BINUS UNIVERSITY

GREATER JAKARTA • BANDUNG • MALANG

People
Innovation
Excellence

COMP6048001
Data Structures

Linked List and Java Collections Framework Design
Week 4

Maria Seraphina Astriani
seraphina@binus.ac.id

Session Learning Outcomes

Upon successful completion of this course, students are expected to be able to:

- LO 1. Describe the use of various data structures
- LO 2. Apply appropriate operations for maintaining common data structures
- LO 3. Apply appropriate data structures and simple algorithms for solving computing problems
- LO 5. Explain the efficiency of some basic algorithms

Topics

- Linked List
- Java Collections Framework Design

Reminder

- Final project requirements (Newbinusmaya > session 1)
  - Please form your group (max. 3 students/group)
  - Start discussing with your teammates
    - Determine the topic
    - Use min. 2 data structures

Linked List

`ArrayList` and Linked List

- The `ArrayList` has the limitation that the add and remove methods operate in linear (O(n)) time because they require a loop to shift elements in the underlying array

FIGURE 2.8
Making Room to Insert an Item into an Array

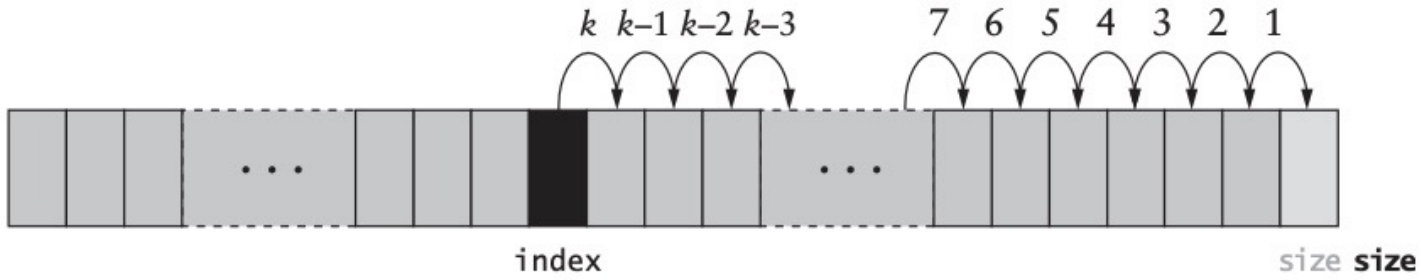
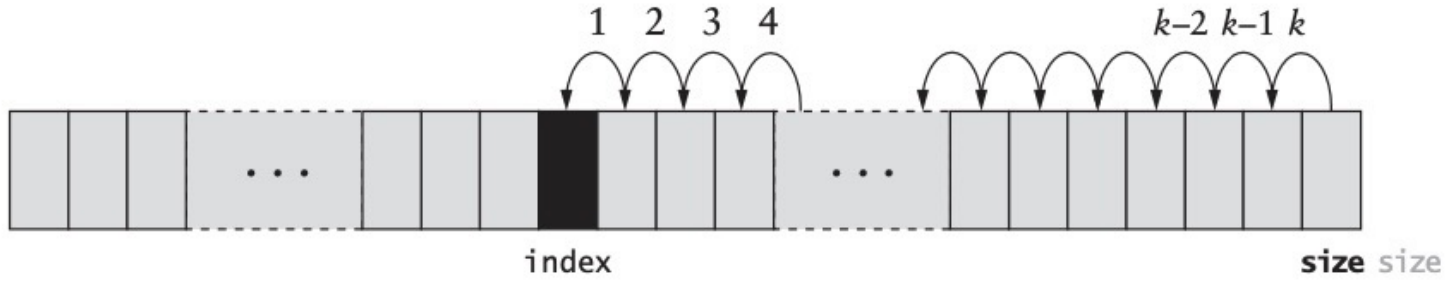


FIGURE 2.9
Removing an Item from an Array



Linked List

- A data structure, *linked list*, able to overcomes this limitation by providing the ability to add or remove items anywhere in the list in constant (O(1)) time.

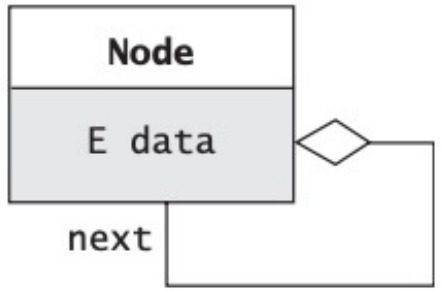
- A linked list is useful when you need to insert and remove elements at arbitrary locations (not just at the end) and when you do frequent insertions and removals.

A List Node

- A node is a data structure that contains a data item and one or more links. A link is a reference to a node.

- A UML (Unified Modeling Language) diagram of this relationship is shown in the following figure

