GIUSEPPE TURINI

CS-102: COMPUTING AND ALGORITHMS 2
LESSON 03
DATA ABSTRACTION, ADTs, AND LISTS

Kettering UNIVERSITY

HIGHLIGHTS

Abstract Data Types (ADTs)

ADT vs Data Structure, Data Abstraction and Axioms, Implementing ADTs

ADT List

Description and Operations, Axioms, and the ADT Sorted List

Implementing Array-based List, Array-based vs Reference-based

Linked Lists (LLs) (Reference-based Lists)

Concept, Basic Implementation, Programming with LLs, Full Implementation

References vs Arrays, Pass LLs to Methods, Process LLs Recursively

LL Variations: with Tail References, Circular, Doubly, with Dummy Head Node

Java Collection Framework (JCF)

Introduction, Generics, Iterators and Iterable Collections, JCF ArrayList

STUDY GUIDE

STUDY MATERIAL

- **This slides.**

SELECTED EXERCISES

- **Set 1:** ex. 1, ex. 5-7, ex. 9-10, ex. 12, ex. 14.
- **Set 2:** ex. 1-5, ex. 7, ex. 10, ex. 12-14, ex. 19.

Additional Resources

- **"Object-Oriented Data Structures Using Java(4th Ed.)", chap. 2, chap. 6.**
- "Data Abstraction and Problem Solving with Java (3rd Ed.)", chap. 4, chap. 5.
- "Java Illuminated (5th Ed.)", chap. 14.
- visualgo.net/en/list

ADTs - ABSTRACT DATA TYPE vs DATA STRUCTURE

Abstract Data Type (ADT): An ADT is composed of:

- a **collection of data**, and
- a **set of operations** on that data.

Typical Operations on Data:

- **Add data** to a data collection.
- **Remove data** from a data collection.
- **Query the data** in a data collection.

ADT Specifications: What ADT operations do, not how to implement them.
ADT Implementation: It includes choosing a particular data structure.
Data Structure: Construct defined to store a collection of data (e.g. arrays).

Abstract data types (ADTs) and data structures are not the same!

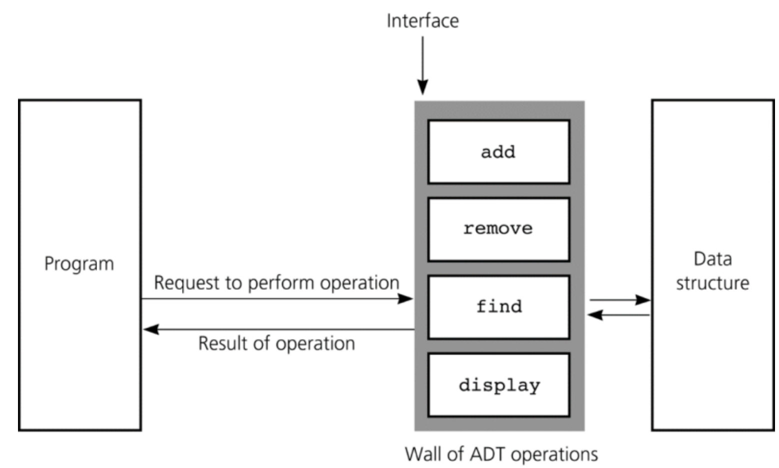
ADTs - DATA ABSTRACTION AND AXIOMS

Data Abstraction: Results in a wall of ADT operations which isolates data structures from the program that accesses the data stored in these data structures.

Designing an Abstract Data Type (ADT): The design of an ADT should evolve naturally during the problem-solving process:

- **What data does a problem require?**
- **What operations does a problem require?**

Note: For complex ADTs, operations are specified by **axioms** (math rules).

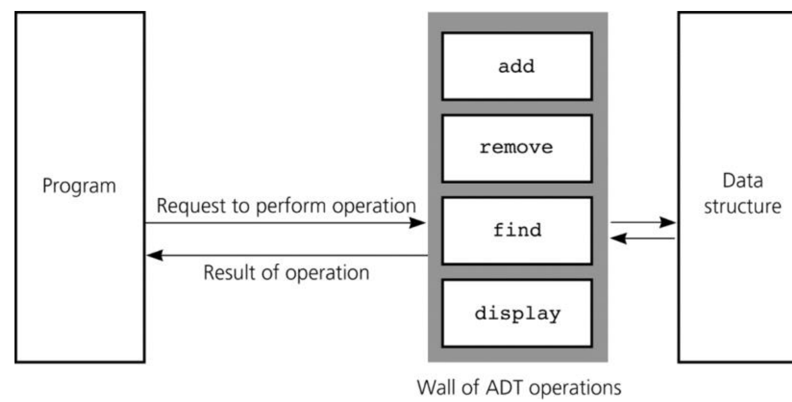


ADTs - IMPLEMENTING ADTs 1

Choose the data structure to represent the data of the ADT, considering:

- the details of the operations of the ADT, and
- the context in which the operations will be used.

Implementation details hidden behind a wall of ADT operations. This means that a program will only be able to access the data structure using the ADT operations.

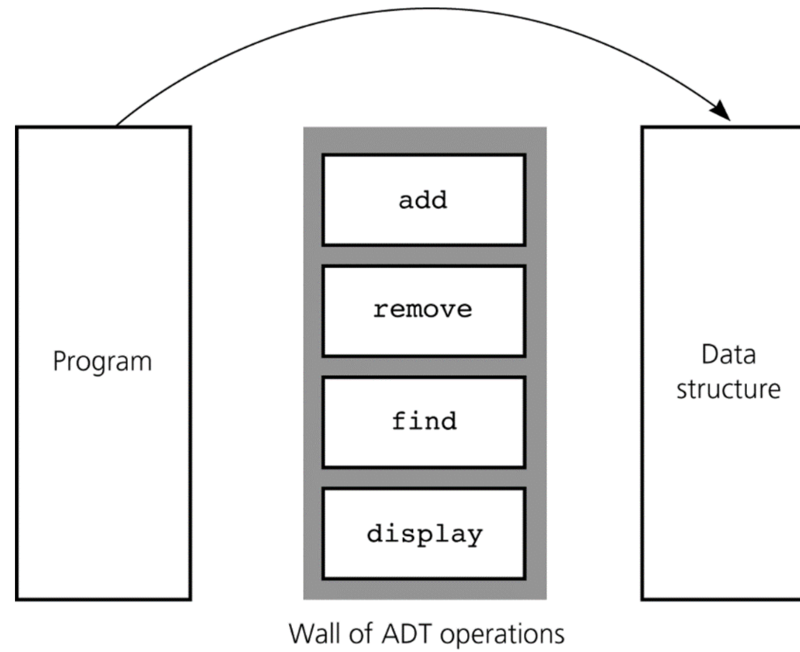


ADTs - IMPLEMENTING ADTs 2

VIOLATING THE WALL OF ADT OPERATIONS

In a **non**-object-oriented implementation, both the **data structure** and the **ADT operations** are distinct pieces…

In this case the data structure is hidden only if the program using the ADT does not look over the wall!



