GIUSEPPE TURINI

CS-102 COMPUTING AND ALGORITHMS 2
LESSON 03
LINKED LISTS

DATA ABSTRACTION AND PROBLEM SOLVING WITH JAVA, **CHAP 5, 241-311**.

HIGHLIGHTS

Preliminaries

Object References

- Reference Variables, and Operations on Reference Variables
- Arrays and References, Equality Between References, and Argument Passing

Resizable Arrays

Reference-Based (Linked) Lists

- Programming with Linked Lists
- Reference-Based ADT List Implementation
- Passing a Linked List to a Method, and Processing Linked Lists Recursively

Variations of Reference-Based (Linked) Lists

- Linked Lists with Tail References, and Circular Linked Lists
- Linked Lists with Dummy Head Nodes, and Doubly Linked Lists

The Java Collection Framework

- Generics, Iterators and Iterable Collections, and the JCF List Interface

PRELIMINARIES 1

DATA STRUCTURES FOR IMPLEMENTING AN ABSTRACT DATA TYPE (ADT)

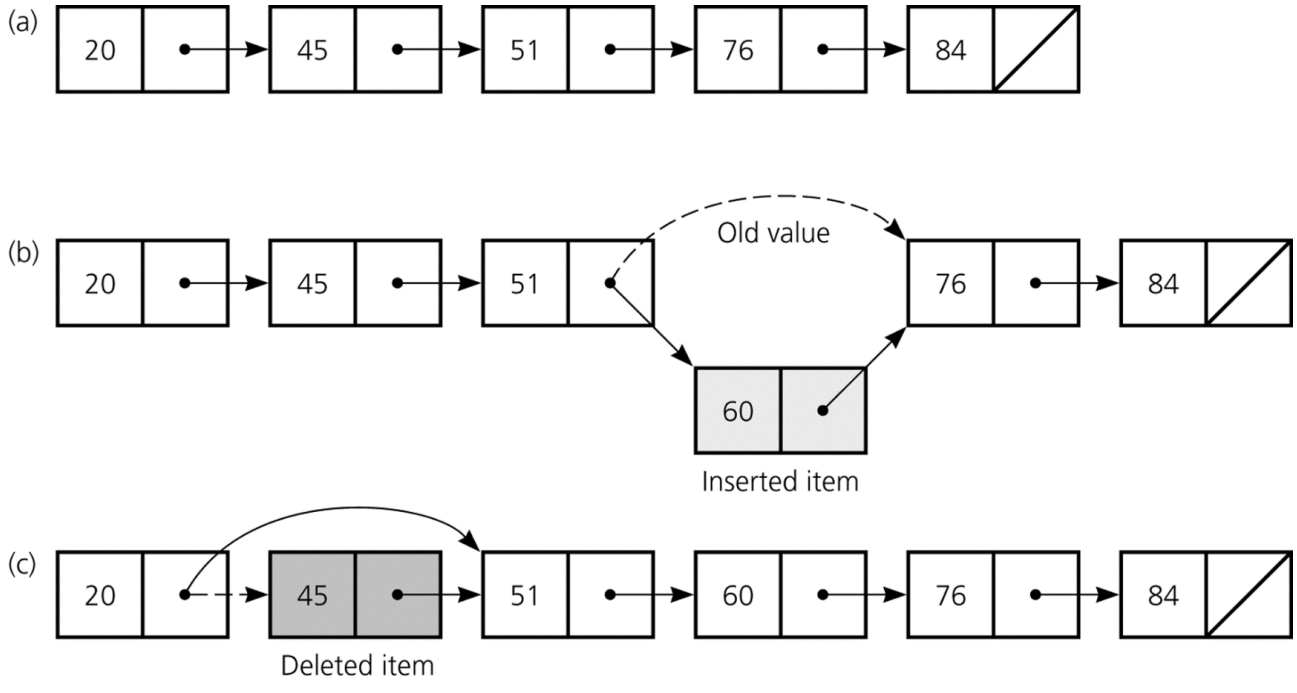
Array-based List: has a **fixed size**, and
the **data must be shifted** during insertions and deletions.

Reference-based List: aka **Linked List**,
is **able to grow in size** as needed, and
does not require to shift items for insertions/deletions.
See: en.wikipedia.org/wiki/linked_list
See: docs.oracle.com/javase/8/docs/api/java/util/linkedlist
See: visualgo.net/en/list

PRELIMINARIES 2

INSERTION (B) AND DELETION (C) IN A LINKED LIST OF INTEGERS (A)

See: visualgo.net/en/list



OBJECT REFERENCES 1

REFERENCE VARIABLES

A variable that refers to an object of a class, is actually a reference to that object. A **reference variable** stores the location (i.e. **memory address**) of an object.

```
Integer intRef; // Declaration of a reference variable intRef.
intRef = new Integer(5); // Instantiate an Integer obj and assign its ref to intRef.
```

When a reference variable is used as a data field of a class, its default value is **null**. But, if it is used as a local variable in a method, it does not have a default value.

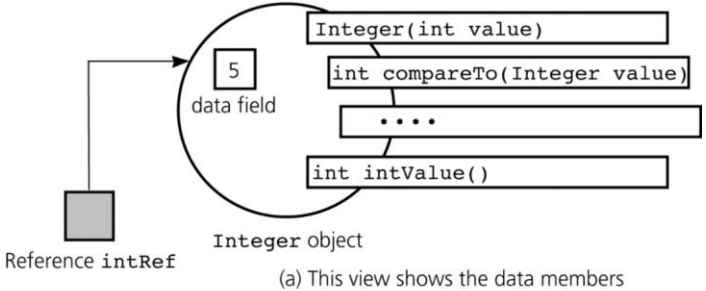
Note: An object of a class does not come into existence until you call one of the class constructors using the **new** operator.

OBJECT REFERENCES 2

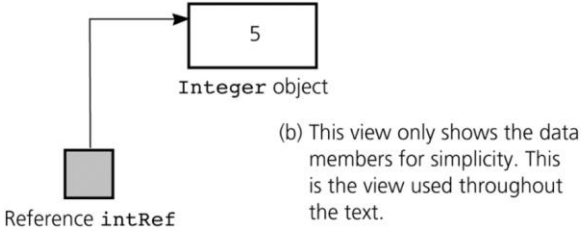
EXAMPLE OF A REFERENCE VARIABLE

```
// Declare a reference variable intRef,
// referencing an object of type Integer.
Integer intRef;

// Instantiate an Integer object, and
// assign its reference to intRef.
intRef = new Integer(5);
```



(a) This view shows the data members and methods for the object.



(b) This view only shows the data members for simplicity. This is the view used throughout the text.

Note: The primitive type **int** is different than the class **Integer**! In fact, the **Integer** class wraps a value of the primitive type **int** in an object.

See: docs.oracle.com/javase/8/docs/api/java/lang/integer
See: docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes

OBJECT REFERENCES 3

OPERATIONS ON REFERENCE VARIABLES A

If you try to use a reference variable that does not currently reference any object (i.e. with value **null**), the **exception java.lang.NullPointerException** will be thrown at runtime! Trying to use a reference variable not initialized will cause a compiler error!

When one reference variable is assigned to another reference variable, both references then refer to the same object

```
Integer p, q; // p and q do not reference any object.
p = new Integer(6); // p now references an Integer object of value 6.
q = p; // Now both p and q reference the same Integer object of value 6.
```

Note: If an object is not referenced by any variable is marked for garbage collection!

OBJECT REFERENCES 4

OPERATIONS ON REFERENCE VARIABLES B

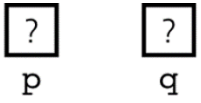
a. Declaration of reference variables **p** and **q**, does not initialize of their references.

b. Allocation of an **Integer** object (**5**), and storage of its reference into **p**.