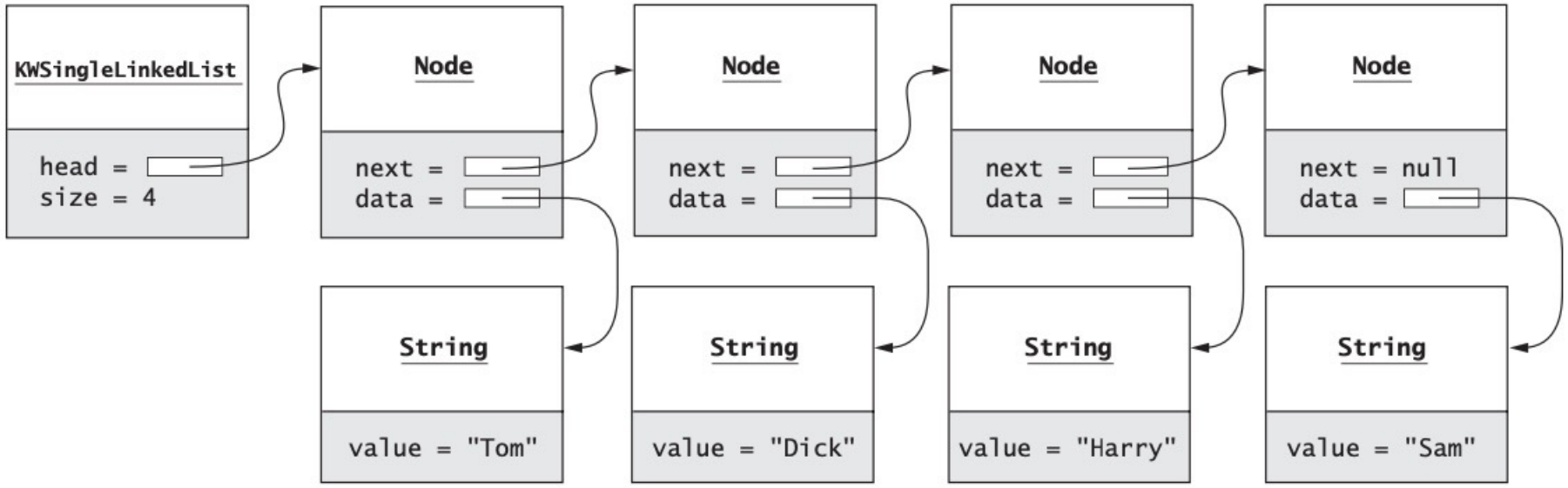
Node and Link



A List Node

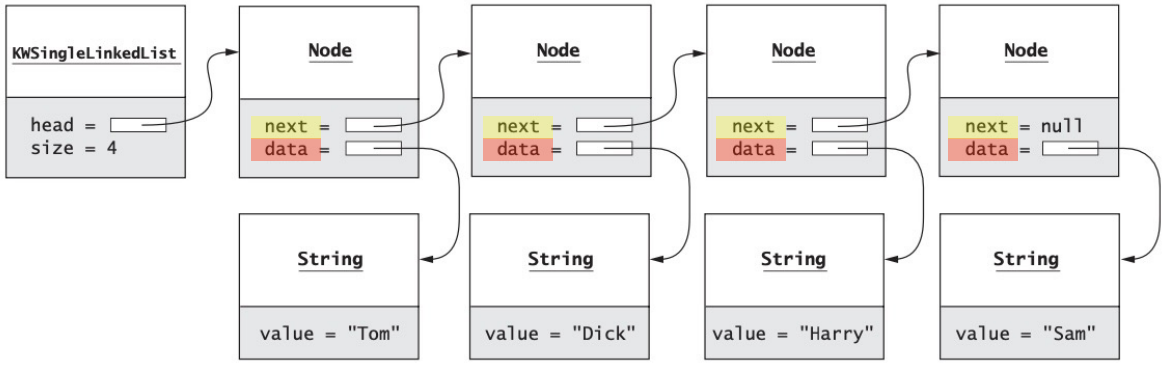
- Figure 2.16 shows four nodes linked together to form the list "Tom" ==> "Dick" ==> "Harry" ==> "Sam".
- In this figure, we show that data references a String object. In subsequent figures, we will show the string value inside the Node

FIGURE 2.16
Nodes in a Linked List



A List Node

FIGURE 2.16
Nodes in a Linked List



An Inner Class Node

```
/** A Node is the building block for a single-linked list. */
private static class Node<E> {
    // Data Fields
    /** The reference to the data. */
    private E data;
    /** The reference to the next node. */
    private Node<E> next;

    // Constructors
    /** Creates a new node with a null next field.
        @param dataItem The data stored
     */
    private Node(E dataItem) {
        data = dataItem;
        next = null;
    }

    /** Creates a new node that references another node.
        @param dataItem The data stored
        @param nodeRef The node referenced by new node
     */
    private Node(E dataItem, Node<E> nodeRef) {
        data = dataItem;
        next = nodeRef;
    }
}
```

Connecting Nodes

- We can construct the list shown in Figure 2.16 using the following sequence of statements:

```
Node<String> tom = new Node<>("Tom");
Node<String> dick = new Node<>("Dick");
Node<String> harry = new Node<>("Harry");
Node<String> sam = new Node<>("Sam");
tom.next = dick;
dick.next = harry;
harry.next = sam;
```

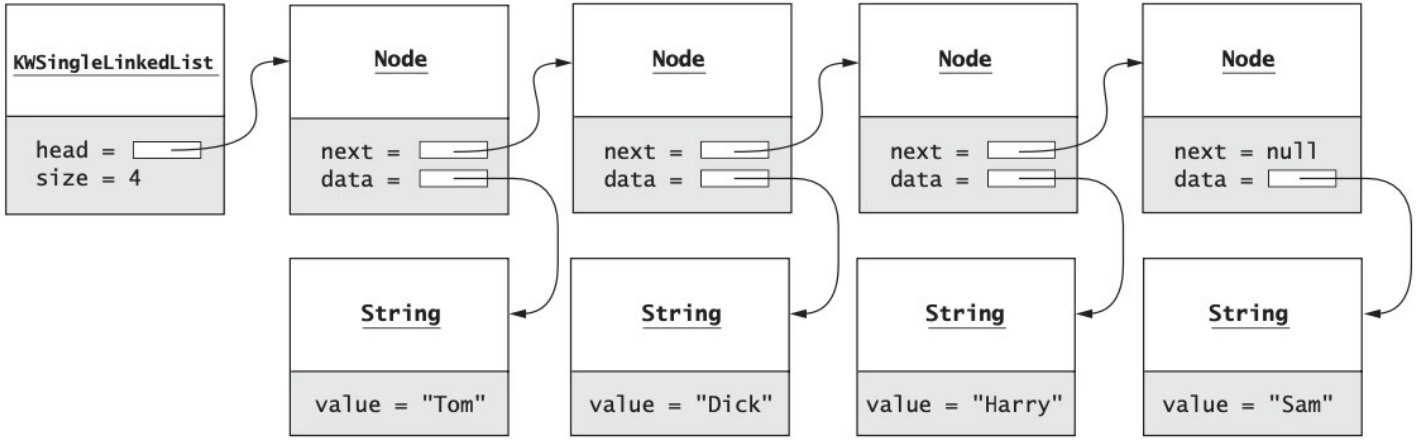
- The assignment statement

```
tom.next = dick;
```

stores a reference (link) to the node with data "Dick" in the variable next of node tom.

Hands-on Tutorial

- Let's create a linked list by using "*manual*" way (manually set the next node and manually printing the data WITHOUT using methods)




```
Tom
Dick
Harry
Sam

Other way to print by using head:
Tom
Dick
Harry
Sam
```

- Create list by using nodes: Tom -> Dick -> Harry -> Sam
- Please do not forget to create the head, and point it to Tom (the first node)
- Print (by using first node "Tom" and by using head)

Code: SimpleListNode.java

Exercise

Continue SimpleListNode.java:

a. Insert "Bill" before "Tom"

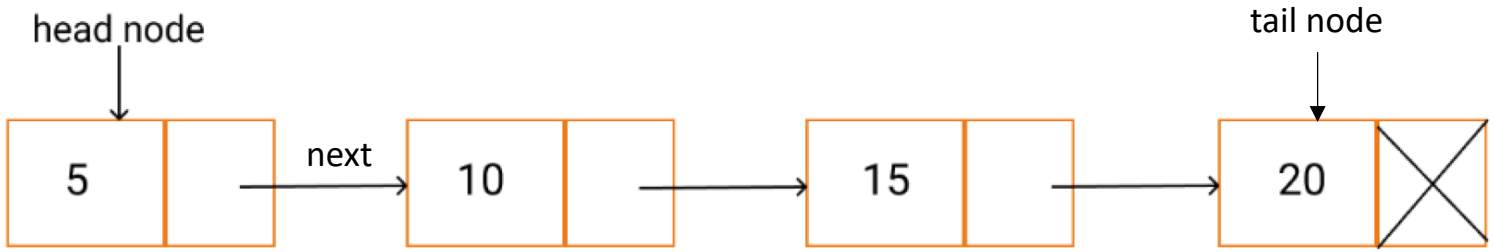
b. Insert "Sue" before "Sam"

```
Exercise a:
Bill
Tom
Dick
Harry
Sam

Exercise b:
Bill
Tom
Dick
Harry
Sue
Sam
```

