

Special Topics No. 6: Linked Lists and Dynamic Allocation

In this document, we will introduce a data structure called a *linked list*. A linked list consists of nodes where each node contains an element, and nodes are linked in a chain. With an array or a vector, memory for the whole structure is allocated at once. With a linked list, memory for nodes is allocated individually. For example, we allocate memory for an array of 20 elements at once, but we allocate memory for 20 nodes in a linked list one by one. The chaining of nodes and the finer granularity of memory allocation of the linked list result in a high performance for certain types of operations. We will describe how to implement linked lists and compare the performances of linked lists and arrays.

Introduction

In the java.util package, there is an interface called List and two classes—ArrayList and LinkedList—that implement the List interface. In this document, we will explain the core concepts behind them. We will define our own two classes that correspond to ArrayList and LinkedList and show how some of the key methods in the List interface can be implemented. (Note: The concept of interface is explained briefly on page 201 and defined on page 612. An interface is a class that includes only constants and method prototypes, the methods with header but no method body. As such, no instances can be created from an interface.)

1 The List ADT

concrete data type

An array is probably the most fundamental data structure in any programming language. We can characterize the array as a *concrete data type* because it is included in the language definition. Classes in an object-oriented programming language, on the other hand, are not a concrete data type because a class is something that is built from the core elements included in the programming language. These object-oriented programming languages provide a mechanism to define classes, but the individual classes themselves, such as Vector, Button, Frame, Applet, etc., are not part of the Java language specification.

abstract data type

As we develop more and more complex problems, concrete data types alone do not provide sufficiently high abstraction. Using object-oriented programming languages, we define classes that provide a higher abstraction. For example, instead of forcing the programmers to use an array and several variables separately, we integrate them together to form a single AddressBook object. We can characterize such a class as an *abstract data type*.

data structure classes

In solving complex problems, it helps to use abstract data types because they provide a higher abstraction so the programmer can think about the problem without being bogged down with the lower-level details. This is especially true for the classes that support generic data maintenance operations, which we call *data structure classes*. Note that a class such as AddressBook does support “generic” data maintenance operations, but the class is customized to the maintenance of Person objects. Data structure classes support generic data maintenance operations so that the programmers can reuse them to build more customized classes for their programs.

list

In the standard java.util package, there are numerous data structure classes that implement very common data structures we study in computer science. We will examine one of the most fundamental data structures called a *list* in this document. A *list* is a linearly ordered collection of elements. Here’s an example of a list name myList that contains five elements:

```
myList = ( L0, L1, L2, L3, L4 )
```


A list is a linearly ordered collection, so L_0 comes before L_1 and L_1 comes before L_2 , and so forth. The first element is L_0 , and the last element is L_4 . The subscript denotes the position of an element. Notice that we are using the zero-based indexing here.


What are the kinds of operations we would like to perform on a list? Here we will go over some of the key operations defined in the `java.util.List` interface. Please read its documentation for full details. To illustrate the operations, we will use the following list of three-lettered animal names:

```
sample = ( cat, ape, dog, bee, eel )
```

add

There are two versions: one will add a new element at the end of the list and another will add a new element at the designated position.


```
sample.add( gnu )       ( cat, ape, dog, bee, eel, gnu )
```

```
sample.add( 2, gnu )       ( cat, ape, gnu, dog, bee, eel )
```

In the second version, notice that the elements at positions 2 to 4 in the original list are shifted right by one position to open up a place to insert a new element.

clear


This operation removes all elements in the list. The list becomes empty.


```
sample.clear( )       ( )
```

contains

This operation tests whether a specified element is in the list or not. If it is, then `true` is returned. Otherwise, `false` is returned. Notice that there could be duplicate objects. Two objects `o1` and `o2` are considered duplicates if `o1.equals(o2)` is

true. if The search will stop immediately after the first match and true is returned. We say

`sample.contains(cow)`  `false`

`sample.contains(ape)`  `true`

get

This operation returns the element at the designated position. If the given position is outside of a valid range, a null is returned. Note: The `get` method of `java.util.ArrayList` will throw an exception in the case of invalid position value.


`sample.get(3)`  `bee`


`sample.add(8)`  `null`

The `get` operation is a read-only operation. In other words, it does not remove the element, so the list remains the same after the `get` operation.

indexOf

This operation returns the position of the specified object in the list. If the specified object is not in the list, then `NOT_FOUND` (-1) is returned. If there are duplicates, then the position of the first match is returned.

`sample.indexOf(cat)`  `0`

`sample.indexOf(yak)`  `NOT_FOUND`

isEmpty

This operation tests whether the list is empty or not. If it is, then true is returned. Otherwise, false is returned.

```
sample.isEmpty( )       false
```

remove

There are two versions: one will remove the element by specifying its position and another will remove the element by specifying the element. The first version returns the index-out-of-bound exception for the invalid index value. The second version removes the element and returns true. If the designated element is not found, then the list remains the same and false is returned. If the duplicates exist, then the first element in the list is removed.

```
sample.remove( 2 )       ( cat, ape, bee, eel )
```

```
sample.remove( ape )       ( cat, dog, bee, eel )
```

In both versions, all elements to the right of the removed element are shifted one position to the left. For example, after the `ape` was removed, `dog`, which was at position 3, is now at position 2.

set

This operation replaces the element at the designated position with the designated element. If the given index is outside of a valid range, nothing will happen. Note: The `set` method of `java.util.ArrayList` will throw an exception in the case of invalid position value.

```
sample.set( 2, fox )       ( cat, ape, fox, bee, eel )
```

size

This operation returns the size of the list.

```
sample.size( )       5
```

2 The Array Implementation

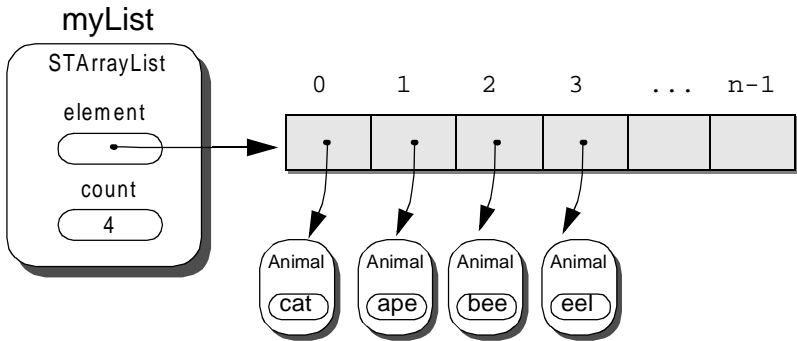
Now that we have the specification of the abstract data type List, we are ready to consider its implementation. There are two basic approaches in implementing the List—array and linked implementations. The `java.util` package already includes a class called `ArrayList` that implements the List interface using an array. Please refer to the Java API documentation for the details on how to use the `ArrayList` class. We will not discuss the `ArrayList` class here. Instead, we will implement a simplified version to present just the core ideas behind the array implementation.