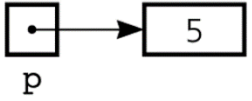
c. Allocation of another **Integer** object (**6**), and storage of its reference again into **p**. This overwrite **p**, dereferencing object (**5**) that is marked for garbage collection (**gray**).

d. Assign **p** to **q**, so now both reference the same **Integer** object (**6**).

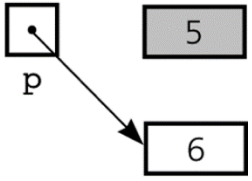
(a)
```
Integer p;
Integer q;
```



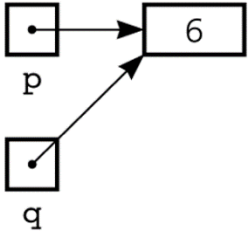
(b)
```
p = new Integer(5);
```



(c)
```
p = new Integer(6);
```



(d)
```
q = p;
```



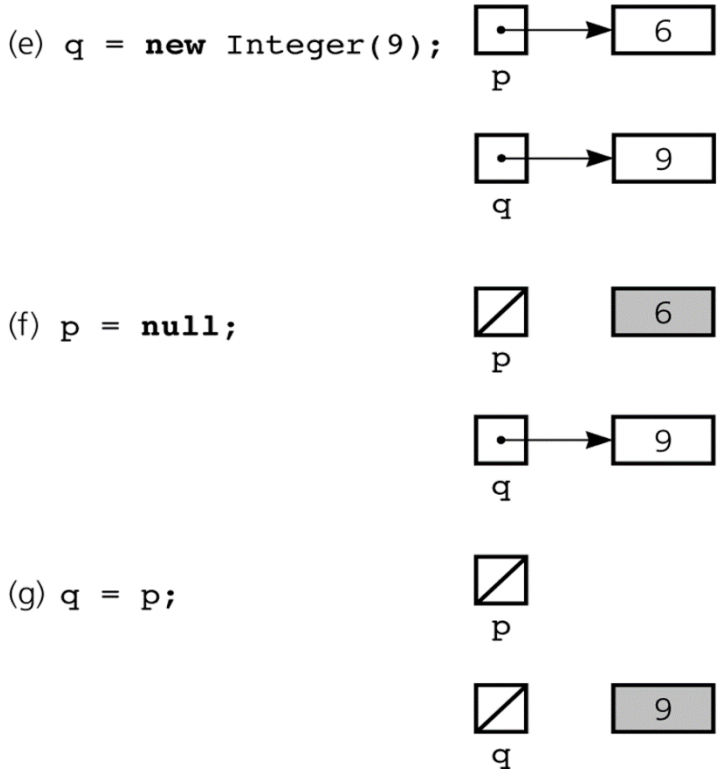
OBJECT REFERENCES 5

OPERATIONS ON REFERENCE VARIABLES C

e. Allocation of an **Integer** object (**9**), and storage of its reference into **q**. Since **p** still references the same object (**6**), that object is not marked for garbage collection.

f. Assignment of a **null** reference to **p**, this dereferences at the same time object (**6**) that is marked for garbage collection (**gray**).

g. Assign **p** to **q**, so now: both variables have a **null** value, and object (**9**) is dereferenced and so it is marked for garbage collection.



OBJECT REFERENCES 6

ARRAYS AND REFERENCES

An array of objects is actually an array of references to the objects.

```
Integer[] scores = new Integer[30]; // An array of 30 references to Integer objects.
```

Note: Remember that you need to instantiate objects for each array item!

```
scores[0] = new Integer(7); // Initialize reference stored into array element 0.
```

See: docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays
See: docs.oracle.com/javase/8/docs/api/java/lang/integer

OBJECT REFERENCES 7

EQUALITY BETWEEN REFERENCES

Equality operators **==** and **!=** compare the values of reference variables, not the values of the referenced objects.

To compare objects field by field, use the method **equals** (**java.lang.Object** class).

See: docs.oracle.com/javase/tutorial/java/nutsandbolts/op2
See: docs.oracle.com/javase/8/docs/api/java/lang/object

OBJECT REFERENCES 8

PASSING OBJECTS TO METHODS

Formal Parameter: A variable as found in the function definition.
Actual Argument: The actual input passed to a function.

When a method is called and has formal parameters that are reference variables, then **the reference value of the actual argument is copied to the formal parameter**.

```
void f( Integer fp ) { ... } // fp is a formal parameter of method f.

Integer aa = new Integer(7); // aa is an actual argument of the following call to f.
f( aa ); // Note: in this call to f both aa and fp reference the same Integer object (7)!


// Note: using the new operator with a formal parameter can produce unexpected results!
void f( Integer fp ) { fp = new Integer(9); ... } // Warning: unexpected results!
```

RESIZEABLE ARRAYS

An array has a fixed size, but we can overcome this limitation...

Resizeable Array: an array capable to grow and shrink at runtime. Obviously this is an illusion created by using an **allocate-and-copy** strategy with fixed-size arrays.

```
float[] newArray = new float[newCapacity]; // Create a new array using the new capacity.
// Copy the content of the original array to the new array.
for( int i = 0; i < myArray.length; i++ ) { newArray[i] = myArray[i]; }
myArray = newArray; // Change the reference to the original array to the new array.
```

Note: java.util.Vector and **java.util.ArrayList** implement similar resizeable arrays.

See: docs.oracle.com/javase/8/docs/api/java/util/vector
See: docs.oracle.com/javase/8/docs/api/java/util/arraylist

REFERENCE-BASED (LINKED) LISTS 1

LINKED LISTS

A linked list contains nodes that are linked to one another.
Each node of a linked list can be implemented by an object of type **Node** containing:

- the object data in the **item** field (a reference to an **Object** object), and
- a link to the next node (a reference to a **Node** object) in the **next** field.

```
package List; // Indicate that this class is part of the package List.
class Node {
   Object item; // Object data.
   Node next; // Reference to the next node.
   ... }
```

Note: The **Node** class is declared **package-private** to prevent package users to access data fields. The **Node** class is only used internally to the **List** package.

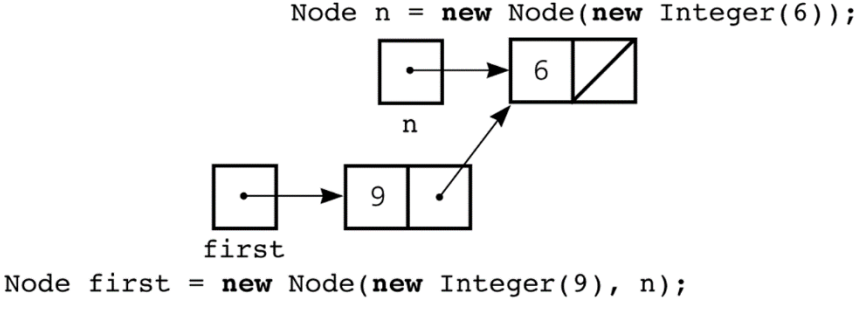
REFERENCE-BASED (LINKED) LISTS 2

OPERATIONS ON LINKED LIST NODES A

Using **Node** constructors to initialize nodes.

```
package List;
class Node {
   Object item;
   Node next;
   Node( Object o ) { item = o; next = null; } // Constructor 1.
   Node( Object o, Node n ) { item = o; next = n; } // Constructor 2.
   ... }
```