Object-oriented languages provide a way to enforce the wall of an ADT!

ADT LIST - DESCRIPTION AND OPERATIONS 1

THE ADT LIST: DESCRIPTION

Each item (except first and last) has: 1 **unique predecessor**, and 1 **unique successor**.
The first item is called **head** or front, and does not have a predecessor.
The last item is called **tail** or end, and does not have a successor.

THE ADT LIST: SPECIFICATIONS OF THE OPERATIONS

- define the contract for the ADT list,
- do not specify how to store the list or how to perform the operations.

ADT operations can be used without knowing their implementation.

ADT LIST - DESCRIPTION AND OPERATIONS 2

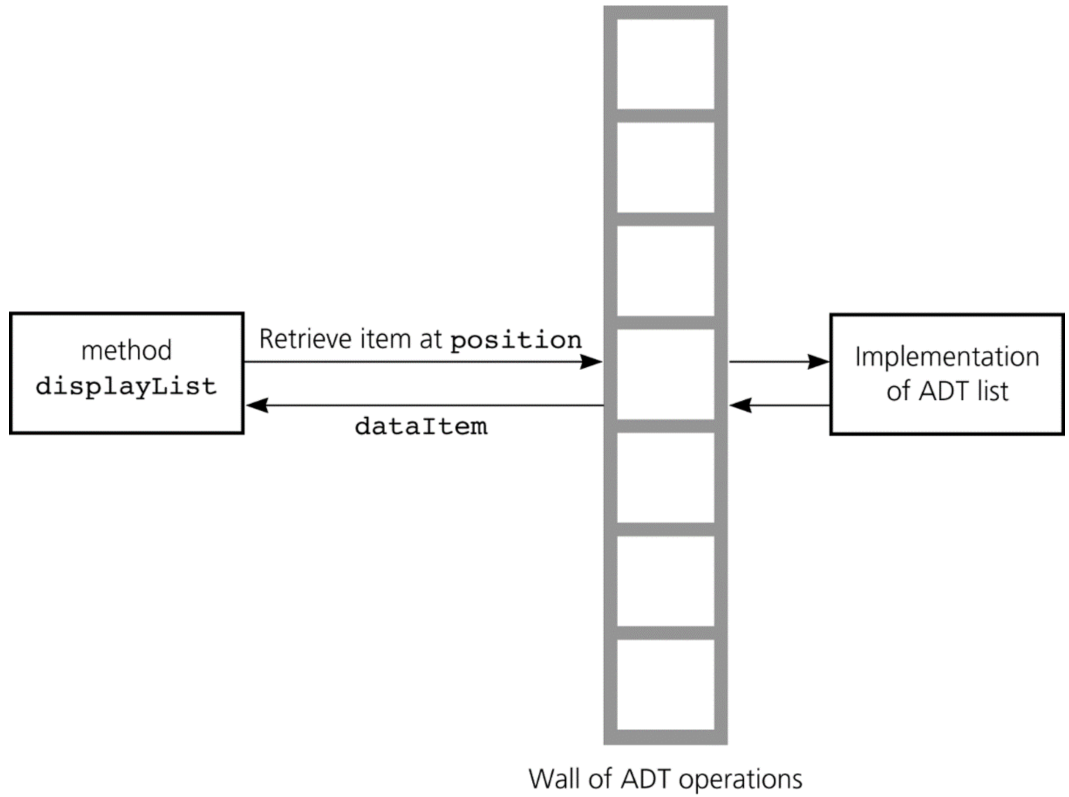
THE ADT LIST: OPERATIONS

1. create an empty list: **aList.createList()**
2. check if a list is empty: **aList.isEmpty()**
3. determine the number of items in a list: **aList.size()**
4. add an item at a given position in the list: **aList.add(i, x)**
5. remove the item at a given position in the list: **aList.remove(i)**
6. remove all the items from the list: **aList.removeAll()**
7. retrieve the item at a given position in the list: **aList.get(i)**
8. print the list: **aList.displayList()**

Note: List items are referenced by their position within the list.

ADT LIST - DESCRIPTION AND OPERATIONS 3

The wall between the method **displayList** and the implementation of the **ADT list**.



ADT LIST - AXIOMS

Example: Axioms to specify the behavior of operations of the **ADT list**:

- Axiom: **(aList.createList()).size() == 0**
- Axiom: **(aList.add(i, x)).size() == aList.size() + 1**
- Axiom: **(aList.remove(i)).size() = aList.size() – 1**
- Axiom: **(aList.createList()).isEmpty() = true**
- Axiom: **(aList.add(i, x)).isEmpty() = false**
- Axiom: **(aList.createList()).remove(i) = error**
- Axiom: **(aList.add(i, x)).remove(i) = aList**
- Axiom: **(aList.createList()).get(i) = error**
- Axiom: **(aList.add(i, x)).get(i) = x**
- Axiom: **aList.get(i) = (aList.add(i, x).get(i + 1)**
- Axiom: **aList.get(i + 1) = (aList.remove(i)).get(i)**

ADT LIST - SORTED LIST (VARIATION)

THE ADT SORTED LIST

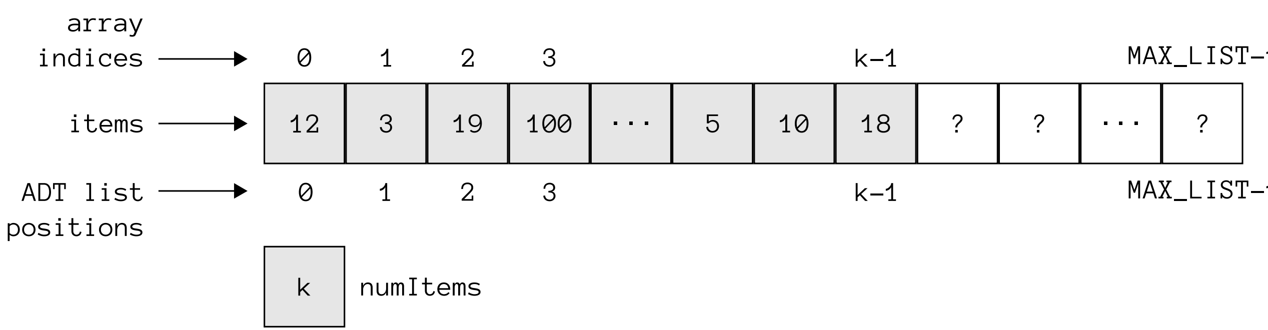
- maintains items in sorted order,
- inserts and deletes items by their values, not by their positions!