To make the illustration concrete, we assume a class called `Animal` whose instances are the elements we add to a list. Figure 1 illustrates the array implementation of a list.

FIGURE 1 An array implementation of the list `myList`.

ADT List: `myList = (cat, ape, bee, eel)`

Array Implementation:



To avoid a confusion with `java.util.ArrayList`, we will name our class `STArrayList` (ST stands for Special Topics). Also, please note that some methods we implement here do not match the methods in the `java.util.ArrayList` class exactly. To simplify our presentation, the return type of some methods is modified here.



To keep the implementation clear and simple, the methods of `STArrayList` are not implemented exactly as the corresponding methods of the `java.util.ArrayList` class. Please refer to the Java API documentation for a precise definition of the `java.util.ArrayList` methods.

The STArrayList Class

We need an array to store the elements and an int variable to keep track of the number of elements currently in the list. These members are declared as

```
private Object[ ] element;
private int      count;
```

and initialized as

```
count = 0;
element = new Object[ DEFAULT_SIZE ];
```

in a constructor. We assume the class constant `DEFAULT_SIZE` is assigned some arbitrary integer.

add

Adding an element at the end of the list can be achieved very easily. You add a new element at the `count` position and increment the `count` by one. The only complication is the overflow condition. The `enlarge` method we used in the `AddressBook` class from Chapter 9 can be reused here. The overflow condition occurs when the value for `count` is equal to the length of the `element` array. Here's the Version 1 `add` method:

```
public void add( Object item )
{
    if ( count == element.length ) {
        enlarge( );
    }
    element[count] = item;
    count++;
}
```

See the companion source code for the full definition of the **enlarge** methods

The second version of the `add` method requires more effort. To insert an element at position `i`, we must shift current elements at positions `i` to `count-1` one position to the right to positions `i+1`, ..., `count`, respectively. If an invalid value is given for the position, then nothing will happen (the `ArrayList` will throw an exception).

```

public void add( int index, Object item )
{
    if (index >=0 && index <= count ) {

        if ( count == element.length ) {
            enlarge( );
        }

        //shift one position to the right
        for (int i = count; i > index; i--) {
            element[i] = element[i-1];
        }

        element[ index ] = item;
        count++;
    }
}

```

clear

We will set `element[i]` to null, where `i = 0, ..., count-1`, so the objects referenced by `element[i]` will be garbage collected. Finally, we reset the `count` variable to 0.

```

public void clear( )
{
    for (int i = 0; i < count; i++) {
        element[i] = null;
    }

    count = 0;
}

```

contains

This method is implemented by using the `indexOf` method. If the `indexOf` method returns a value other than `NOT_FOUND` (-1), then the specified object is in the list, so the `contains` method returns `true`. If the `indexOf` method returns `NOT_FOUND`, then the `contains` method returns `false`.

```

public boolean contains( Object item )
{
    boolean result = true;

    int loc = indexOf( item );

    if ( loc == NOT_FOUND ) {
        result = false;
    }
}

```



```

    }

    return result;
}

```

get

The method is straightforward.

```

public Object get( int index )
{
    Object    item = null;

    if ( index >= 0 && index < count ) {
        item = element[index];
    }

    return item;
}

```

indexOf

This method scans the list from the beginning and stops when the first match is located. If the specified object is not in the list, then NOT_FOUND (-1) is returned. For this method to work properly, the element object must support the equals method, which we will use to locate the matching element.

```

public int indexOf( Object item )
{
    int    loc    = 0;

    while ( loc < count &&
           !element[loc].equals( item ) ) {
        loc++;
    }

    if (loc == count) {
        loc = NOT_FOUND;
    }

    return loc;
}

```

isEmpty

This method is also straightforward. We just check the value of the count variable.

```

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

```

remove

After the element at position *i* is removed, we must shift current elements at positions *i*+1 to count one position to the left to positions *i*, ... , *count*-1, respectively. And we decrement the count variable by one.

```

public void remove( int index )
{
    if (index >=0 && index < count ) {

        //shift one position to the left
        for (int i = index; i < count; i++) {
            element[i] = element[i+1];
        }

        count--;
    }
}

```

The second version of **remove** requires a routine to find the index of the item to be removed. If there are duplicates then the first match is removed. If the passed item is not found, then nothing happens.

```

public void remove( Object item )
{
    int loc = indexOf( item );

    if (loc >= 0 && loc < count ) {

        remove( loc );
    }
}

```

set

This method works just like the **get** method, only direction of data flow is reversed.

```

public void set( int index, Object item )
{

```

```

        if ( index >= 0 && index < count ) {
            element[index] = item;
        }
        //do nothing if index is out of valid range
    }

```

size

This method is straightforward. We simply return the value of the count variable.

```

    public int size(    )
    {
        return count;
    }

```

We are now ready to list the complete STArrayList class. A sample main class to test the methods of the STArrayList class is provided after the class listing.