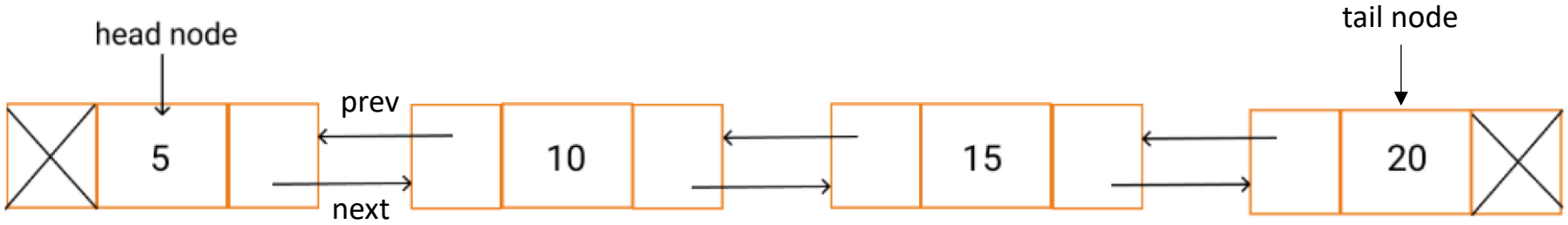
Code: SimpleListNode.java

Recall: Types of Linked List

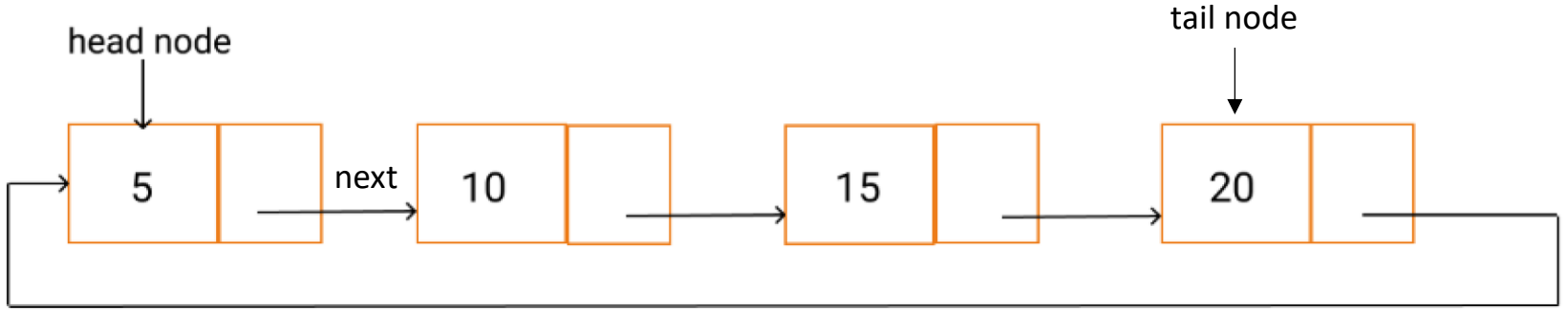
Singly Linked List
(Single-Linked List)



Doubly Linked List
(Double-Linked List)



Circular Linked List

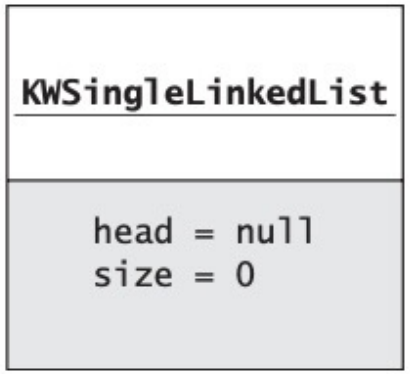


Node: a record in a linked list that contains a data field and a reference, a self-referential structure.
next pointer: the field of a node that contains a reference to the next node.
prev pointer: the field of the node that contains a reference to the previous node.
Head Node: the first node of the linked list.
Tail Node: the last node of the linked list.

https://devopedia.org/linked-list-data-structure

A Single-Linked List Class

- Java **does not have** a class that implements **single-linked lists**.
- However, we will create a "`KWSingleLinkedList`" class to show you how these operations could be implemented.




```
/** Class to represent a linked list with a link from each node to the next
    node. SingleLinkedList does not implement the List interface.
 */
public class KWSingleLinkedList<E> {
    /** Reference to list head. */
    private Node<E> head = null;
    /** The number of items in the list */
    private int size = 0;
    ...
```

- The data field head will reference the first list node called the list *head*.

Hands-on: Linked List Tutorial (by using methods)

- Many variations/styles to code linked list
  - Scratch - using Class (1st tutorial) – we will create some methods for insert and print the data
  - Using API (2nd tutorial)

1st tutorial (make all of the code from scratch - using Class)

```
Singly-linked list = A b c d e
```

- Create linked list (**`Node`** class)
  - **`data`** (String)
  - **`next`** (Node)
- Represent `head` and `tail` nodes (set the default as `null`)
- `addNode` method = insert the data
- `printLinkedList` method (need the help from `current` node as a "pointer"/"cursor")

Node = Whatever object you create, it can store text, number characters and code (method pointers). And it can store references to other objects, which again have text, number, characters and code of their own.
https://stackoverflow.com/questions/71503678/what-type-of-variable-type-has-node-in-java

You may rename the Main.java to SinglyLinkedList.java by using "refactor"

Code: SinglyLinkedList.java

A Single-Linked List Class

- **Method `addFirst` below inserts one element at a time** to the front of the list, thereby changing the node pointed to by head.
- In the call to the constructor for Node, the argument head references the current first list node.
- A new node is created, which is referenced by head and is linked to the previous list head.
- Variable data of the new list head references `item`.

```
/** Add an item to the front of the list.
    @param item The item to be added
 */
public void addFirst(E item) {
    head = new Node<>(item, head);
    size++;
}
```