ADT LIST - IMPLEMENTING ARRAY-BASED LIST 1

FIRST SKETCH OF AN ARRAY-BASED ADT LIST

The following is a sketch of an array-based implementation of the ADT list:

- all the items of the list are stored in an array **items**,
- the **kth item** will be stored in **items[k]**,
- **max size** of the array is a fixed value **MAX_LIST** (i.e. **physical size**),
- **current number of items** in the list stored in **numItems** (i.e. **logical size**).

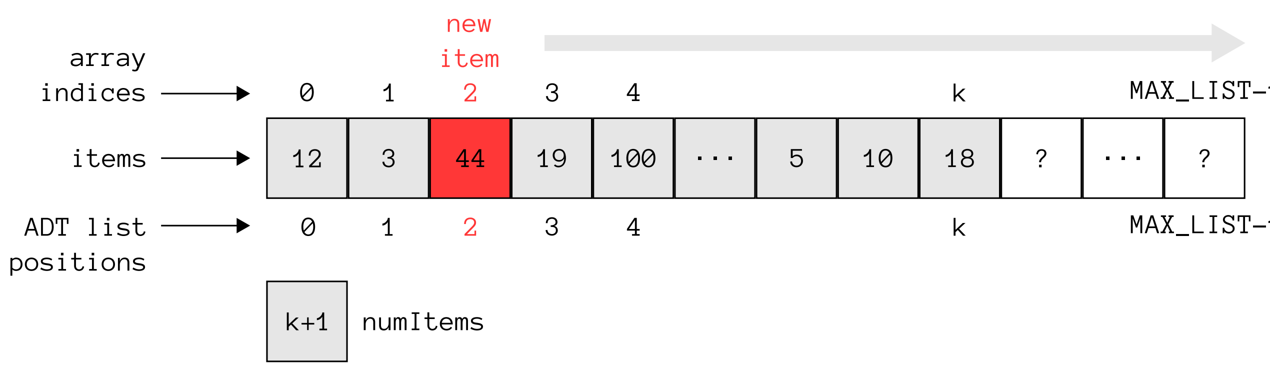


ADT LIST - IMPLEMENTING ARRAY-BASED LIST 2

INSERTION OF AN ITEM INTO THE ARRAY-BASED ADT LIST

To insert a new item at a given position **p** in the array of list items, you must:

1. perform a **shift to the right** of the items from position **p** on, and
2. perform the **insertion of the new item** in the newly created opening (at **p**).

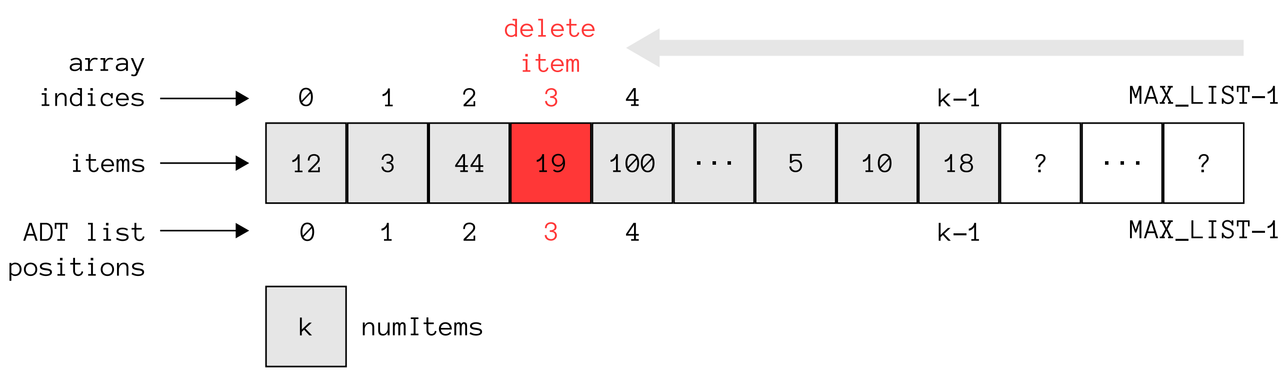


ADT LIST - IMPLEMENTING ARRAY-BASED LIST 3

DELETION OF AN ITEM FROM THE ARRAY-BASED ADT LIST

To delete an item from the list, you must not only to erase the target item but also to remove the gap created by the removal. Therefore:

1. at first, **delete the item** from the array, and
2. then **shift left all the items on the right side** of the newly created gap.



LIST INDEX OUT OF BOUNDS EXCEPTION

```
package List;
// Exception used for an out-of-bounds list index.
public class ListIndexOutOfBoundsException extends IndexOutOfBoundsException {
   // Constructor.
   public ListIndexOutOfBoundsException( String s ) { super(s); } }
```

LIST EXCEPTION

```
package List;
// Exception used when the array storing the list becomes full.
public class ListException extends RuntimeException {
   // Constructor.
   public ListException( String s ) { super(s); } }
```

ADT LIST - IMPLEMENTING ARRAY-BASED LIST 5

LIST INTERFACE

```java
package List;
// 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 i, Object o) throws ListIndexOutOfBoundsException, ListException;

   // Retrieves a list item by position.
   public Object get(int i) throws ListIndexOutOfBoundsException;

   // Deletes an item from the list at a given position.
   public void remove(int i) throws ListIndexOutOfBoundsException;
}
```

ADT LIST - IMPLEMENTING ARRAY-BASED LIST 6

LIST ARRAY BASED A

```
package List;
// Array-based implementation of the ADT list.
public class ListArrayBased implements ListInterface {
   private static final int MAX_LIST = 50; // Maximum (physical) size of the list.
   private Object[] items; // An array of list items.
   private int numItems; // Number of items (logical size) of the list.

   public ListArrayBased() { items = new Object[ MAX_LIST ]; numItems = 0; }

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

   public int size() { return numItems; }

   public void removeAll() {
      // Creates a new array, and marks old array for garbage collection.
      items = new Object[ MAX_LIST ];
      numItems = 0; }
```

LIST ARRAY BASED B

```
public void add( int index, Object item )
   throws ListIndexOutOfBoundsException, ListException {
      if( numItems == MAX_LIST ) {
         throw new ListException( "ListException on add." ); }
      if( ( index >= 0 ) && ( index <= numItems ) ) {
         // Insert new item by right shifting all items at position >= index.
         for( int pos = numItems; pos > index; pos-- ) {
            items[ pos ] = items[ pos-1 ]; }
         items[ index ] = item; // Insert new item.
         numItems++; }
      else {
         throw new ListIndexOutOfBoundsException("ListIndexOutOfBoundsException"); } }
```