File: [STArrayList.java](#)

```

/**
 * Introduction to OOP with Java 2nd Edition, McGraw-Hill
 *
 * <p>
 * Special topics No. 6 Linked List
 *
 * <p>
 * This class implements a simplified version of java.util.List
 * interface using an array. Some of implemented methods are slightly
 * modified version of the corresponding methods of
 * java.util.ArrayList.
 *
 * @author Dr. Caffeine
 */

class STArrayList
{
    //-----
    //      Data Members
    //-----

    /**
     * Default initial size of an array
     */
    public static final int DEFAULT_SIZE = 25;

    /**
     * Flag for not finding the specified element
     */
    public static final int NOT_FOUND = -1;

    /**

```

```

    * An array to store the elements
    */
private Object[] element;

/**
 * The number of elements in the list
 */
private int count;

//-----
// Constructors
//-----

/**
 * Default constructor
 */
public STArrayList( )
{
    this( DEFAULT_SIZE );
}

/**
 * Default constructor
 */
public STArrayList( int size )
{
    element = new Object[ size ];
    count = 0;
}

//-----
// Public Methods:
//-----
// void add ( int, Object )
// void add ( Object )
// void clear ( )
// void contains ( )
// Object get ( int )
// int indexOf ( Object )
// boolean isEmpty ( )
// void remove ( int )
// void remove ( Object )
// void set ( int, Object )
// int size ( )
//-----

/**
 * Adds an item to end of the list. Overflow condition
 * is handled automatically.
 *
 * @param item an item to add.
 */

```

```

    */
    public void add( Object item )
    {
        if ( count == element.length ) {
            enlarge( );
        }

        element[count ] = item;
        count++;
    }

    /**
     * Adds an item to the list at index. Overflow condition
     * is handled automatically.
     *
     * @param index the position the item is added
     * @param item  an item to add.
     */
    */
    public void add( int index, Object item )
    {
        if (index >=0 && index <= count ) {
            if ( count == element.length ) {
                enlarge( );
            }

            //shift one position to the right
            for (int i = count; i > index; i--) {
                element[i] = element[i-1];
            }

            element[ index ] = item;
            count++;
        }
    }

    /**
     * Resets the list to be an empty list.
     */
    */
    public void clear( )
    {
        for (int i = 0; i < count; i++) {
            element[i] = null;
        }

        count = 0;
    }

    /**
     * Determines if the passed item in the list
     * or not. If it is, return true. Otherwise,
     * return false;
     */
    */
    public boolean contains( Object item )
    {
        boolean result = true;

```

```

        int      loc      = indexOf( item );

        if ( loc == NOT_FOUND ) {
            result = false;
        }

        return result;
    }

    /**
     * Retrieves the object at the index position.
     * If index is invalid, null is returned.
     */
    public Object get( int index )
    {
        Object      item = null;

        if ( index >= 0 && index < count ) {
            item = element[index];
        }

        return item;
    }

    /**
     * Finds the index position of the specified item.
     * If the item is not found, then NOT_FOUND is
     * returned.
     */
    public int indexOf( Object item )
    {
        int      loc      = 0;

        while ( loc < count &&
                !element[loc].equals( item ) ) {
            loc++;
        }

        if (loc == count) {
            loc = NOT_FOUND;
        }

        return loc;
    }

    /**
     * Determines whether the list is empty or not.
     *
     * @return true if the list is empty; false otherwise.
     */
    public boolean isEmpty( )
    {
        return ( count == 0 );
    }

    /**
     * Removes the element at the index position.
     */

```

```

    * @param index the position of the element to be removed
    */
    public void remove( int index )
    {
        if (index >=0 && index < count ) {

            //shift one position to the left
            for (int i = index; i < count; i++) {
                element[i] = element[i+1];
            }

            count--;
        }
    }

    /**
     * Removes the designated element
     *
     * @param item the item to remove from the list
     */
    public void remove( Object item )
    {
        int loc = indexOf( item );

        if (loc >= 0 && loc < count ) {

            remove( loc );
        }
    }

    /**
     * Replaces the current element at the index position
     * with the passed element.
     *
     * @param index the position of the object to be replaced
     * @param item the new item to replace the current element
     */
    public void set( int index, Object item )
    {
        if ( index >= 0 && index < count ) {
            element[index] = item;
        }
        //do nothing if index is out of valid range
    }

    /**
     * Returns the number of elements in the list
     *
     * @return the number of elements in the list
     */
    public int size( )
    {
        return count;
    }

    //-----
    // Private Methods:

```

```
//
//      void      enlarge      (      )
//
//-----

/**
 * Enlarges the size of the array to
 * eliminate the overflow condition. The new array
 * is 50 percent larger than the current array.
 */
private void enlarge( )
{
    //create a new array whose size is 150% of
    //the current array
    int newLength = (int) (1.5 * element.length);
    Object[] temp = new Object[newLength];

    //now copy the data to the new array
    for (int i = 0; i < element.length; i++) {
        temp[i] = element[i];
    }

    //finally set the variable entry to point to the new array
    element = temp;
}
}
```

And here's a simple test program to verify the methods of the STArrayList class.

```
import javabook.*;

class TestSTArrayList
{
    public static void main (String[] args)
    {
        MainWindow      mainWindow;
        OutputBox        outputBox;
        InputBox         inputBox;

        mainWindow = new MainWindow
                        ("Test the ArrayList Operations");
        outputBox  = new OutputBox( mainWindow );
        inputBox   = new InputBox ( mainWindow );

        mainWindow.setVisible( true );
        outputBox.setVisible( true );

        STArrayList myList = new STArrayList( );
        Animal myPet;
```



```

Animal ape = new Animal("ape");
Animal bee = new Animal("bee");
Animal cat = new Animal("cat");
Animal dog = new Animal("dog");
Animal eel = new Animal("eel");

myList.add( cat );
myList.add( eel );

myList.add( 1, ape );
myList.add( 2, bee );

for (int i = 0; i < myList.size(); i++ ) {
    myPet = (Animal) myList.get(i);
    outputBox.println( myPet.getName() );
}

outputBox.skipLine( 2 );

if ( myList.contains( ape ) ) {
    outputBox.println( "ape is in the list" );
}

if ( !myList.contains( dog ) ) {
    outputBox.println( "dog is not in the list");
}

myList.set( 2, dog );
if ( myList.contains( dog ) ) {
    outputBox.println( "dog is now in the list");
}

myList.remove( 1 );
myList.remove( eel );

for (int i = 0; i < myList.size(); i++ ) {
    myPet = (Animal) myList.get(i);
    outputBox.println( myPet.getName() );
}
}
}

```

Running the program will produce the output shown below. Make sure you walk through the program and agree with the output before actually running the program.



```

cat
ape
bee
eel

ape is in the list
dog is not in the list
dog is not in the list
cat
dog
  