A Single-Linked List Class

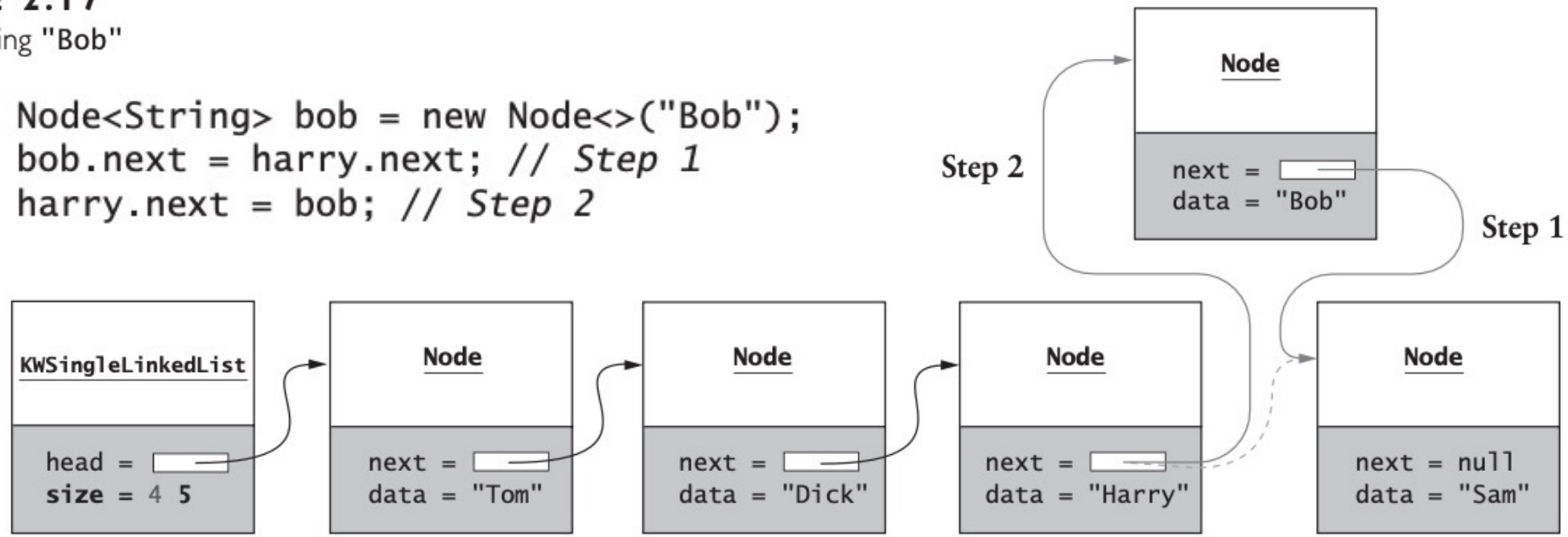
- The following fragment creates a linked list names and builds the list shown in Figure 2.16 using method `addFirst`:

```
KWSingleLinkedList<String> names = new KWSingleLinkedList<>();
names.addFirst("Sam");
names.addFirst("Harry");
names.addFirst("Dick");
names.addFirst("Tom");
```

Inserting a Node in a List

FIGURE 2.17
After Inserting "Bob"

```
Node<String> bob = new Node<>("Bob");
bob.next = harry.next; // Step 1
harry.next = bob; // Step 2
```

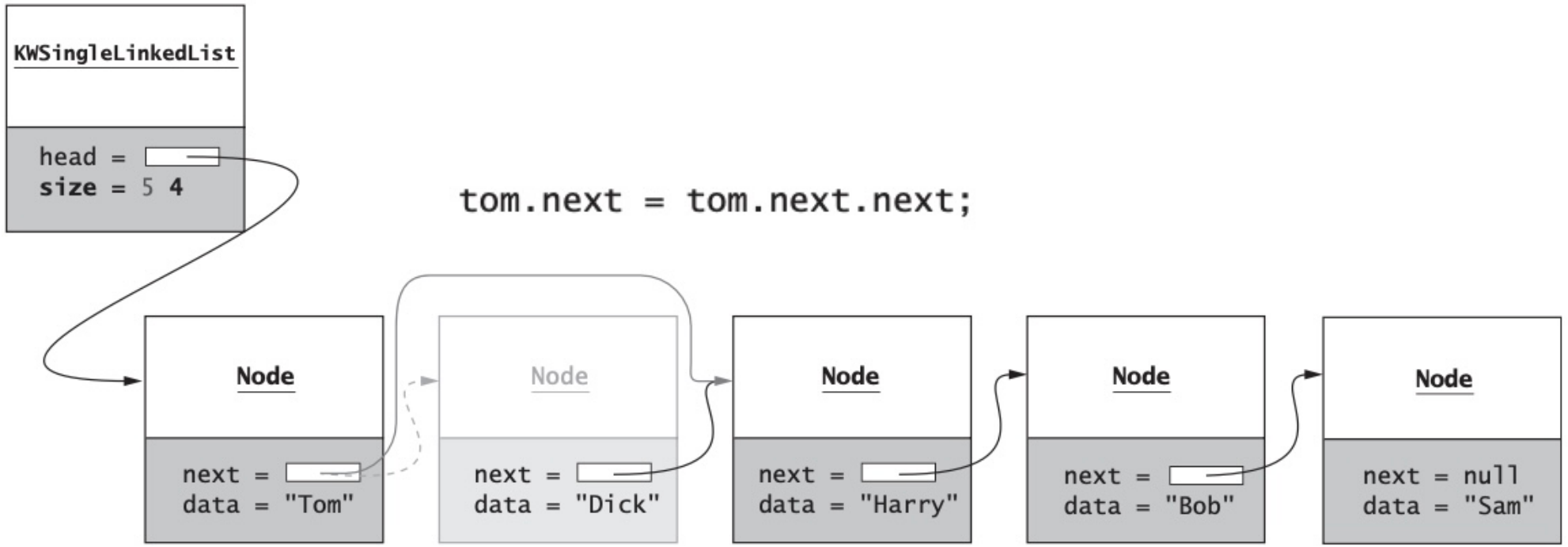


We can generalize this by writing the method addAfter as follows:

```
/** Add a node after a given node
    @param node The node preceding the new item
    @param item The item to insert
 */
private void addAfter(Node<E> node, E item) {
    node.next = new Node<>(item, node.next);
    size++;
}
```

Removing a Node

FIGURE 2.18
After Removing "Dick"



Again, we can generalize this by writing the removeAfter method:

```
/** Remove the node after a given node
    @param node The node before the one to be removed
    @return The data from the removed node, or null
            if there is no node to remove
 */
```

```
private E removeAfter(Node<E> node) {
    Node<E> temp = node.next;
    if (temp != null) {
        node.next = temp.next;
        size--;
        return temp.data;
    } else {
        return null;
    }
}
```

Removing a Node

- The `removeAfter` method works on all nodes except for the first one. For that we need a special method, `removeFirst`:

```
/** Remove the first node from the list
    @return The removed node's data or null if the list is empty
 */
private E removeFirst() {
    Node<E> temp = head;
    if (head != null) {
        head = head.next;
    }
    // Return data at old head or null if list is empty
    if (temp != null) {
        size--;
        return temp.data;
    } else {
        return null;
    }
}
```

Exercise (remove/delete manually)

Continue SimpleListNode.java:

c. Remove "Bill"

d. Remove "Sam"

e. Remove "Harry"

```
Exercise c:
Tom
Dick
Harry
Sue
Sam

Exercise d:
Tom
Dick
Harry
Sue

Exercise e:
Tom
Dick
Sue
```

Code: SimpleListNode.java

Activities

```
Singly-linked list = A b c d e
*Delete A*
Singly-linked list = b c d e
*Delete d*
Singly-linked list = b c e
*Delete E*
Data not found
Singly-linked list = b c e
```

1. Create `deleteNode` from 1st tutorial code (SinglyLinkedList.java) – using method

2. What is the big-O for the single-linked list get operation?

3. Draw a single-linked list of Integer objects containing the integers 5, 10, 7, and 30 and referenced by `head`. Complete the following fragment, which adds all Integer objects in a list. Your fragment should walk down the list, adding all integer values to sum.

```
int sum = 0;
Node<Integer> nodeRef = _____;
while (nodeRef != null) {
    int next = _____;
    sum += next;
    nodeRef = _____;
}
```