ADT LIST - IMPLEMENTING ARRAY-BASED LIST 8

LIST ARRAY BASED C

```
   public Object get( int index ) throws ListIndexOutOfBoundsException {
      if( ( index >= 0 ) && ( index < numItems ) ) { return items[ index ]; }
      else {
         throw new ListIndexOutOfBoundsException("ListIndexOutOfBoundsException"); } }

   public void remove( int index ) throws ListIndexOutOfBoundsException {
      if( ( index >= 0 ) && ( index < numItems ) ) {
         // Delete item by left shifting all items at position > index.
         for( int pos = index+1; pos < numItems; pos++ ) {
            items[ pos-1 ] = items[ pos ]; }
         items[ numItems-1 ] = null;
         numItems--; }
      else {
         throw new ListIndexOutOfBoundsException("ListIndexOutOfBoundsException"); } }

}
```

ADT LIST - ARRAY-BASED VS REFERENCE-BASED

DATA STRUCTURES FOR IMPLEMENTING AN ABSTRACT DATA TYPE (ADT)

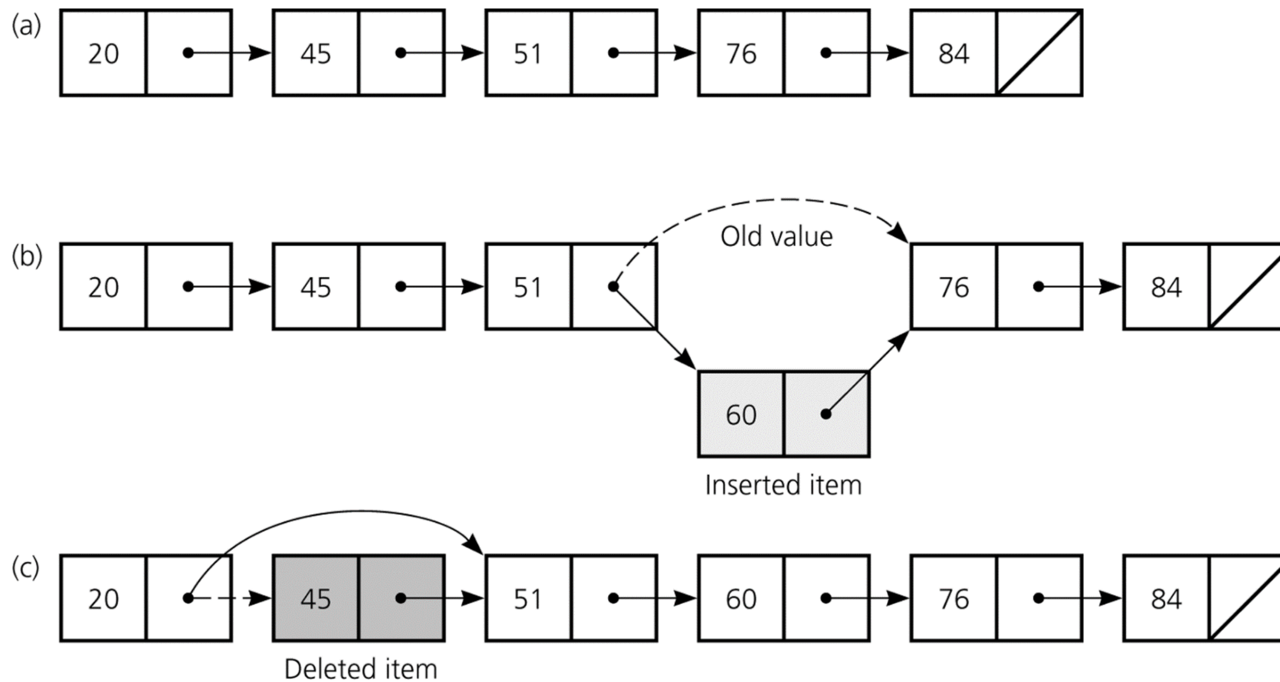
Array-based List: has a **fixed size**, and
the **data must be shifted** during insertions and deletions.

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

LINKED LIST - CONCEPT

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

See: visualgo.net/en/list



Kettering University

LINKED LIST - BASIC IMPLEMENTATION 1

DEFINITION

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.