```

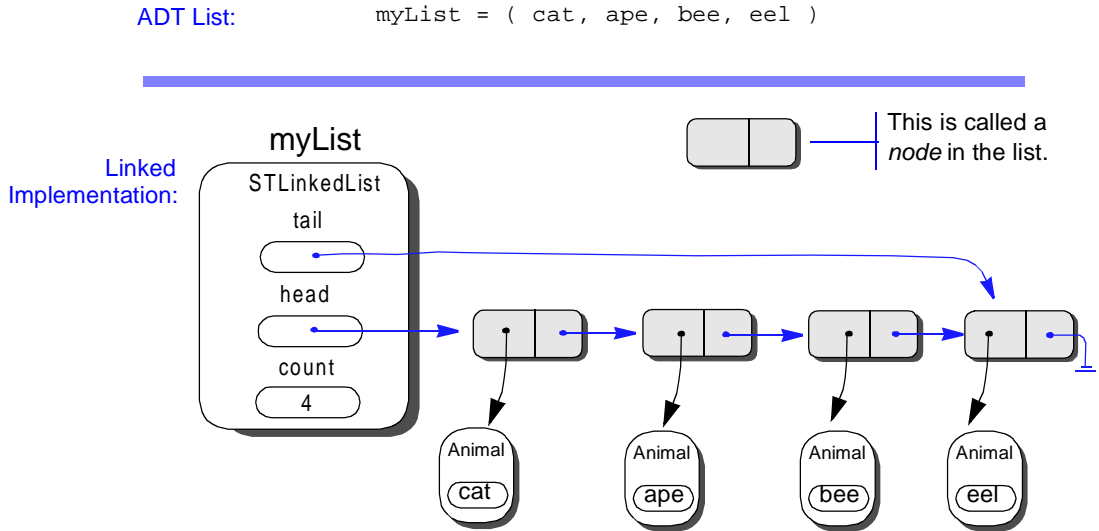
3 The Linked Implementation

With the array implementation, a whole block of memory is allocated at once. When the overflow conditions occur, a new larger block of memory is allocated, and the contents of the original array are copied to the new array. The original array will eventually get garbage collected when the variable is set to refer to the new array. Instead of allocating a large block of memory, the linked implementation allows us to allocate a small amount of memory that is just large enough for a single element. This finer granularity of memory allocation leads to a better performance for the operations such as adding to or removing nodes from a list. In this section, we will describe how the linked implementation works. Figure 2 illustrates the linked implementation of a list.

To avoid a confusion with `java.util.LinkedList`, we will name our class `STLinkedList` (ST stands for Special Topics). Also, please note that some methods we implement here do not match the methods in the `java.util.LinkedList` class exactly. To simplify our presentation, the return type of some methods is modified here.

To keep the implementation clear and simple, the methods of `STLinkedList` are not implemented exactly as the corresponding methods of the `java.util.LinkedList` class. Please refer to the Java API documentation for a precise definition of the `java.util.LinkedList` methods.



FIGURE 2 A linked implementation of the list **myList**.

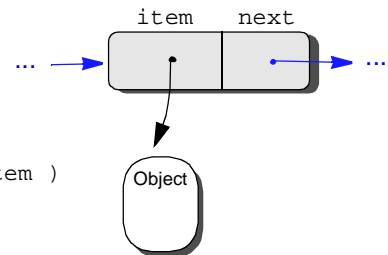
The ListNode Class

The first task for us to solve is the structure we use to represent a single node in the linked list. Notice that there are two components in each node: a reference to the data item and a reference to another instance of **ListNode**. To distinguish the two different types of references, we use a black arrow for the first reference and a blue arrow for the second reference. Each node will be represented by a single **ListNode** object. The **ListNode** class has two data members, and the class is declared as

```
class ListNode
{
    private Object item;

    private ListNode next;

    public ListNode( Object item )
    {
        this.item = item;
        this.next = null;
    }
}
```



data field
link field

The item data member is called a *data field* because it contains a data item and the next data member is called a *link field* because it links the list nodes.

The STLinkedList Class

Similar to the STArrayList class, we need an int variable to keep track of the number of elements currently in the list. In addition, we need two references, one refers to the first node and the other refers to the last node. We use the reference to the last node to make the addition of a new node at the end of the list easy. The traditional term for a reference that refers to the node in a list is a *pointer*, which we will use in the following discussion. We will restrict the use of the term *pointer* to the references to the list nodes. We will continue to use the term *reference* for the references to the data item objects to differentiate the two. These data members are declared as

```
private ListNode    head;
private ListNode    tail;
private int         count;
```

and initialized as

```
head  = null;
tail  = null;

count = 0;
```

in a constructor (the actual constructor calls the `clear` method, which resets the list as an empty list by executing the above three statements).

add

There are two cases we need to consider when adding an element at the end of the list. The first case is adding an element to an empty list. Figure 3 illustrates this case. Both `head` and `tail` are null when the list is empty. When a node is added to an empty list, this node is both the first and last node of the list, so both `head` and `tail` are set to point to this newly added node. This is the end case.

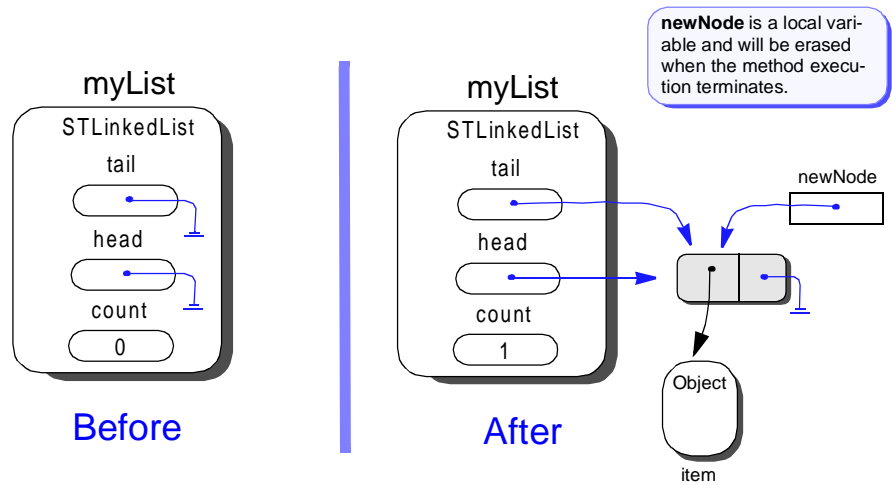
In the general case, where the list has one or more nodes, a newly added node is added at the end so this new added node becomes the new tail node of the list. The pointer from the tail node, before the addition, is set to point to the new tail node, after the addition. Figure 4 illustrates the general case.

