- `int previoustIndex()`: returns the index of the item to be returned by `previous()`