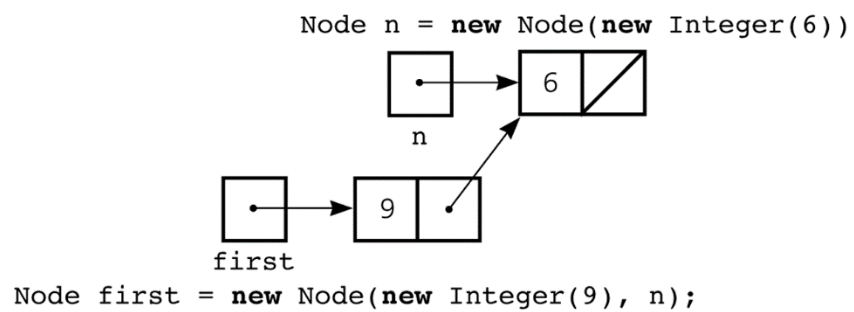
LINKED LIST - BASIC IMPLEMENTATION 2

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.
   ... }
```

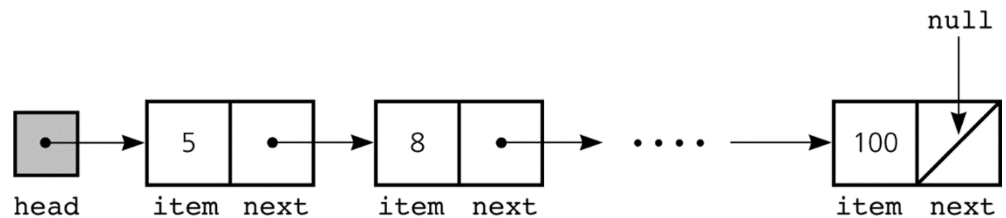
```
// Example of usage of the Node constructors.
Node n = new Node( new Integer(6) );
Node first = new Node( new Integer(9), n );
```



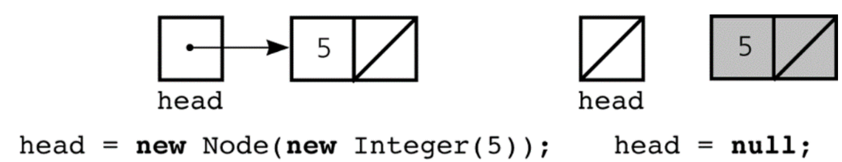
LINKED LIST - BASIC IMPLEMENTATION 3

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**.



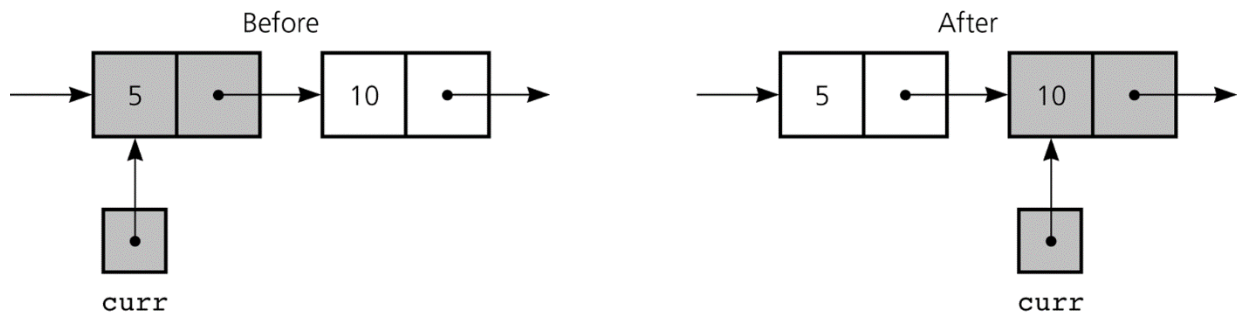
LINKED LIST - PROGRAMMING WITH LLs 1

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:

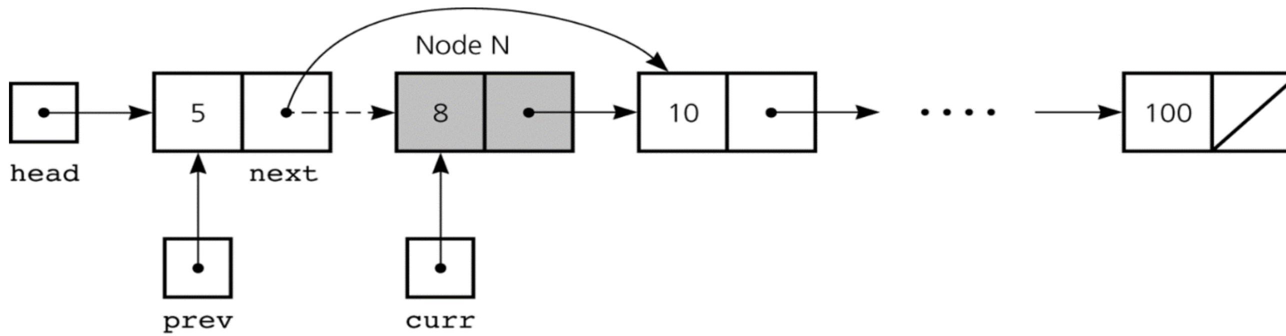
```
for(Node curr = head; curr != null; curr = curr.next) { System.out.println(curr.item); }
```

LINKED LIST - PROGRAMMING WITH LLs 2

DELETING A SPECIFIED NODE FROM A LINKED LIST

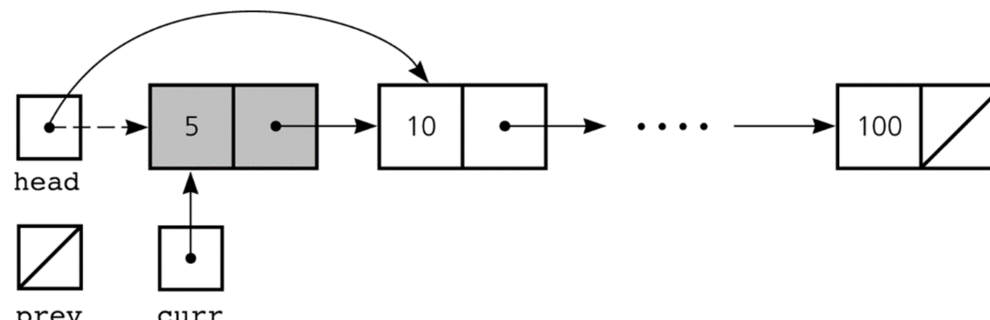
To delete node **N** referenced by **curr**, set **next** in the node that precedes **N** (referenced by **prev**) to reference the node that follows **N**:

```
// 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;
```



LINKED LIST - PROGRAMMING WITH LLs 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:

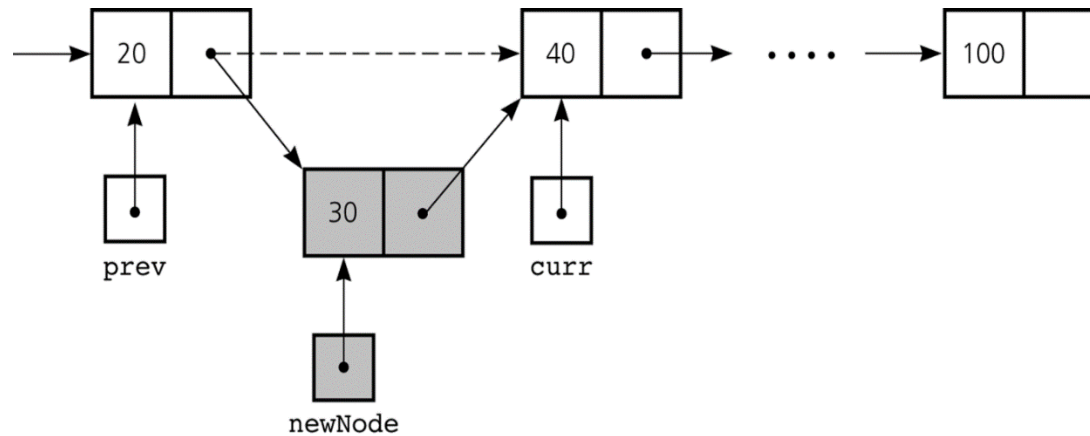
1. locate the **Node** object that you want to delete,
2. disconnect this **Node** object from the linked list by changing references,
3. return the **Node** object to the system.

LINKED LIST - PROGRAMMING WITH LLs 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;
```