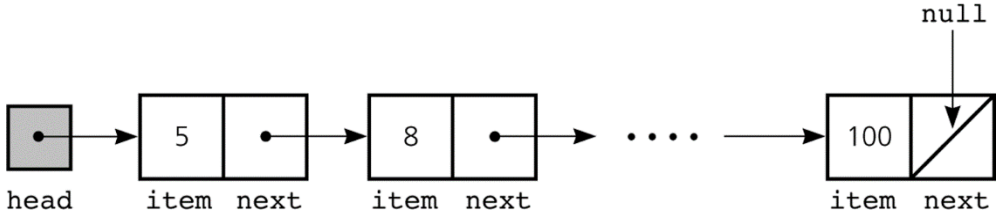
```
// Example of usage of the Node constructors.
Node n = new Node( new Integer(6) );
Node first = new Node( new Integer(9), n );
```



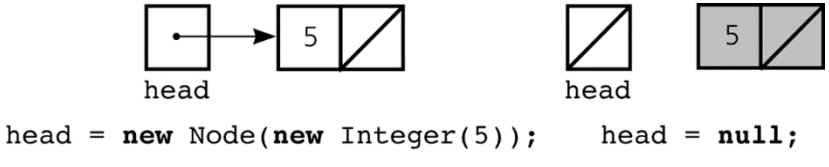
REFERENCE-BASED (LINKED) LISTS 3

OPERATIONS ON LINKED LIST NODES B

Data field **next** in the last node is set to **null** (to detect the end). The reference variable **head** references the first list node, and it exists even if the list is empty.



The reference variable **head** can be assigned **null** without first using **new**. Avoiding, in this way, to loose the Node object created with the **new**.



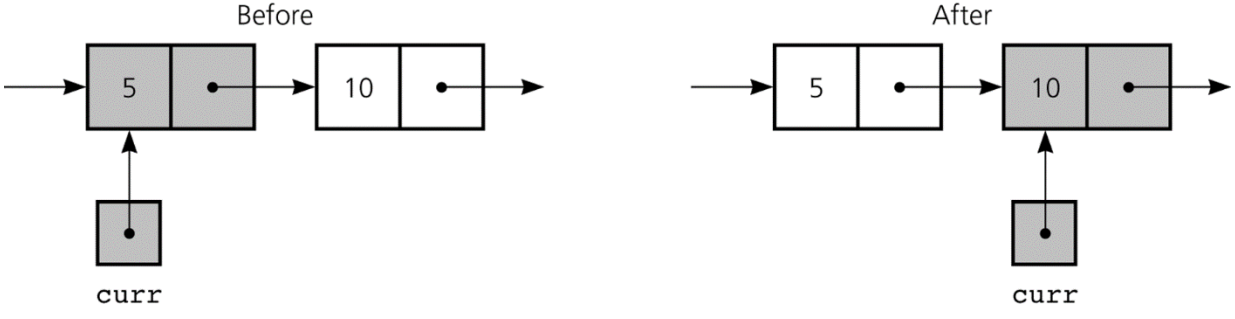
PROGRAMMING WITH LINKED LISTS 1

DISPLAYING THE CONTENTS OF A LINKED LIST

Reference variable **curr** references current node (**1st node of the list**).

To advance **curr** from current position (**before**) to next node (**after**):

```
curr = curr.next;
```



To perform a **list traversal** displaying all the data items in a linked list:

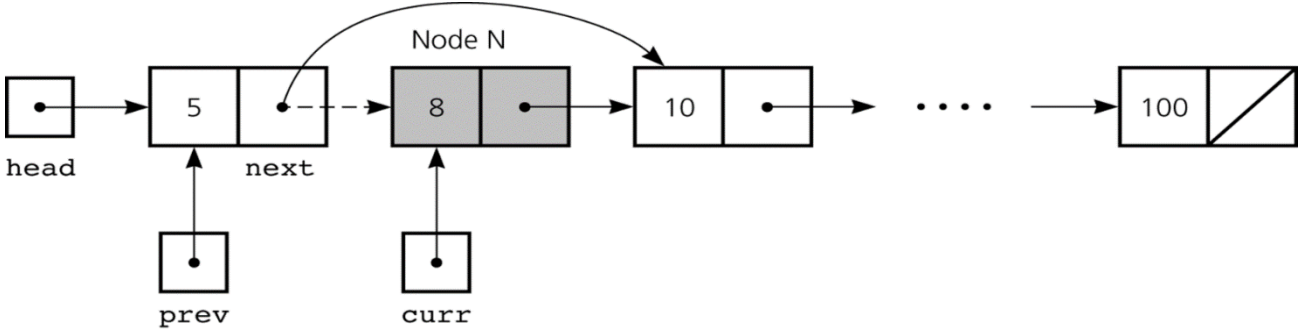
```
for(Node curr = head; curr != null; curr = curr.next) { System.out.println(curr.item); }
```

PROGRAMMING WITH LINKED LISTS 2

DELETING A SPECIFIED NODE FROM A LINKED LIST

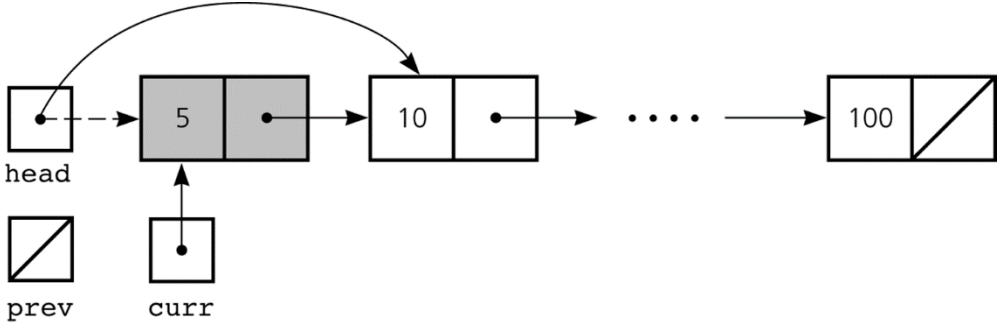
To delete node **N** referenced by **curr**, set **next** in the node that precedes **N** (referenced by **prev**) to reference the node that follows **N**:

```
// Bypass node N.
prev.next = curr.next;
// Opt: unlink node N.
curr.next = null;
// Opt: update curr.
curr = prev.next;
```



Deleting 1st node is a special case:

```
// Special case: delete 1st node.
head = head.next;
```



PROGRAMMING WITH LINKED LISTS 3

RETURN A NODE NO LONGER NEEDED TO THE SYSTEM

To return a **Node** object (referenced by **curr**) that is no longer needed to the system:

1. set its field **next** to **null**, and then
2. set **curr** (referencing the **Node** object) to **null**:

```
curr.next = null; // Unlink node N.
curr = null; // Remove reference to node N.
```

So, in general, the **3 steps to delete a Node object from a linked list** are:

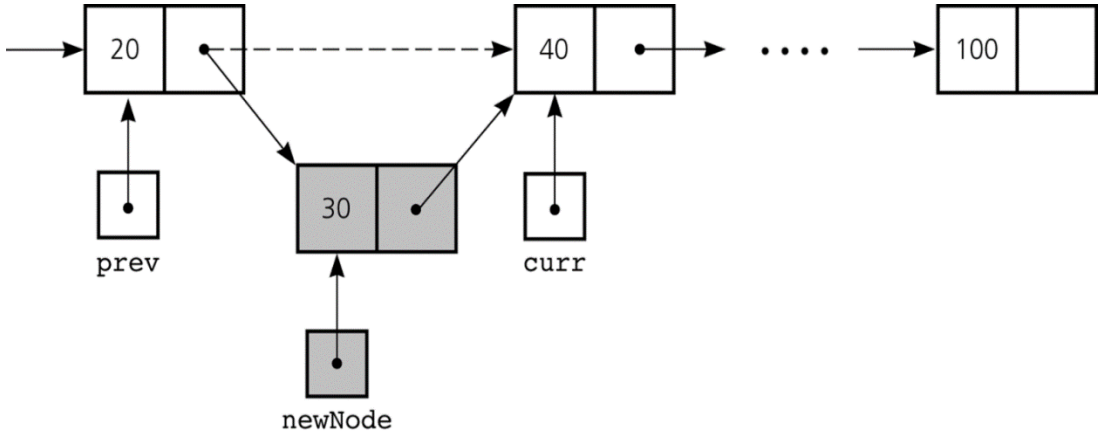
1. locate the **Node** object that you want to delete,
2. disconnect this **Node** object from the linked list by changing references,
3. return the **Node** object to the system.