Here's the Version 1 `add` method:

```
public void add( Object item )
```

FIGURE 3

Adding an element at the end of an empty list.



```

ListNode newNode = new ListNode( item );

head = newNode;

tail = newNode;

count++;

```

```

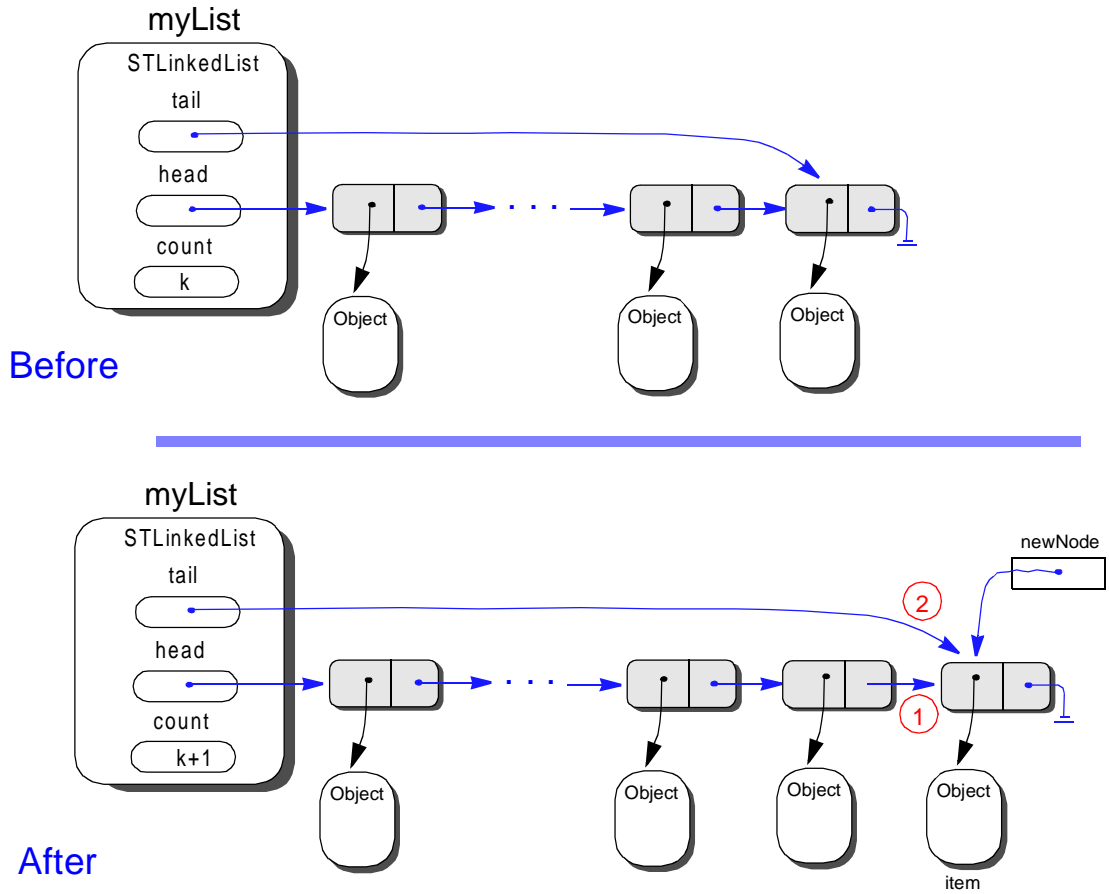
{
    //creates a new ListNode
    ListNode newNode = new ListNode( item );

    if ( count == 0 ) {
        head = tail = newNode;
    }
    else {
        tail.next = newNode;
        tail = newNode;
    }

    count++;
}

```

The second version of the add method requires more effort. Again we have to handle the two cases separately. The end case is when the new node is added

FIGURE 4 The general case of adding an element at the end of a list.

```

ListNode newNode = new ListNode( item );

tail.next = newNode; ①

tail      = newNode; ②

count++;

```

as the first element, that is, `index == 0`. In this case we have to adjust the head to point to the newly added node and the link field of this new node to point to the node that was the first node before the addition. Figure 5 illustrates this case.

In the general case, we must locate the position to insert the new node as the *i*'th node. Unlike the array implementation where we can locate an element at position *i* by simply using an indexed expression, in the case of linked implementation we must traverse the pointers in the linked fields. To insert a new node as the *i*'th node in the list (the first node being the 0 node), we need a pointer to the *i*-1st node. Once we locate this node, the rest is a matter of adjusting two link fields. Figure 6 illustrates the general case.