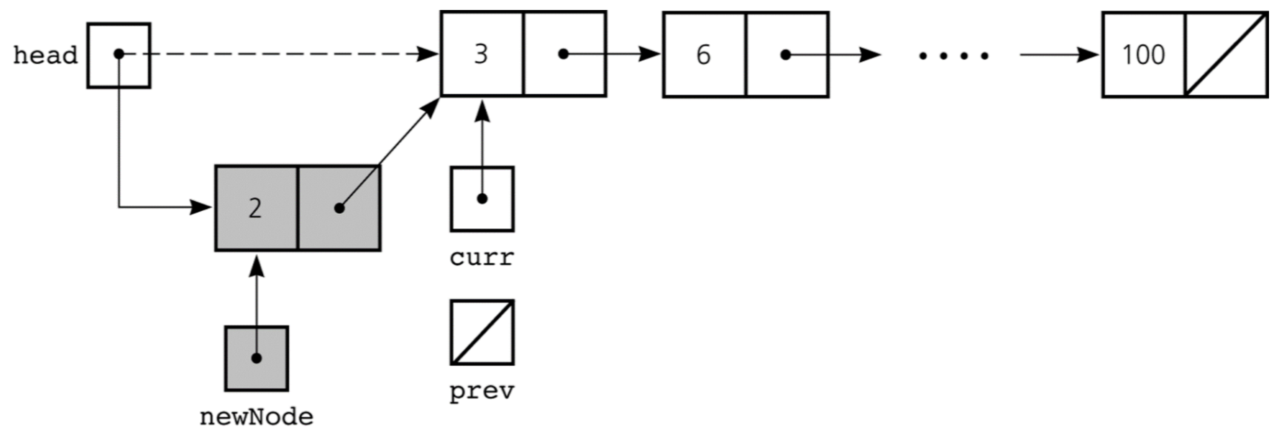
See: visualgo.net/en/list

LINKED LIST - PROGRAMMING WITH LLs 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;
```



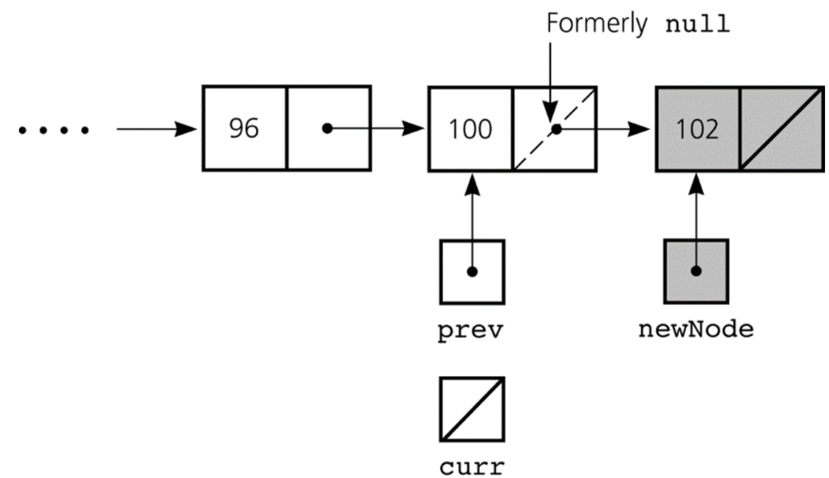
See: visualgo.net/en/list

LINKED LIST - PROGRAMMING WITH LLs 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

LINKED LIST - PROGRAMMING WITH LLs 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 ) { ... }
```

LINKED LIST - FULL IMPLEMENTATION 1

The following is a Java implementation of a linked list including:

- The **package List** to bundle all classes related to this implementation.
- The **class ListIndexOutOfBoundsException** representing a non-critical runtime error (unchecked exception) relative to a list index out-of-range.
- The **interface ListInterface** providing the specifications of all the methods a class has to implement in order to represent a list.
- The **class Node** representing a list node. This class is an internal class (package-private) since it is only needed by the class implementing the list.
- The **class ListReferenceBased** providing an implementation of a linked list.

LINKED LIST - FULL IMPLEMENTATION 2

LIST INDEX OUT OF BOUNDS EXCEPTION

```java
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); }

}
```

LINKED LIST - FULL 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;
}
```

LINKED LIST - FULL 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.
}
```

LINKED LIST - FULL 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;
   }
```

LINKED LIST - FULL IMPLEMENTATION 6

LIST REFERENCE BASED B

```java
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)!" ); } }
```

LINKED LIST - FULL 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)!" ); } }
```

LINKED LIST - FULL 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)!" ); } }
```

LINKED LIST - FULL IMPLEMENTATION 9

LIST REFERENCE BASED E

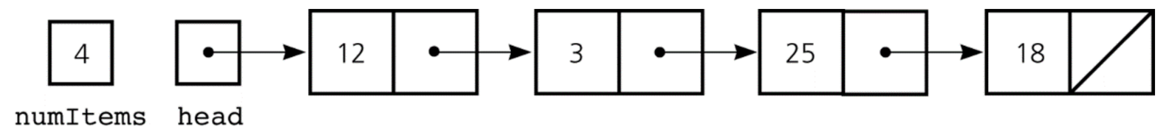
```
  // Desc: Removes all nodes (public, external method).
  public void removeAll() {
    head = null; // Set head to null.
    // Note: The 1st node is now unreferenced, so it is marked for garbage collection.
    // Note: The deletion of 1st node will trigger a garbage collection chain reaction.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

}
```

LINKED LIST - REFERENCES vs ARRAYS 1

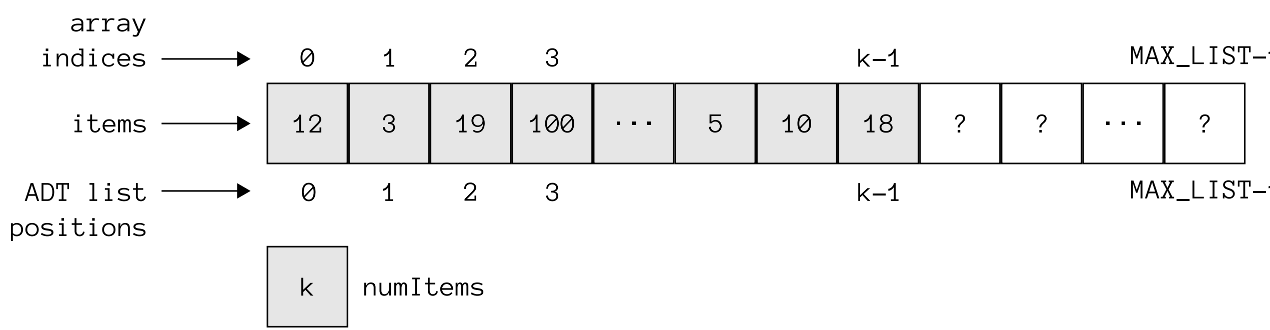
In respect to an array-based implementation, a **reference-based implementation of the ADT list** provides the following advantages:

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

Reference-based List:



Array-based List:



LINKED LIST - REFERENCES vs ARRAYS 2

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.

LINKED LIST - REFERENCES vs ARRAYS 3

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.

LINKED LIST - REFERENCES vs ARRAYS 4

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

LINKED LIST - PASS LINKED LISTS TO METHODS

There are 2 ways to pass a linked list to a method:

- **Client-code (external to the package)** sees a linked list as an object, so this object can be passed to a method using a reference.