PROGRAMMING WITH LINKED LISTS 4

INSERTING A NODE INTO A SPECIFIED POSITION OF A LINKED LIST A

Create a new **Node** object **newNode** to store a new item, and insert the new node between 2 nodes (**prev** and **curr**):

```
// Instantiate a new node.
Node newNode = new Node( item );
// Set the new node next field.
newNode.next = curr;
// Update the prev next field.
prev.next = newNode;
```



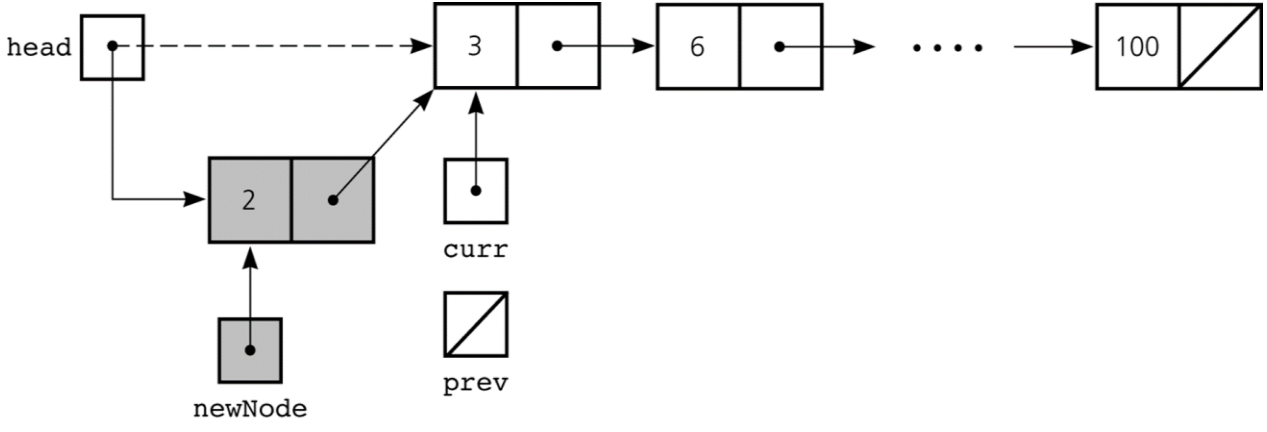
See: visualgo.net/en/list

PROGRAMMING WITH LINKED LISTS 5

INSERTING A NODE INTO A SPECIFIED POSITION OF A LINKED LIST B

Create a new **Node** object **newNode** to store a new item, and add the new node at the beginning of the linked list (**head**):

```
// Instantiate a new node.
Node newNode = new Node(i);
// Add new node in front.
newNode.next = head;
// Link head to new node.
head = newNode;
```



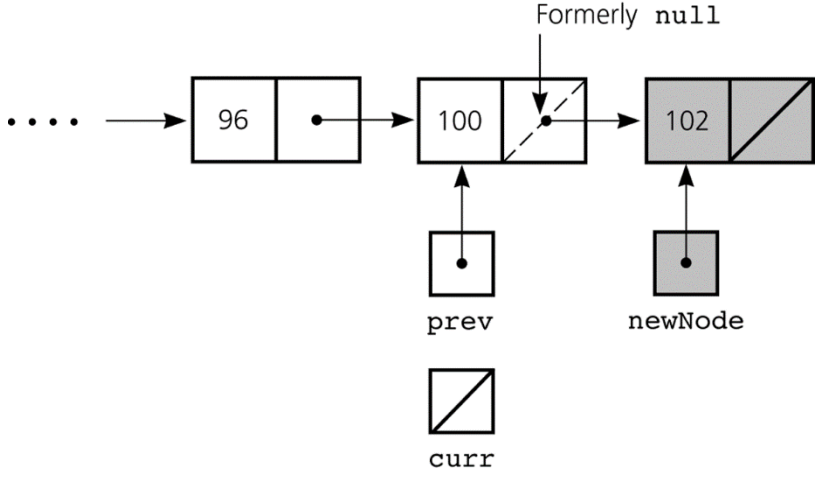
See: visualgo.net/en/list

PROGRAMMING WITH LINKED LISTS 6

INSERTING A NODE INTO A SPECIFIED POSITION OF A LINKED LIST C

Create a new **Node** object **newNode** to store a new item, and insert the new node at the end of a linked list. This is not a special case, since it works as usual if **curr** is **null**:

```
// Instantiate a new node.
Node newNode = new Node( item );
// Set the new node next field.
newNode.next = curr; // Note: curr == null.
// Update the prev next field.
prev.next = newNode;
```



See: visualgo.net/en/list

PROGRAMMING WITH LINKED LISTS 7

INSERTING A NODE INTO A SPECIFIED POSITION OF A LINKED LIST D

So, in general, the **3 steps to insert a new Node object into a linked list** are:

1. determine the point of insertion,
2. create a new **Node** object and store the new data in it,
3. connect the new **Node** object to the linked list by properly changing references.

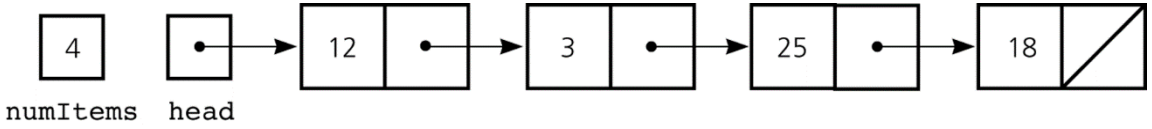
NAVIGATING A SORTED LINKED LIST TO INSERT A NEW VALUE

```
// Note: Java uses "short-circuit evaluation" to evaluate logical expressions.
for( prev = null, curr = head;
     ( curr != null ) && ( newValue.compareTo( curr.item ) > 0 );
     prev = curr, curr = curr.next ) { ... }
```

REFERENCE-BASED ADT LIST IMPLEMENTATION 1

In respect to an array-based implementation, a **reference-based implementation of the ADT list** provides the following advantages:

- it does **not require to shift items** during insertions and deletions, and
- it does **not impose a fixed maximum length** on the list.



Default Constructor: initializes the data fields **numItems** and **head**.

List of Operations: public: **isEmpty**, **size**, **add**, **remove**, **get**, **removeAll**.
private: **find**.