Code: SinglyLinkedList.java

Double-Linked Lists

FIGURE 2.19
Double-Linked List Node UML Diagram

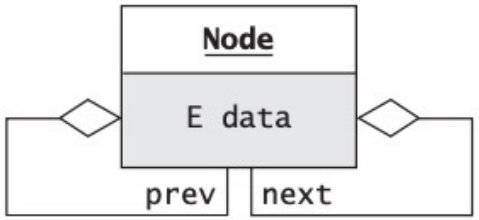
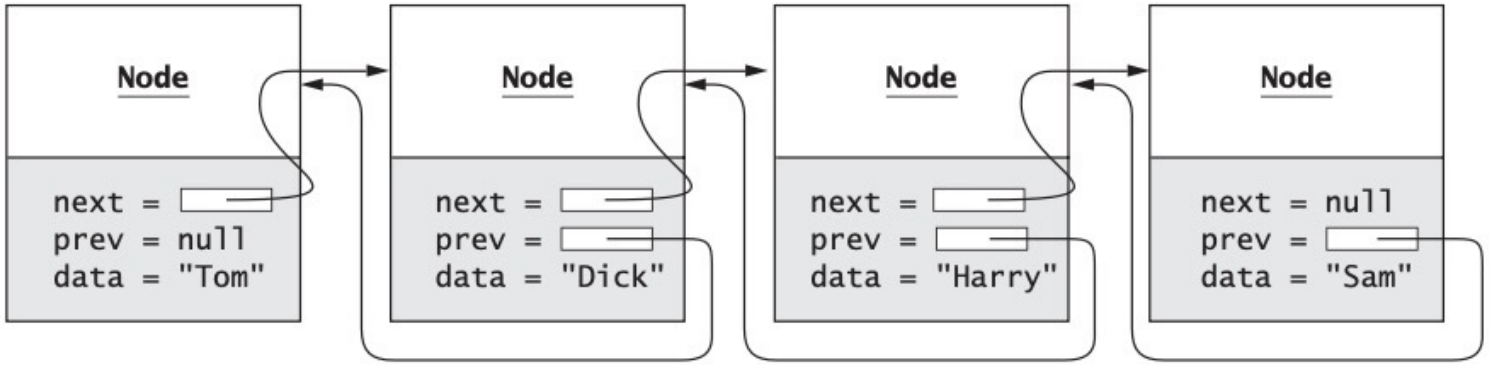


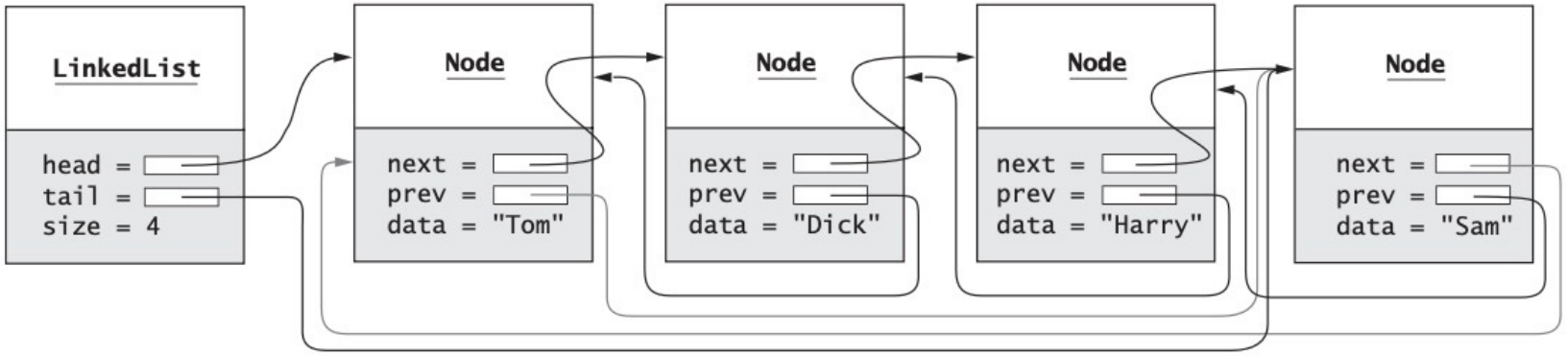
FIGURE 2.20
A Double-Linked List



Circular Lists

FIGURE 2.24
A Double-Linked List Object

FIGURE 2.25
A Circular Linked List



The `LinkedList` Class

- The `LinkedList` class, part of the Java API package `java.util`, is a double-linked list that implements the List interface.
- A selected subset of the methods from this Java API is shown in the following table.

Selected Methods of the `java.util.LinkedList<E>` Class

Method	Behavior
`public void add(int index, E obj)`	Inserts object `obj` into the list at position `index`
`public void addFirst(E obj)`	Inserts object `obj` as the first element of the list
`public void addLast(E obj)`	Adds object `obj` to the end of the list
`public E get(int index)`	Returns the item at position `index`
`public E getFirst()`	Gets the first element in the list. Throws `NoSuchElementException` if the list is empty
`public E getLast()`	Gets the last element in the list. Throws `NoSuchElementException` if the list is empty
`public boolean remove(E obj)`	Removes the first occurrence of object `obj` from the list. Returns `true` if the list contained object `obj`; otherwise, returns `false`
`public int size()`	Returns the number of objects contained in the list

Because the LinkedList class, like the `ArrayList` class, implements the List interface, it contains many of the methods found in the `ArrayList` class as well as some additional methods.

`LinkedList` provides several methods to do certain operations more efficiently

Linked List Tutorial

- **2nd tutorial**
  - Linked List Implementation by using Java API

```
[A, b, c, d, e]
*Print linked list using for =
A
b
c
d
e
* Add at index 2 =
[A, b, Hello World, c, d, e]
* Delete b =
[A, Hello World, c, d, e]
```

Code: LinkedListAPI.java

Linked List Tutorial

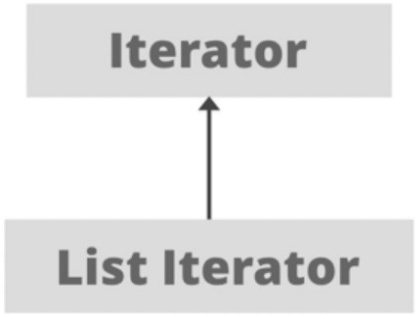

```
import java.util.LinkedList;
```

```
LinkedList<String> LL = new LinkedList<String>();
LL.add("A");
LL.add("b");
LL.add("c");
LL.add("d");
LL.add("e");
System.out.println(LL);

// print by using for loop
System.out.println("*Print linked list using for = ");
for(String s: LL)
{
    System.out.println(s + " ");
}
```

```
// add at a particular position
System.out.println("* Add at index 2 =");
LL.add(2, "Hello World");
System.out.println(LL);

// delete
System.out.println("* Delete b = ");
LL.remove("b");
System.out.println(LL);
```

```
[A, b, c, d, e]
*Print linked list using for =
A
b
c
d
e
* Add at index 2 =
[A, b, Hello World, c, d, e]
* Delete b =
[A, Hello World, c, d, e]
```

The `Iterator` and `ListIterator` in Java



https://www.geeksforgeeks.org/listiterator-in-java/

The `Iterator`

- Think of an *iterator* as a moving place marker that keeps track of the current position in a particular linked list.
- The Iterator object for a list starts at the first element in the list.
- The programmer can use the `Iterator` object's next method to retrieve the next element.
- Each time it does a retrieval, the Iterator object advances to the next list element, where it waits until it is needed again.
- We can also ask the Iterator object to determine whether the list has more elements left to process (method `hasNext`).
- Iterator objects throw a `NoSuchElementException` if they are asked to retrieve the next element after all elements have been processed.