Here's the complete definition for the second version of the add method:

```
public void add( int index, Object item )
{
    if (index >= 0 && index <= count ) {

        ListNode ptr = head;

        ListNode newNode = new ListNode( item );

        if ( index == 0 ) { //adding the first node

            newNode.next = head;
            head = newNode;
        }
        else { //the general case

            //set ptr points to the i-1st node
            for (int i = 1; i < index; i++) {
                ptr = ptr.next;
            }

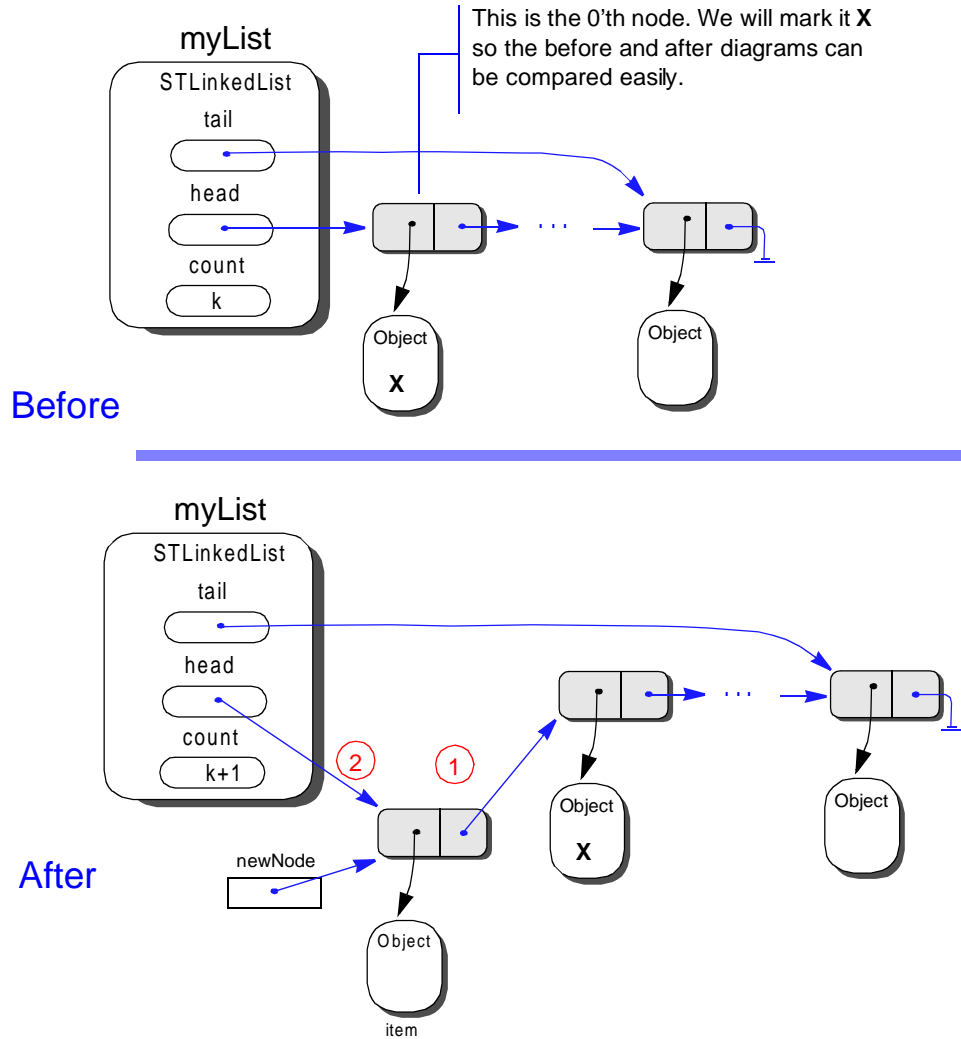
            newNode.next = ptr.next;
            ptr.next      = newNode;
        }

        //adjust tail if the new node added is
        //the last node in the list
        if ( index == count ) {
            tail = newNode;
        }
        count++;
    }
}
```

Notice the last if statement in the method. If the newly added node is the last node after the insertion, then we must adjust the tail pointer. We know that the

FIGURE 5

The second version of add with **index == 0**, i.e., adding an element as the first node in a list.



```

ListNode newNode = new ListNode( item );

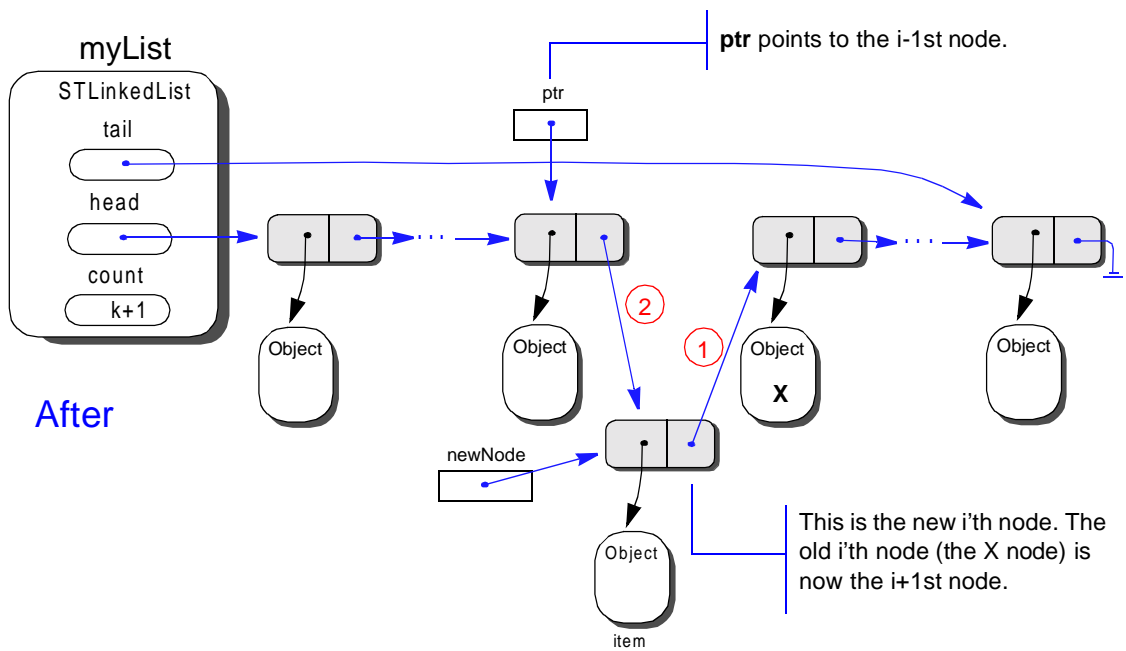
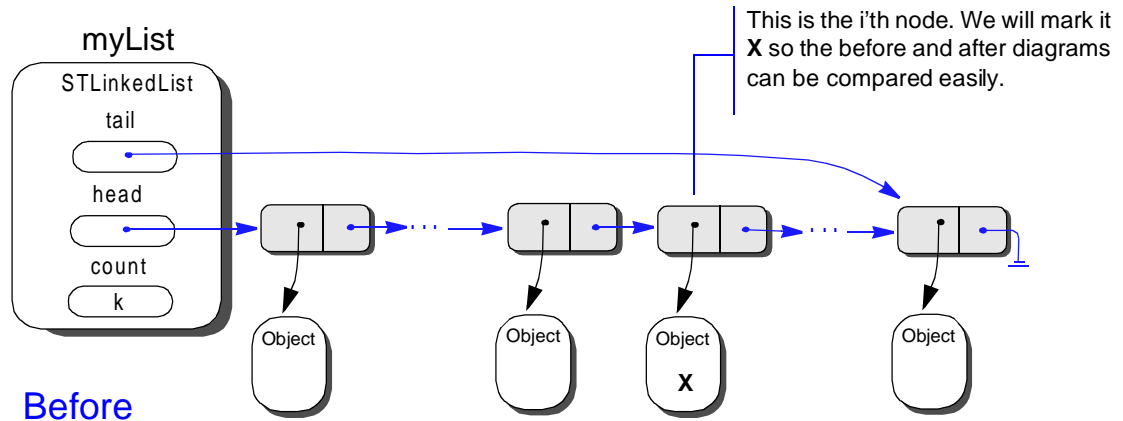
newNode.next = head;           ①

head = newNode;                ②

count++;