REFERENCE-BASED ADT LIST IMPLEMENTATION 2

LIST INDEX OUT OF BOUNDS EXCEPTION

```
package List;

import java.lang.IndexOutOfBoundsException;
import java.lang.String;

// Exception used for an out-of-bounds list index.
public class ListIndexOutOfBoundsException extends IndexOutOfBoundsException {

   // Constructor.
   public ListIndexOutOfBoundsException( String s ) { super(s); }

}
```

REFERENCE-BASED ADT LIST IMPLEMENTATION 3

LIST INTERFACE

```
package List;
import java.lang.Object;

// Interface providing the specifications for the ADT list operations.
public interface ListInterface {
   public boolean isEmpty(); // Determine whether a list is empty.
   public int size(); // Detemines the length of a list.
   public void removeAll(); // Deleted all the items from the list.
   // Adds an item to the list at position index.
   public void add( int index, Object item ) throws ListIndexOutOfBoundsException;
   // Retrieves a list item by position.
   public Object get( int index ) throws ListIndexOutOfBoundsException;
   // Deletes an item from the list at a given position.
   public void remove( int index ) throws ListIndexOutOfBoundsException;
}
```

REFERENCE-BASED ADT LIST IMPLEMENTATION 4

NODE

```
package List;

// Node of the reference-based ADT list (access is package private).
class Node {
   Object item; // Object data (access is package private).
   Node next; // Reference to the next node (access is package private).

   public Node( Object o ) { item = o; next = null; } // Constructor 1.
   public Node( Object o, Node n ) { item = o; next = n; } // Constructor 2.

   // Note: No other methods needed, because:
   //       - the class is internal to this package, so it is hidden;
   //       - both data fields are accessible directly by other classes in this package.
}
```

REFERENCE-BASED ADT LIST IMPLEMENTATION 5

LIST REFERENCE BASED A

```
package List;

// Reference-based implementation of ADT list.
public class ListReferenceBased implements ListInterface {
   private Node head; // Reference to linked list of items;
   private int numItems; // Number of items in the list.

   // Desc: Locates a specified node in a linked list (private, internal method).
   // Input: index is the position of the desired node ( 0 <= index < numItems ).
   //        Note: index is supposed to be valid (validity check performed elsewhere).
   // Output: Returns a reference to the desired node.
   private Node find( int index ) {
      Node curr = head;
      for( int skip = 0; skip < index; skip++ ) { curr = curr.next; }
      return curr;
   }
```

REFERENCE-BASED ADT LIST IMPLEMENTATION 6

LIST REFERENCE BASED B

```
public ListReferenceBased() { head = null; numItems = 0; } // Default constructor.

public boolean isEmpty() { return ( numItems == 0 ); }

public int size() { return numItems; }

// Desc: Searches and returns a list item by position (public, external method).
// Input: index is the position of desired item ( 0 <= index < numItems ).
//        Note: index could be non-valid (validity check required).
// Output: Returns a reference to the desired item, or an exception if input invalid.
public Object get( int index ) throws ListIndexOutOfBoundsException {
   if( ( index >= 0 ) && ( index < numItems ) ) {
      Node curr = find( index ); // Get the reference to the desired node.
      return curr.item; } // Return (only) the reference to the node data.
   else {
      throw new ListIndexOutOfBoundsException( "Index out of bounds (get)!" ); } }
```

REFERENCE-BASED ADT LIST IMPLEMENTATION 7

LIST REFERENCE BASED C

```
// Desc: Inserts a list item at a specific position (public, external method).
// Input: index is the position of insertion ( 0 <= index < numItems + 1 ).
//        Note: index could be non-valid (validity check required).
// Output: Returns an exception if input index is invalid.
public void add( int index, Object item ) throws ListIndexOutOfBoundsException {
   if( ( index >= 0 ) && ( index < ( numItems + 1 ) ) ) {
      if( index == 0 ) {
         Node newNode = new Node( item, head ); // Create a new node.
         head = newNode; } // Insert new node at the beginning of the list.
      else {
         Node prev = find( index - 1 ); // Find node before insertion position.
         Node newNode = new Node( item, prev.next ); // Insert node (part 1).
         prev.next = newNode; } // Insert node (part 2).
      numItems++; }
   else {
      throw new ListIndexOutOfBoundsException( "Index out of bounds (add)!" ); } }
```

REFERENCE-BASED ADT LIST IMPLEMENTATION 8

LIST REFERENCE BASED D

```
// Desc: Removes a node at a specific position (public, external method).
// Input: index is the position of insertion ( 0 <= index < numItems + 1 ).
//        Note: index could be non-valid (validity check required).
// Output: Returns an exception if input index is invalid.
public void remove( int index ) throws ListIndexOutOfBoundsException {
   if( ( index >= 0 ) && ( index < numItems ) ) {
      if( index == 0 ) { head = head.next; } // Delete the first node of the list.
      else {
         Node prev = find( index - 1 ); // Find the node right before removal index.
         Node curr = prev.next; // Delete the node (part 1).
         prev.next = curr.next; } // Delete the node (part 2).
      numItems--; } // Update list size.
   else {
      throw new ListIndexOutOfBoundsException( "Index out of bounds (remove)!" ); } }
```

REFERENCE-BASED ADT LIST IMPLEMENTATION 9

LIST REFERENCE BASED E

```
   // Desc: Removes all nodes (public, external method).
   public void removeAll() {
      head = null; // Set head to null.
      // Note: The 1st node is now unreferenced, so it is marked for garbage collection.
      // Note: The deletion of 1st node will trigger a garbage collection chain reaction.
      numItems = 0; // Update list size.
   }

}
```

REFERENCE-BASED ADT LIST IMPLEMENTATION 10

COMPARING ARRAY-BASED AND REFERENCE-BASED IMPLEMENTATIONS A

Size (Array-Based): Fixed size means to predict the max number of nodes.
Fixed size involves a waste of storage.

Size (Reference-Based): No fixed size, no max num of nodes, no storage wasted.

Storage (Array-Based): Need less memory than a reference-based ADT list.
Require a contiguous memory area to store the array.

Storage (Reference-Based): Need more storage for the references.
Can store nodes in non-contiguous memory areas.

REFERENCE-BASED ADT LIST IMPLEMENTATION 11

COMPARING ARRAY-BASED AND REFERENCE-BASED IMPLEMENTATIONS B

Access (Array-Based): Constant access time.

Access (Reference-Based): Linear access time (depending on node position).
Linked lists are inherently sequential access.
Nodes non-contiguous, so greater access time.

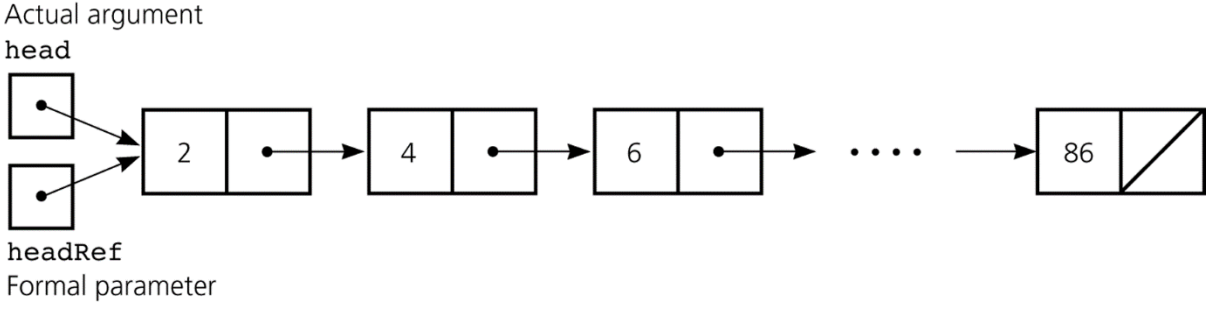
Insert-Delete (Array-Based): Require a shifting of the data.

Insert-Delete (Reference-Based): Do not require a shifting of the data.
Require a list traversal.

PASSING A LINKED LIST TO A METHOD

If a method can access the **head** reference of a linked list it can access the entire list!