```
LinkedList myList = new LinkedList(); // Create a new empty LL.
// ...
printList(myList); // Passing a LL by reference.
```

- **Internal code (inside the package)** could see the **Node** class, if so a linked list can be passed to a method just by passing a reference to its head node.

```
// ...
deleteList(myList.head); // Passing a LL by passing a reference to its head node.
```

LINKED LIST - PROCESS LLs 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.

LINKED LIST - PROCESS LLs 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; }
```

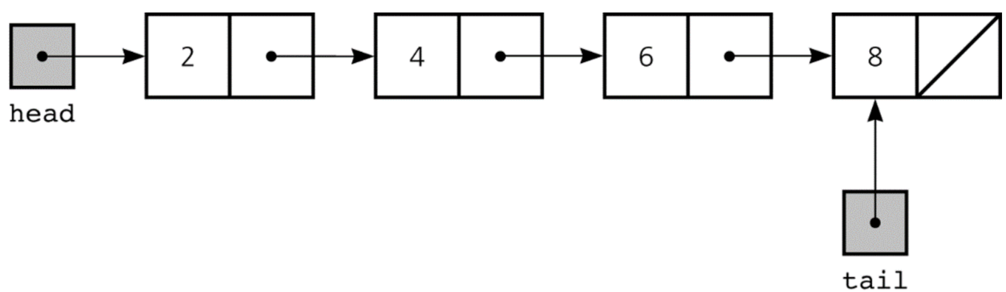
LINKED LIST – LL VARIATIONS 1

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.

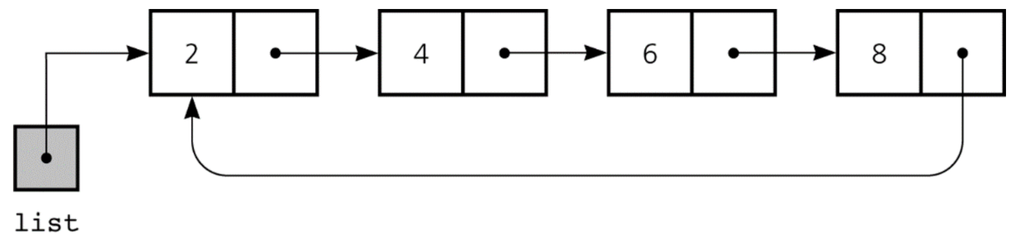
```
tail.next = new Node( item, null ); // Add a new node at the end of the list.
tail = tail.next; // Update tail so that it references the new last node.
```



LINKED LIST – LL VARIATIONS 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).



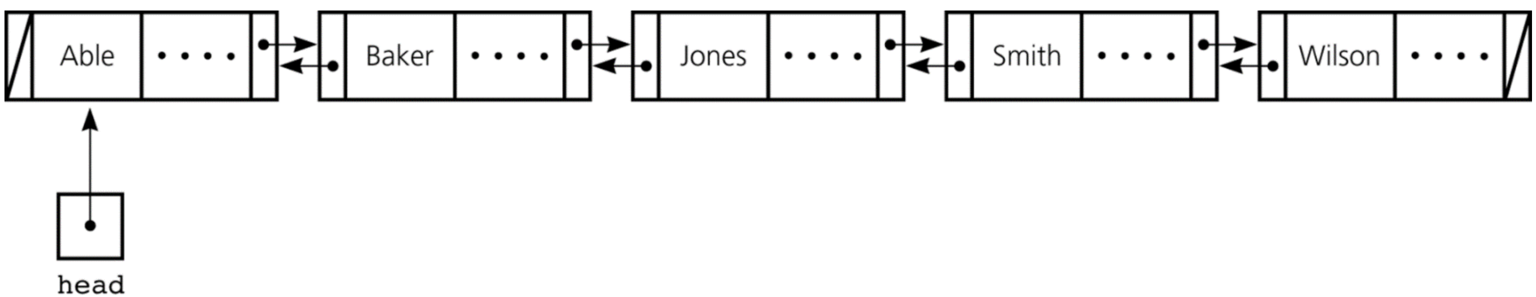

```
// Display the data in a circular linked list, where "list" references the last node.
if( list != null ) {
   Node first = list.next; // Get the reference to the first node.
   Node curr = first; // Start at first node.
   do { System.out.println( curr.item ); // Display node data.
        curr =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

LINKED LIST – LL VARIATIONS 3

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.



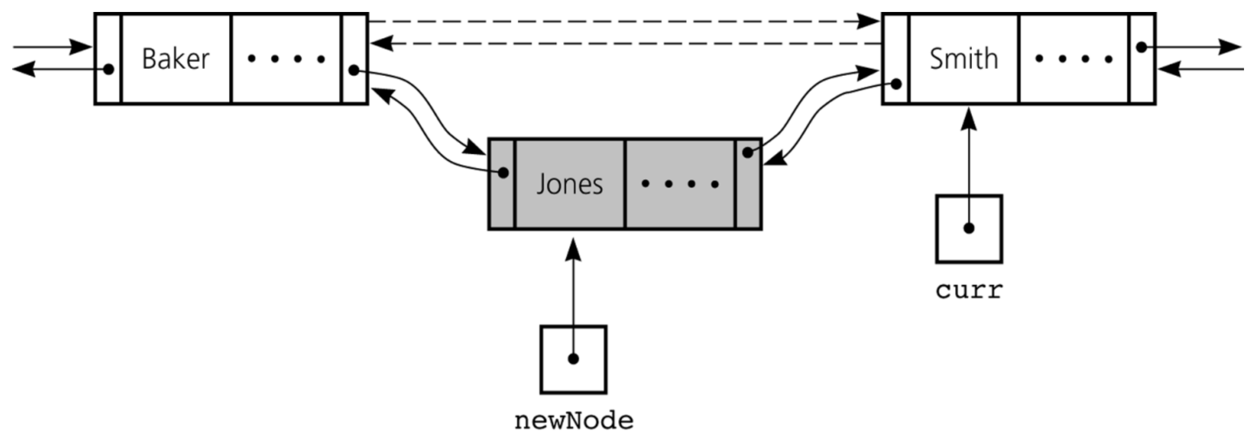
See: visualgo.net/en/list

LINKED LIST – LL VARIATIONS 4

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.
```