```


FIGURE 6 The general case of adding a new node as the i 'th node.



```
ListNode newNode = new ListNode( item );  
newNode.next = ptr.next; ①  
ptr.next      = newNode; ②  
count++;
```

newly added node is the last node if its index is equal to count. If the test is true, we set tail points to the new node.

clear

This method empties the list, that is, it resets the list as an empty list. Its implementation is straightforward. We reset head and tail to null and count to 0. By setting head to null, all the nodes in the list will eventually get garbage collected. Here's the method:

```
public void clear( )
{
    head = tail = null;

    count = 0;
}
```

contains

This method is implemented by using the indexOf method. If the indexOf method returns a value other than NOT_FOUND (-1), then the specified object is in the list, so the contains method returns true. If the indexOf method returns NOT_FOUND, then the contains method returns false. The method implementation is the same for both STArrayList and STLinkedList.

```
public boolean contains( Object item )
{
    boolean result = true;

    int    loc    = indexOf( item );

    if ( loc == NOT_FOUND ) {
        result = false;
    }

    return result;
}
```

get

The get method for the linked implementation requires the traversing of link fields to locate the desired node. The traversing is essentially the same as we did for the second version of the add method. The only difference is that we are setting the pointer ptr points the index'th node, not the index-1st node as was the case with the add method. Here's the method:

This loop traverses the list by following the link field for index times.

```

public Object get( int index )
{
    Object      item = null;

    ListNode    ptr  = head;

    if (index >= 0 && index < count) {
        for (int i = 0; i < index; i++) {
            ptr = ptr.next;
        }

        item = ptr.item;
    }

    return item;
}

```

indexOf

This method traverses the links from head and stops when the first match is located. The counter loc is increment as every time the next node is visited. If the specified object is not in the list, then NOT_FOUND (-1) is returned.

```

public int indexOf( Object item )
{
    int      loc      = 0;

    ListNode ptr      = head;

    while ( loc < count && !ptr.item.equals( item ) ) {

        loc++;
        ptr = ptr.next;
    }

    if (loc == count) {

        loc = NOT_FOUND;
    }

    return loc;
}

```

isEmpty

This method is straightforward. We just check the value of the count variable.

```

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

```

remove

For the first version of **remove**, we must locate the *i*-1st node. Once this node is located, all that is left to do is to adjust this node's *next* field. Figure 7 illustrates the operation.

```

public void remove( int index )
{
    if (index >=0 && index < count ) {

        ListNode ptr = head;

        for (int i = 1; i < index; i++) {
            ptr = ptr.next;
        }

        ptr.next = ptr.next.next;

        if (ptr.next == null) { //very last node was
            tail = ptr;        //removed, we have a
                               //new last node
        }

        count--;
    }
}

```

The second version of **remove** requires a traversal to locate the node to be removed. If there are duplicates then the first match is removed. If the passed item is not found, then nothing happens. To locate the desired node, we use two pointers *ptr* and *trail*. The *ptr* will point to the node to be removed, and the *trail* will point to the node one before the *ptr* node. In other words, the following relationship holds between the two:

```

trail.next == ptr