When **head** is an actual argument of a method call, its value is copied into the corresponding formal parameter.



PROCESSING LINKED LISTS RECURSIVELY 1

RECURSIVE TRAVERSALS OF A LINKED LIST

Recursive strategy to display (traverse) a list (forward):

1. display the first node of the list, and then
2. display the list minus its first node.

```
private static void displayList( Node currNode ) {
   if( currNode != null ) { // Check if current node reference is valid (not end list).
      System.out.println( currNode.item ); // Display the current (1st) node data.
      displayList( currNode.next ); } } // Display the list minus this node (the 1st).
```

Recursive strategies to display (traverse) a list (backward):

- Version A: display last node **(!)**, then display the list minus its last node backward.
- Version B: display the list minus its first node backward, then display first node.

PROCESSING LINKED LISTS RECURSIVELY 2

RECURSIVE VIEW OF A SORTED LINKED LIST

The linked list that **head** references is a **sorted linked list** if:

head is **null** (an empty list is a sorted list) **OR**
head.next is **null** (a list with a single node is a sorted list) **OR**
(**head.item** < **head.next.item**) **AND** (**head.next** references a sorted list)

```
// Note: Check the use of the Comparable interface in input arguments!
private static Node insertRecursive( Node currNode, java.lang.Comparable newItem ) {
   if( ( currNode == null ) || ( newItem.compareTo( currNode.item ) < 0 ) ) {
      // Base case: Insert newItem at beginning of the list referenced by currNode.
      Node newNode = new Node( newItem, currNode ); currNode = newNode; }
   else { // Recurrence Relation: Insert newItem into rest of linked list (size - 1).
      Node nextNode = insertRecursive(currNode.next, newItem); currNode.next=nextNode; }
   return currNode; }
```

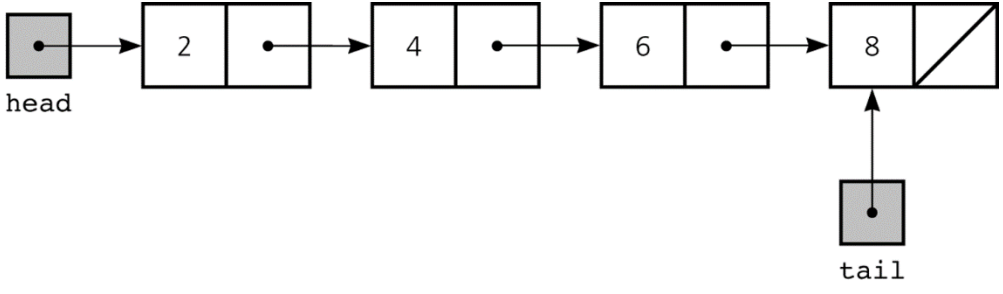
VARIATIONS OF LINKED LISTS 1

LINKED LIST WITH TAIL REFERENCES

A standard linked list can be modified integrating **tail references** in order to:

- remember where the end of the linked list is,
- easily add a node to the end.

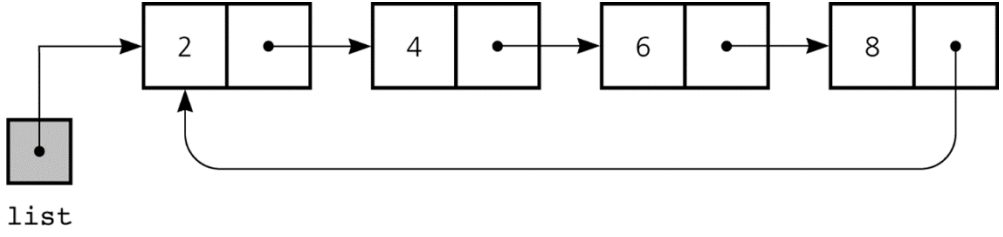
```
tail.next = new Node( item, null ); // Add a new node at the end of the list.
tail = tail.next; // Update tail so that it references the new last node.
```



VARIATIONS OF LINKED LISTS 2

CIRCULAR LINKED LISTS

In a **circular linked list** the last node references the first node, and every node has a successor. A circular linked list still has an **external reference to one of the nodes** (i.e. the **list** variable).




```
// Display the data in a circular linked list, where "list" references the last node.
if( list != null ) {
   Node first = list.next; // Get the reference to the first node.
   Node curr = first; // Start at first node.
   do { System.out.println( curr.item ); // Display node data.
        curr = curr.next; } // Get the reference to the next node.
   while( curr != first ); } // List traversal completed.
```

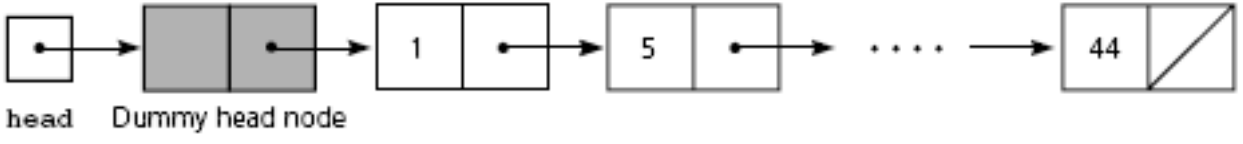
VARIATIONS OF LINKED LISTS 3

LINKED LISTS WITH DUMMY HEAD NODES

In some cases it may be useful to eliminate the need for special cases to handle insertion/deletion at the beginning of a linked list. A solution is to integrate a **dummy head node** at the beginning of a linked list:

- the dummy head node is always present (even when the linked list is empty),
- insert-delete init **prev** to reference the dummy head node (rather than **null**).

```
// Remove node referenced by curr (it works even if curr is the 1st node of the list).
prev.next = curr.next;
curr = curr.next;
```

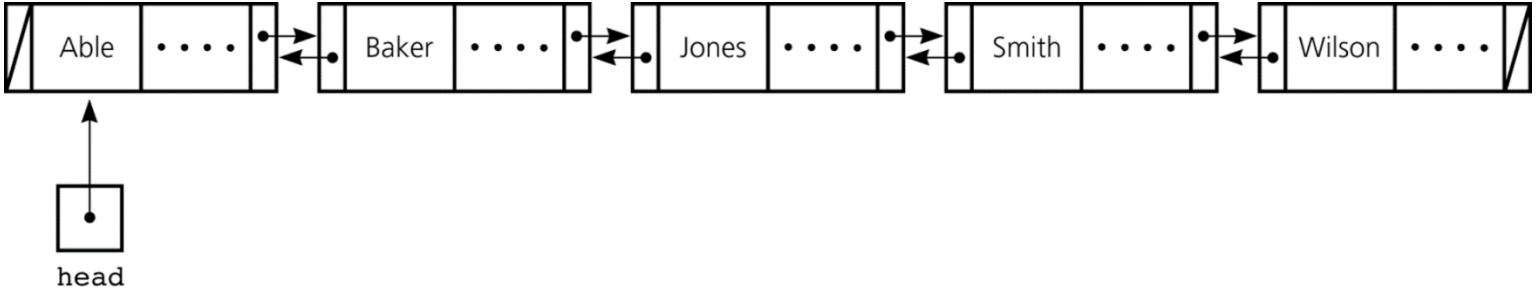


VARIATIONS OF LINKED LISTS 4

DOUBLY LINKED LISTS

If we have to traverse the list forward and backward, we need a **doubly linked list**, where each node references both its predecessor (**prev**) and its successor (**next**).