The `Iterator`

- Assume `iter` is declared as an Iterator object for `LinkedList myList`.
- We can replace the fragment shown at the beginning of this section with the following.

```
// Access each list element.
while (iter.hasNext()) {
    E nextElement = iter.next();
    // Do something with the next element (nextElement).
    . . .
}
```

- This fragment is O(n) instead of $O(n^2)$. All that remains is to determine how to declare `iter` as an `Iterator` for `LinkedList` object `myList`.

The `Iterator` Interface

- The interface `Iterator<E>` is defined as part of API package `java.util`.
- The following table summarizes the methods declared by this interface.

The `java.util.Iterator<E>` Interface

Method	Behavior
`boolean hasNext()`	Returns `true` if the `next` method returns a value
`E next()`	Returns the next element. If there are no more elements, throws the `NoSuchElementException`
`void remove()`	Removes the last element returned by the `next` method

Iterator Tutorial

- Enhance the 2nd tutorial code (LinkedListAPI.java) and add iterator

```
[A, b, c, d, e]
*Print linked list using for =
A
b
c
d
e
* Add at index 2 =
[A, b, Hello World, c, d, e]
* Delete b =
[A, Hello World, c, d, e]
* Print 3rd data using iterator = c
* Print Linked List by using iterator =
A Hello World c d e
```

Code: LinkedListAPI.java

Iterator Tutorial

```
import java.util.Iterator;

// Use the iterator to print the 3rd data
Iterator<String> it = LL.iterator();
it.next();
it.next();
System.out.println("* Print 3rd data using iterator = " + it.next());

// Print all data in Linked List by using iterator
System.out.println("* Print Linked List by using iterator = ");
Iterator<String> itPrint = LL.iterator();
while(itPrint.hasNext())  // traversing elements
{
    System.out.print(itPrint.next() + " ");
}
```

```
[A, b, c, d, e]
*Print linked list using for =
A
b
c
d
e
* Add at index 2 =
[A, b, Hello World, c, d, e]
* Delete b =
[A, Hello World, c, d, e]
* Print 3rd data using iterator = c
* Print Linked List by using iterator =
A Hello World c d e
```

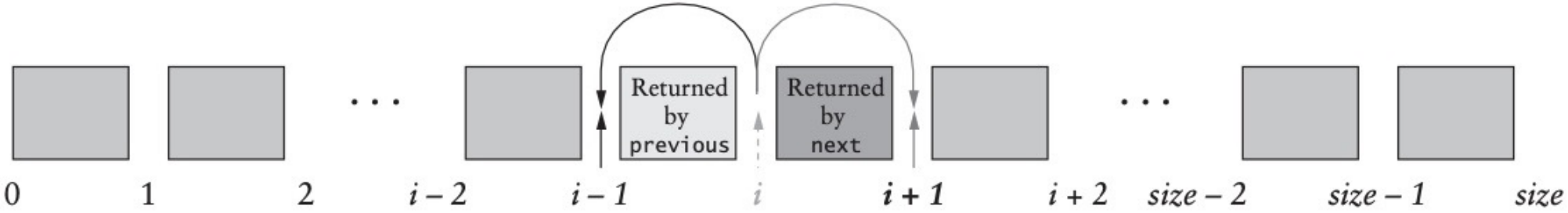
The `ListIterator` Interface

- The **Iterator has some limitations**.
- It can traverse the List only in the forward direction.
- It also **provides only a remove method**, not an add method.
- Also, to start an Iterator somewhere other than at first `List` element, you must write your own loop to advance the Iterator to the desired starting position.

There is **no current element** in `ListIterator`. Its cursor always lies between the previous and next elements.
The **`previous()`** will return to the previous elements and the **`next()`** will return to the next element.

https://www.geeksforgeeks.org/listiterator-in-java/

The `ListIterator`



The `ListIterator` Interface

- The methods defined by the `ListIterator` interface

TABLE 2.8
The `java.util.ListIterator<E>` Interface

Method	Behavior
`void add(E obj)`	Inserts object `obj` into the list just before the item that would be returned by the next call to method `next` and after the item that would have been returned by method `previous`. If the method `previous` is called after `add`, the newly inserted object will be returned
`boolean hasNext()`	Returns `true` if `next` will not throw an exception
`boolean hasPrevious()`	Returns `true` if `previous` will not throw an exception
`E next()`	Returns the next object and moves the iterator forward. If the iterator is at the end, the `NoSuchElementException` is thrown
`int nextIndex()`	Returns the index of the item that will be returned by the next call to `next`. If the iterator is at the end, the list size is returned
`E previous()`	Returns the previous object and moves the iterator backward. If the iterator is at the beginning of the list, the `NoSuchElementExcepton` is thrown
`int previousIndex()`	Returns the index of the item that will be returned by the next call to `previous`. If the iterator is at the beginning of the list, −1 is returned
`void remove()`	Removes the last item returned from a call to `next` or `previous`. If a call to `remove` is not preceded by a call to `next` or `previous`, the `IllegalStateException` is thrown
`void set(E obj)`	Replaces the last item returned from a call to `next` or `previous` with `obj`. If a call to `set` is not preceded by a call to `next` or `previous`, the `IllegalStateException` is thrown

The `ListIterator` Interface

- To obtain a `ListIterator`, you call the `listIterator` method of the `LinkedList` class.
- This method has two forms, as shown in Table 2.9.