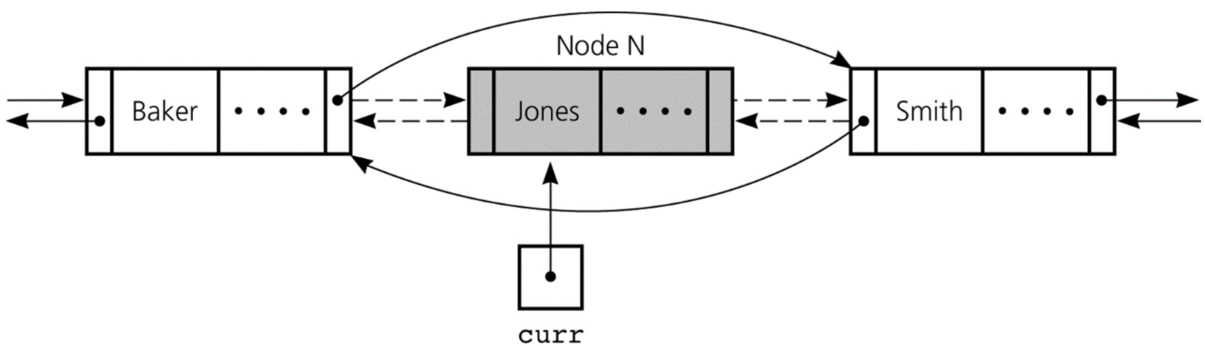
See: visualgo.net/en/list

LINKED LIST – LL VARIATIONS 5

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

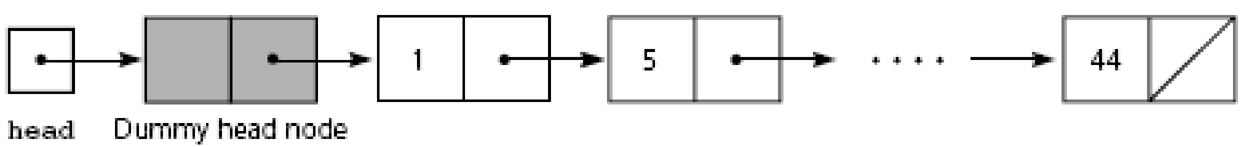
LINKED LIST – LL VARIATIONS 6

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:

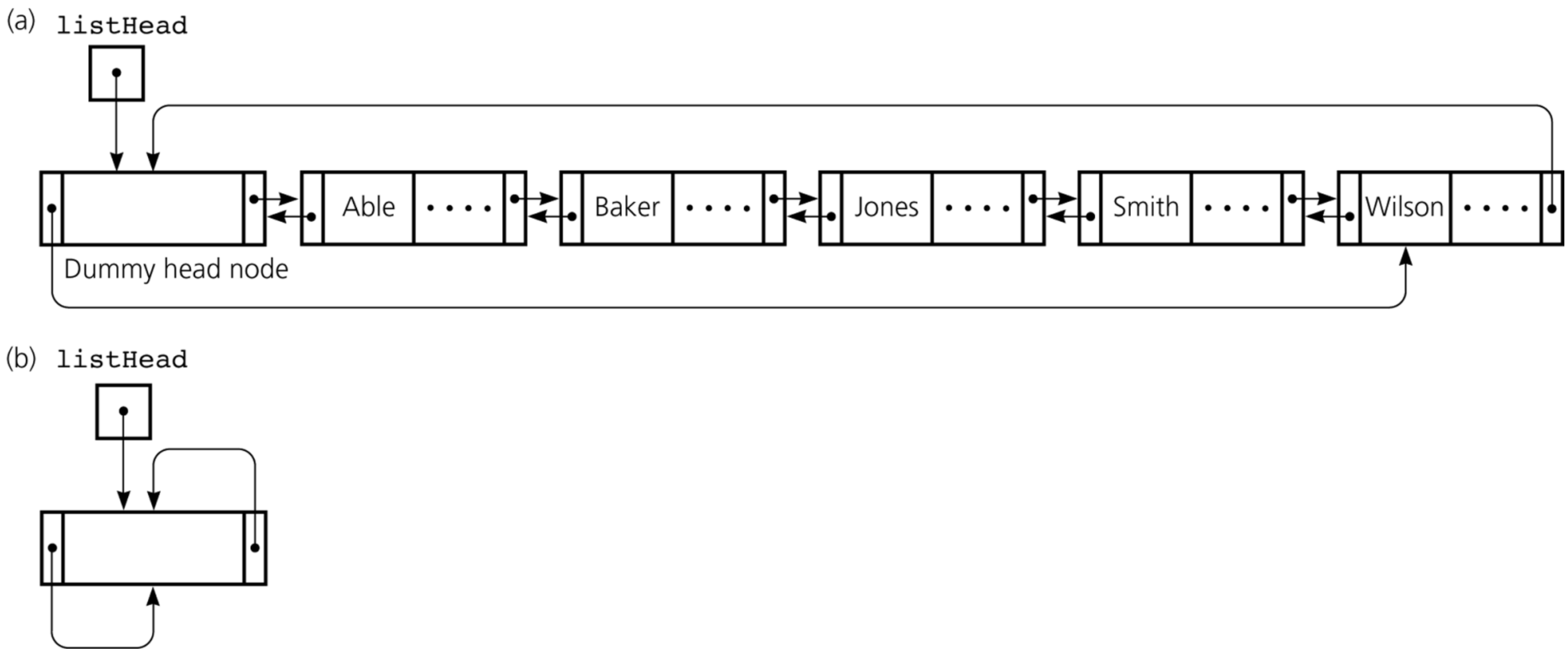
- the dummy head node is always present (even when the linked list is empty),
- insert-delete init **prev** to reference the dummy head node (rather than **null**).

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



LINKED LIST – LL VARIATIONS 7

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



JCF - INTRODUCTION

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

JCF - 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; } }
```

JCF - ITERATORS AND ITERABLE COLLECTIONS 1

- 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.
   ... }
```

JCF - ITERATORS AND ITERABLE COLLECTIONS 2

- 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.
```

JCF - ITERATORS AND ITERABLE COLLECTIONS 3

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

JCF - ITERATORS AND ITERABLE COLLECTIONS 4

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)!

JCF - ITERATORS AND ITERABLE COLLECTIONS 5

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).
```

JCF - ITERATORS AND ITERABLE COLLECTIONS 6

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.
   ... }
```

JCF - ITERATORS AND ITERABLE COLLECTIONS 7

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

JCF - THE ARRAYLIST CLASS

The **ArrayList** class is a resizable-array implementation of the interface **List**.
The following is an example of how to use the **ArrayList** class.

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

```
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 ); }
```

Kettering
UNIVERSITY