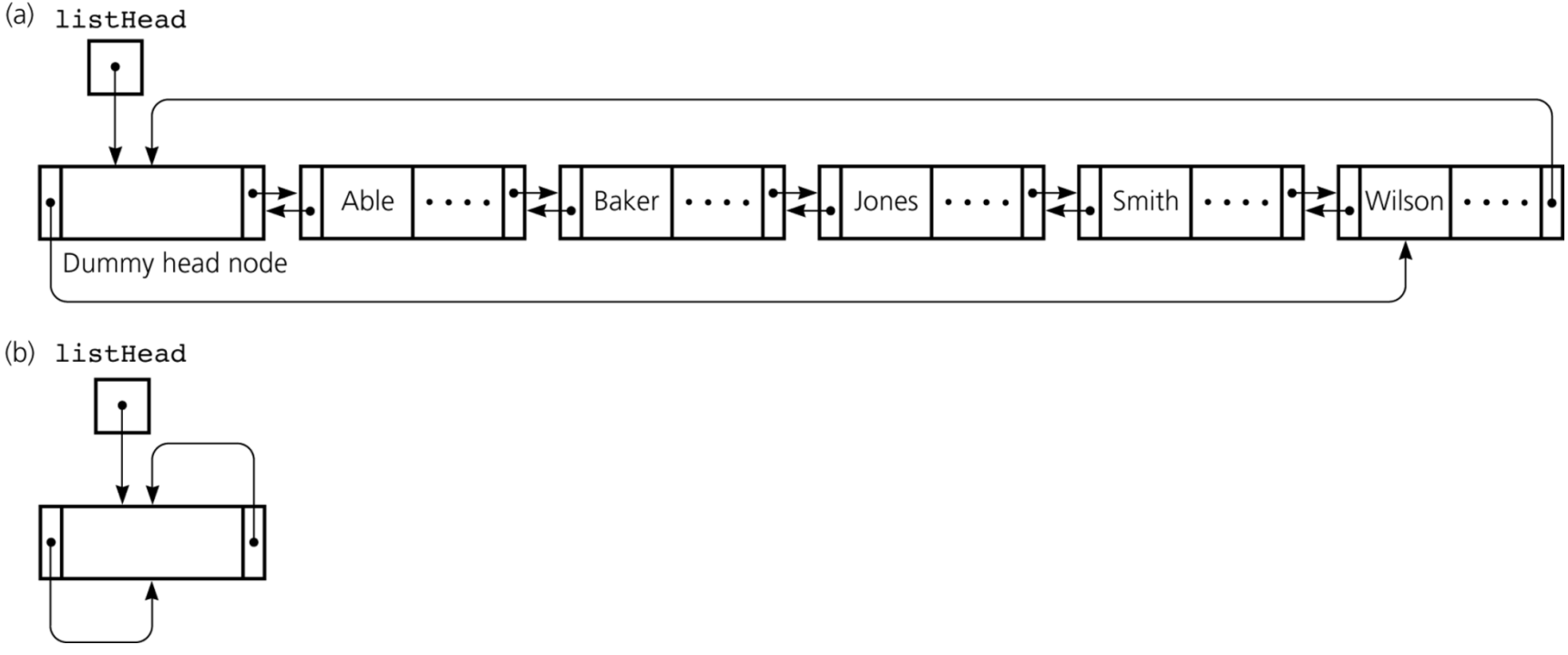
In these lists, **dummy head nodes** can also be useful to simplify insertion-deletion.



See: visualgo.net/en/list

VARIATIONS OF LINKED LISTS 5

Examples of **circular doubly linked lists with dummy head nodes**:
a list with 5 nodes **(a)**, and an empty list **(b)**.

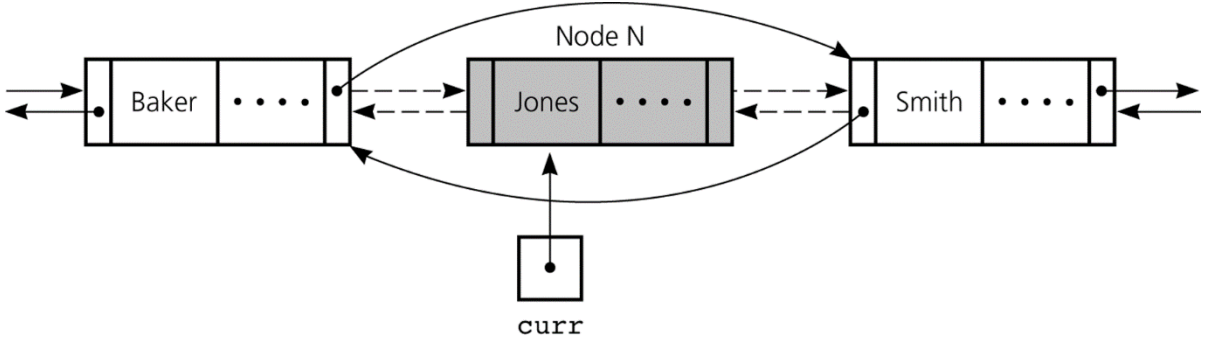


VARIATIONS OF LINKED LISTS 6

DELETION IN DOUBLY LINKED LISTS

To delete the node referenced by **curr**:

```
curr.prev.next = curr.next; // Set next of node before curr to the node after curr.
curr.next.prev = curr.prev; // Set prev of node after curr to the node before curr.
```



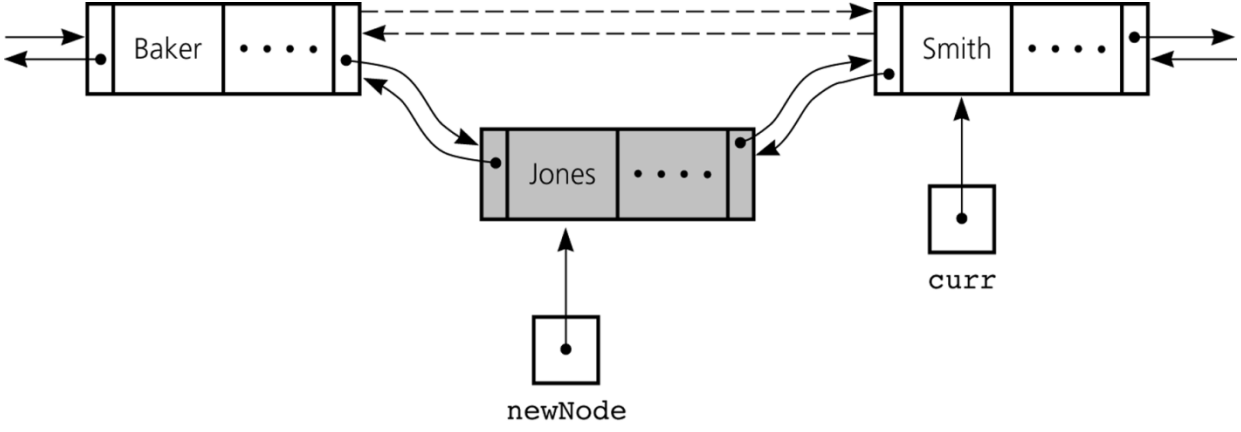
See: visualgo.net/en/list

VARIATIONS OF LINKED LISTS 7

INSERTION IN DOUBLY LINKED LISTS

To insert a new node that **newNode** references **before** the node referenced by **curr**:

```
newNode.next = curr; // Set next in new node to curr.
newNode.prev = curr.prev; // Set prev in new node to node before curr.
curr.prev = newNode; // Set prev of curr to new node.
newNode.prev.next = newNode; // Set next in node before new node to new node.
```



See: visualgo.net/en/list

THE JAVA COLLECTION FRAMEWORK 1

The **Java Collection Framework (JCF)** implements many of the standard ADTs, and it includes: **interfaces**, **implementations**, **iterators**, and (**polymorphic**) **algorithms**.