TABLE 2.9
Methods in `java.util.LinkedList<E>` that Return `ListIterators`

Method	Behavior
`public ListIterator<E> listIterator()`	Returns a `ListIterator` that begins just before the first list element
`public ListIterator<E> listIterator(int index)`	Returns a `ListIterator` that begins just before the position `index`

ListIterator Tutorial
(Enhance from Iterator Tutorial)

```
import java.util.ListIterator;
System.out.println("");

// Print all data in Linked List by using ListIterator
System.out.println("* Print Linked List by using ListIterator = ");
ListIterator<String> lit = LL.listIterator();
while(lit.hasNext())  // traversing elements
{
    System.out.print(lit.next() + " ");
}

System.out.println("");

System.out.println("* Print backward = ");
while(lit.hasPrevious()) // traversing elements backward
{
    System.out.print(lit.previous() + " ");
}
```

```
* Print Linked List by using ListIterator =
A Hello World c d e
* Print backward =
e d c Hello World A
```

Code: LinkedListAPI.java

Iterator VS ListIterator Code

```
import java.util.Iterator;

LinkedList<String> LL = new LinkedList<String>();
LL.add("A");
LL.add("b");

Iterator<String> it = LL.iterator();

while(it.hasNext())
{
    System.out.print(it.next() + " ");
}
```

```
import java.util.ListIterator;

LinkedList<String> LL = new LinkedList<String>();
LL.add("A");
LL.add("b");

ListIterator<String> it = LL.listIterator();

while(it.hasNext())
{
    System.out.print(it.next() + " ");
}

while(it.hasPrevious())
{
    System.out.print(lit.previous() + " ");
}
```

Iterator VS ListIterator

Iterator	ListIterator
It can traverse a collection of any type (example: Map, List, and Set).	It traverses only list collection implemented classes like LinkedList, ArrayList, etc.
Traversal can only be done in forwarding direction.	Traversal of elements can be done in both forward and backward direction.
Iterator object can be created by calling iterator() method of the collection interface.	ListIterator object can be created by calling directions listIterator() method of the collection interface.
Deletion of elements is not allowed.	Deletion of elements is allowed.
It throws **ConcurrentModificationException** on doing addition operation. Hence, addition is not allowed.	Addition of elements is allowed.
In iterator, we can't access the index of the traversed element.	In listIterator, we have nextIndex() and previousIndex() methods for accessing the indexes of the traversed or the next traversing element.
Modification of any element is not allowed.	Modification is allowed.

https://www.geeksforgeeks.org/listiterator-in-java/

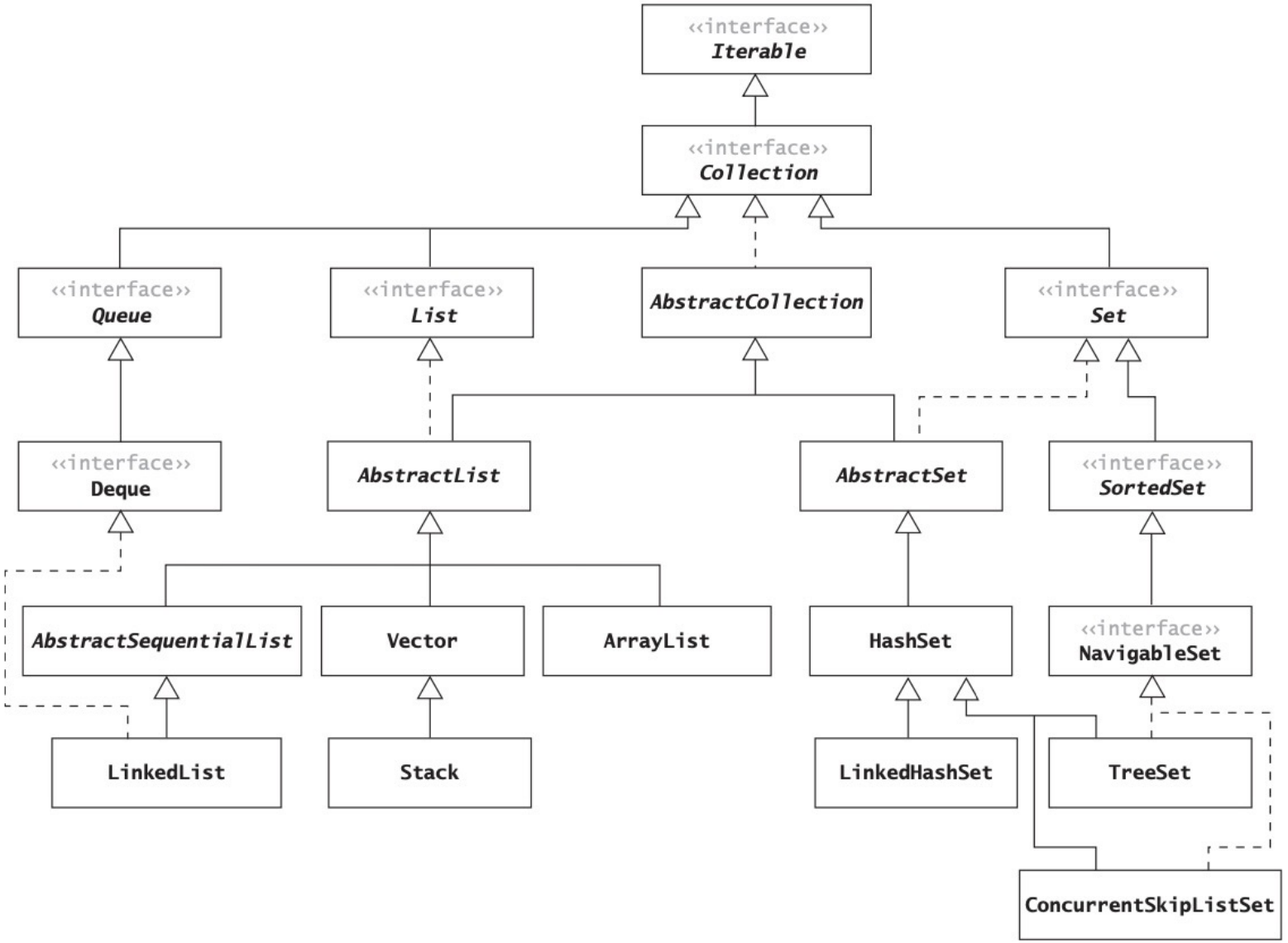
Framework

- A framework is a set of classes and interfaces which provide a ready-made architecture.
- In order to implement a new feature or a class, there is no need to define a framework.
- However, an optimal object-oriented design always includes a framework with a collection of classes such that all the classes perform the same kind of task.

https://www.geeksforgeeks.org/collections-in-java-2/

The Collections Framework Design

The Collections Framework



The `Collection` Interface

- The `Collection` interface is part of the Collections Framework.
- The Collection interface specifies a subset of the methods specified in the List interface.
- Specifically, the `add(int, E)`, `get(int)`, `remove(int)`, `set(int, E)`, and related methods (all of which have an int parameter that represents a position) are not in the Collection interface, but the add(E) and remove(Object) methods, which do not specify a position, are included.
- The iterator method is also included in the Collection interface.
- Thus, you can use an Iterator to access all of the items in a Collection, but the order in which they are retrieved is not necessarily related to the order in which they were inserted.

Common Features of Collections

- Because it is the superinterface of `List`, `Queue`, and `Set`, the `Collection` interface specifies a set of common methods.
- If you look at the documentation for the Java API `java.util.Collection`, you will see that this is a fairly large set of methods and other requirements.
- A few features can be considered fundamental:
  - Collections grow as needed.
  - Collections hold references to objects.
  - Collections have at least two constructors: one to create an empty collection and one to make a copy of another collection.

Common Features of Collections

- Table 2.13 shows selected methods defined in the `Collection` interface.