See: docs.oracle.com/javase/8/docs/technotes/guides/collections/overview

THE JAVA COLLECTION FRAMEWORK 2

GENERICS

Generics classes/interfaces **defer certain data-type info until these classes/interfaces are used**. In a generic, the class/interface definition is followed by **<E>**, where **E** represents the data type (only **object types**) the client will specify.

See: docs.oracle.com/javase/tutorial/java/generics/index

```
public class MyGenericClass<E> { // Example usage: new MyGenericClass<String>("",1).
   private E data;
   private int num;
   public MyGenericClass( E initD, int initN ) { data = initD; num = initN; }
   public void setData( E newD ) { data = newD; }
   public E getData() { return data; }
   public int getNum() { return num; } }
```

THE JAVA COLLECTION FRAMEWORK 3

ITERATORS AND ITERABLE COLLECTIONS A

- An **iterator** allows to cycle items in a **collection** (an object storing other objects).
- An iterator **iter** allows to access the next collection item with: **iter.next()**.
- JCF has 2 main iterator interfaces: **java.util.Iterator** and **java.util.ListIterator**.
- Each ADT collection in the JCF has a method to return an iterator object.

See: docs.oracle.com/javase/tutorial/collections/interfaces/collection
See: docs.oracle.com/javase/8/docs/api/java/util/iterator
See: docs.oracle.com/javase/8/docs/api/java/util/listiterator

```
public interface Iterator<E> { // The java.util.Iterator interface.
   boolean hasNext(); // Returns true if the iteration has more elements.
   E next(); // Returns the next element in the iteration.
   ... }
```

THE JAVA COLLECTION FRAMEWORK 4

ITERATORS AND ITERABLE COLLECTIONS B

- When an iterator is created, the first call to **next()** returns the 1st collection item.
- The basis for the ADT collections in the JCF is the interface **java.util.Iterable**, with the subinterface **java.util.Collection**. Thus, every ADT collection in the JCF will have a method to return an iterator object for the underlying collection.

Note: You can use inheritance to derive new interfaces (called **subinterfaces**).

See: docs.oracle.com/javase/8/docs/api/java/lang/iterable
See: docs.oracle.com/javase/8/docs/api/java/util/collection

```
public interface Iterable<E> { // The java.util.Iterable interface.
   Iterator<E> iterator(); } // Returns an iterator over this collection elements.
```

THE JAVA COLLECTION FRAMEWORK 5

ITERATORS AND ITERABLE COLLECTIONS C

The following is a **portion** of the subinterface **java.util.Collection**:

```
public interface Collection<E> extends Iterable<E> { // java.util.Collection interface.
   // Note: only a portion of the interface appears here!
   boolean add( E o ); // Ensures that collection contains "o" (optional).
   boolean remove( Object o ); // Removes "o" from collection (optional).
   void clear(); // Removes all of the elements from this collection (optional).
   boolean contains( Object o ); // Returns true if collection contains element "o".
   boolean equals( Object o ); // Compares "o" with this collection for equality.
   boolean isEmpty(); // Returns true if this collection contains no elements.
   int size(); // Returns the number of elements in this collection.
   Object[] toArray(); // Returns an array containing all elements in collection.
   ... }
```

See: docs.oracle.com/javase/8/docs/api/java/util/collection

THE JAVA COLLECTION FRAMEWORK 6

ITERATORS AND ITERABLE COLLECTIONS D

This example shows how an iterator can be used with the JCF list class **LinkedList**:

```
import java.util.LinkedList;
import java.util.Iterator;
public class TestLinkedList {
   public static void main( String[] args ) {
      LinkedList<Integer> myList = new LinkedList<Integer>();
      Iterator iter = myList.iterator();
      if( !iter.hasNext() ) { System.out.println( "The list is empty!" ); }
      for( int i = 1; i <= 5; i++ ) { myList.add( new Integer(i) ); }
      iter = myList.iterator(); // Collection modified, request another iterator!
      while( iter.hasNext() ) { System.out.println( iter.next() ); } } }
```

Note: The iterator behavior is unspecified if the collection is modified while the iteration is in progress (in any way other than by calling the **remove** method)!

THE JAVA COLLECTION FRAMEWORK 7

ITERATORS AND ITERABLE COLLECTIONS E

The **java.util.ListIterator** subinterface extends the **java.util.Iterator**, by providing support also for **bidirectional access** (**next** and **previous**) to the collection.

See: docs.oracle.com/javase/8/docs/api/java/util/listiterator

```
public interface ListIterator<E> extends Iterator<E> { // java.util.ListIterator.
   void add( E o ); // Inserts the specified element into the list (optional).
   boolean hasNext(); // True if iterator has more elements when forward traversing.
   boolean hasPrevious(); // True if iterator has more elements when reverse traversing.
   E next(); // Returns the next element in the list.
   int nextIndex(); // Index of the element returned by a subsequent next call.
   E previous(); // Returns the previous element in the list.
   int previousIndex(); // Index of the element returned by a subsequent previous call.
   void remove(); // Removes from list last element returned by next/previous (optional).
   void set( E o ); } // Set last element returned by next/previous to "o" (optional).
```

THE JAVA COLLECTION FRAMEWORK 8

ITERATORS AND ITERABLE COLLECTIONS F

The JCF **java.util.List** subinterface supports an **ordered collection** (aka **sequence**), allowing add/remove by index and providing a **ListIterator** for bidirectional access.

See: docs.oracle.com/javase/8/docs/api/java/util/list

```
public interface List<E> extends Collection<E> { // The java.util.List subinterface.
   void add( int i, E o ); // Inserts "o" at position "i" (optional).
   E get( int i ); // Returns the element at position "i" in this list.
   int indexOf( Object o ); // Returns index of first occurrence of "o", otherwise -1.
   ListIterator<E> listIterator(); // Returns list iterator of elements in proper order.
   ListIterator<E> listIterator( int i ); // List iterator starting at position "i".
   E remove( int i ); // Removes the element at position "i" in this list (optional).
   E set( int i, E o ); // Replaces element at position "i" with "o" (optional).
   List<E> subList( int fromIndex, int toIndex ); // Returns subset of the list.
   ... }
```

THE JAVA COLLECTION FRAMEWORK 9

ITERATORS AND ITERABLE COLLECTIONS G

The JCF provides many classes that implement the **java.util.List** interface, including:

- **java.util.LinkedList**,
- **java.util.ArrayList**, and
- **java.util.Vector**.

See: docs.oracle.com/javase/8/docs/api/java/util/list
See: docs.oracle.com/javase/8/docs/api/java/util/linkedlist
See: docs.oracle.com/javase/8/docs/api/java/util/arraylist
See: docs.oracle.com/javase/8/docs/api/java/util/vector

THE JAVA COLLECTION FRAMEWORK 10

ITERATORS AND ITERABLE COLLECTIONS H

The following is an example of how to use the **ArrayList** class:

See: docs.oracle.com/javase/8/docs/api/java/util/arraylist

```
import java.util.ArrayList;
import java.util.Iterator;
...
ArrayList<String> groceryList = new ArrayList<String>(); // New empty ArrayList.
groceryList.add( "Apples" ); // Add as many items you want...
System.out.println( "Number of items on my grocery list: " + groceryList.size() );
System.out.println( "Items are: " );
Iterator<String> iter = groceryList.iterator(); // Get the iterator.
while( iter.hasNext() ) { // Traverse the list until there is no other element.
   String nextItem = iter.next(); // Get the next element using the iterator.
   System.out.println( groceryList.indexOf( nextItem ) + " - " + nextItem ); }
```