TABLE 2.13
Selected Methods of the `java.util.Collection<E>` Interface

Method	Behavior
`boolean add(E obj)`	Ensures that the collection contains the object `obj`. Returns `true` if the collection was odified
`boolean contains(E obj)`	Returns `true` if the collection contains the object `obj`
`Iterator<E> iterator()`	Returns an `Iterator` to the collection
`int size()`	Returns the size of the collection

The `AbstractCollection`, `AbstractList`, and `AbstractSequentialList` Classes

- If you look at the Java API documentation, you will see that the Collection and List interfaces specify a large number of methods.
- To help implement these interfaces, the Java API includes the `AbstractCollection` and `AbstractList` classes.
- You can think of these classes as a kit (or as a cake mix) that can be used to build implementations of their corresponding interface.
- Most of the methods are provided, but you need to add a few to make it complete.

The `List` and `RandomAccess` Interfaces (Advanced)

- The primary purpose of this interface is to allow generic algorithms to alter their behavior to provide good performance when applied to either random or sequential access lists.

(https://docs.oracle.com/javase/7/docs/api/java/util/RandomAccess.html#:~:text=Interface%20RandomAccess&text=The%20primary%20purpose%20of%20this,random%20or%20sequential%20access%20lists.)

- If a collection class implements RandomAccess interface then we can access any of its element with the same speed.
- RandomAccess interface is marker interface and it does not contains any methods.
- **`ArrayList`** and **`vector`** classes implements this interface.

https://www.allindiaexams.in/interview/interview-questions-and-answers/java/java-collections/discussion/5963

Comparison

Which one is faster?

Repeated access using `List.get( )`:

```
Object o;
for (int i=0, n=list.size(  ); i < n; i++)
  o = list.get(i);
```

Repeated access using `Iterator.next( )`:

```
Object o;
for (Iterator itr=list.iterator(  ); itr.hasNext(  ); )
  o = itr.next(  );
```

http://etutorials.org/Programming/Java+performance+tuning/Chapter+11.+Appropriate+Data+Structures+and+Algorithms/11.6+The+RandomAccess+Interface/

Combine Them!

- Combines the previous two loops to avoid the repeated Iterator.hasNext( ) test on each loop iteration:

```
Object o;
Iterator itr=list.iterator(  );
for (int i=0, n=list.size(  ); i < n; i++)
  o = itr.next(  );
```

http://etutorials.org/Programming/Java+performance+tuning/Chapter+11.+Appropriate+Data+Structures+and+Algorithms/11.6+The+RandomAccess+Interface/

The `List` and `RandomAccess` Interfaces (Advanced)

- The `RandomAccess` interface is applied only to those implementations in which indexed operations are efficient (e.g., `ArrayList`).
- An algorithm can then test to see if a parameter of type List is also of type `RandomAccess` and, if not, copy its contents into an `ArrayList` temporarily so that the indexed operations can proceed more efficiently.
- After the indexed operations are completed, the contents of the `ArrayList` are copied back to the original.

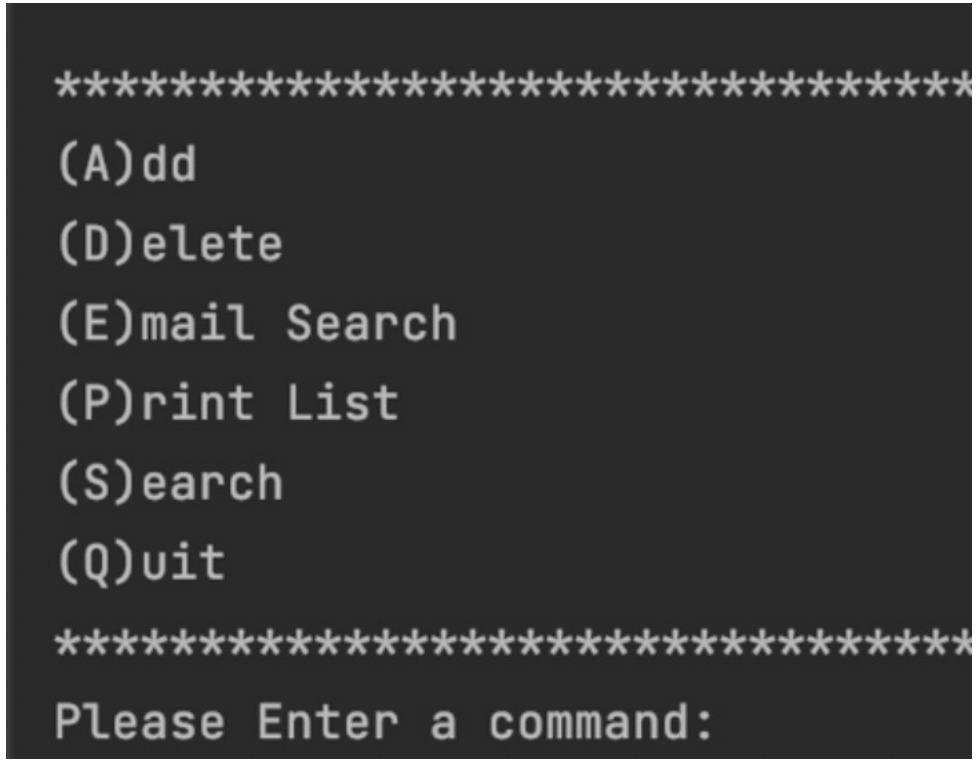
RandomAccess Example (Concept)

```
Object o;
if (listObject instanceof RandomAccess)
{
  for (int i=0, n=list.size(  ); i < n; i++)
  {
    o = list.get(i);
    //do something with object o
  }

}
else
{
  Iterator itr = list.iterator(  );
  for (int i=0, n=list.size(  ); i < n; i++)
  {
    o = itr.next(  );
    //do something with object o

  }
}
```

RandomAccess Example

```
Collection<Integer> c = new ArrayList<Integer>();

c.add(1);
c.add(2);
c.add(3);
Object obj;

if (c instanceof RandomAccess)
{
    for (int i=0; i<c.size(); i++)
    {
     obj = ((ArrayList<Integer>) c).get(i);
     System.out.println(obj);
    }
}
```

Week 4 Assignment: Contact Book Using Linked List



- Task: Create a contact book program that manage your friends contact (name, phone number, email)
- Input: The program prompts as shown in the figure
- Output: The results of the operations

- **Due: before week 5 class**
- Submission: https://forms.office.com/r/pq2Xzs2qdg

Week 4 Assignment: Contact Book Using Linked List

- Hint

```
switch (menu) {
    case "A":
        System.out.println("Add an entry");
        ……
        break;
    ……
        …
    default:
        System.out.println("Unknown entry");
}
```

References

- Koffman, E. B., & Wolfgang, P. A. (2021). Data structures: abstraction and design using Java (4th Edition). John Wiley & Sons. [DSA]
- Weiss, M. A. (2010). Data Structures and Problem Solving Using Java (Fourth Edition). Addison-Wesley. [DSPS]
- Weiss, M. A. (2014). Data structures and algorithm analysis in Java (3rd Ed). Pearson.
- Karumanchi, N. (2017). Data Structures and Algorithms Made Easy In JAVA. CareerMonk.
- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in Java (6th Ed.). John Wiley & Sons.

Thank you