```

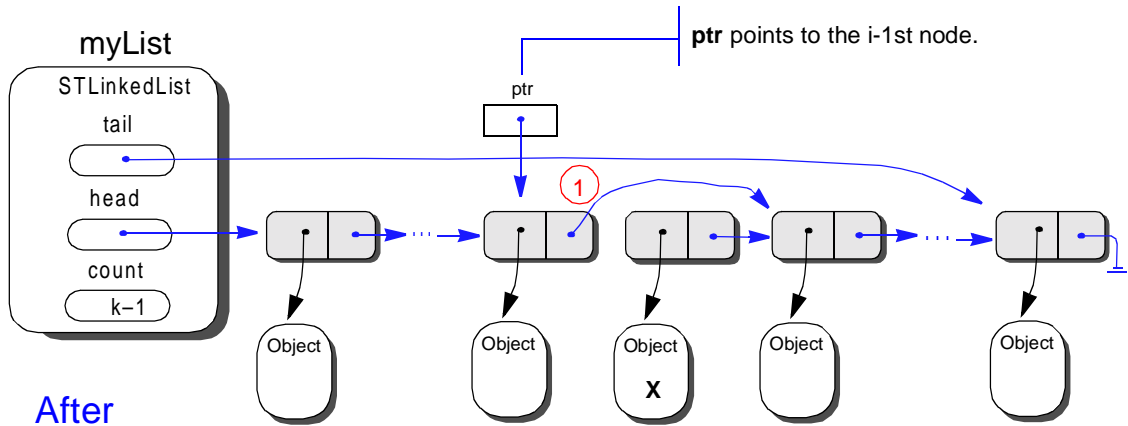
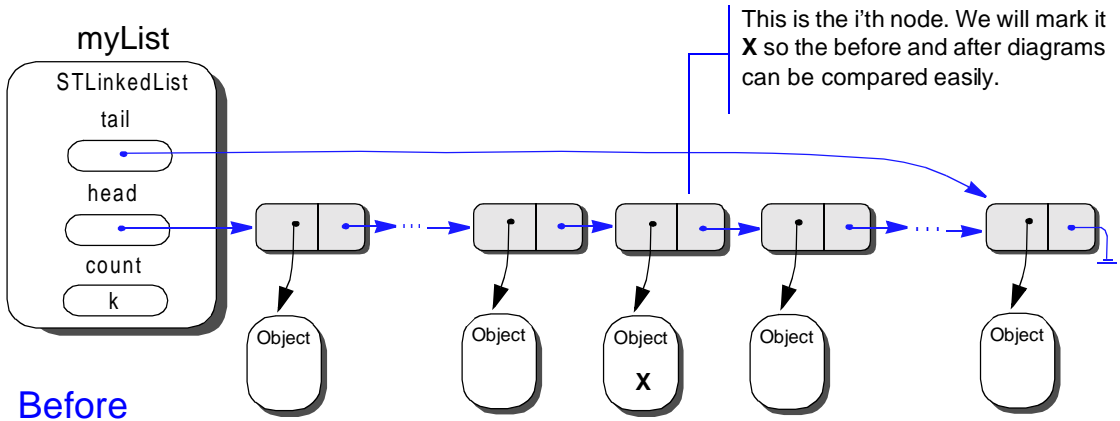
Figure 8 illustrates how the second version of **remove** works.

```

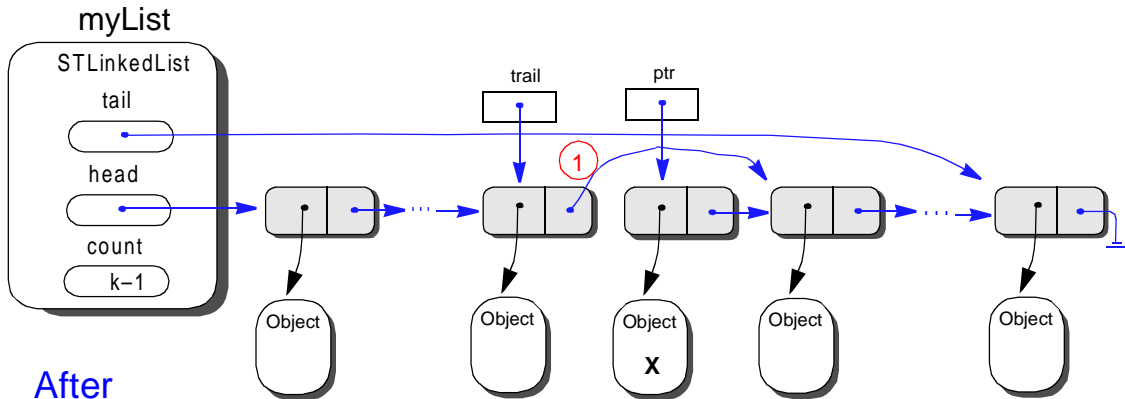
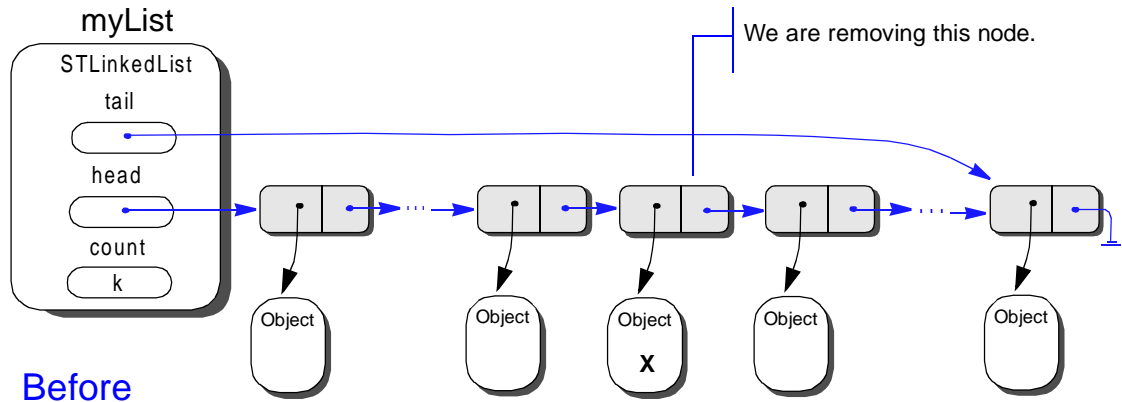
public void remove( Object item )
{
    ListNode ptr = head;

```

FIGURE 7 The first version of the removal operation. Remove the i 'th node given the value i .



```
//set ptr to i-1st node
ptr.next = ptr.next.next; 1
count--;
```

FIGURE 8 The second version of the removal operation.

```
//set ptr to the node to be removed
//and trail to one node before

trail.next = ptr.next; 1
count--;
```

```

ListNode trail = null;

while ( ptr != null && !ptr.item.equals( item ) ) {

    trail = ptr;
    ptr    = ptr.next;
}

if ( ptr != null ) {

    trail.next = ptr.next;
}

if ( trail.next == null ){ //very last node was
    tail = trail;          //removed we have a
                           // new last node
}

count--;
}

```

Notice the last if statement in the method. If the deleted node was the last node, then we must adjust the `tail` pointer. We know that the deleted node was the last node if its link field was null.

set

This method traverses the list for `index` times and sets the pointer `ptr` to the node to be modified. Once this node is located, its data field is set to refer to the passed `item`.

```

public void set( int index, Object item )
{
    if ( index >= 0 && index < count ) {

        ListNode ptr = head;

        for (int i = 0; i < index; i++) {
            ptr = ptr.next;
        }

        ptr.item = item;
    }
    //do nothing if index is out of valid range
}

```

size

This method is straightforward. We simply return the value of the count variable.

```
public int size(    )
{
    return count;
}
```

Here's the complete STLinkedList class:

File: [STLinkedList.java](#)

```
/**
 * Introduction to OOP with Java 2nd Edition, McGraw-Hill
 *
 * <p>
 * Special topics No. 6 Linked List
 *
 * <p>
 * This class implements a simplified version of java.util.List
 * interface using linked nodes. Some of implemented methods are
 * slightly modified version of the corresponding methods of
 * java.util.LinkedList.
 *
 * @author Dr. Caffeine
 */

class STLinkedList
{
    //-----
    //    Data Members
    //-----

    /**
     * Flag for not finding the specified element
     */
    public static final int NOT_FOUND = -1;

    /**
     * The pointer to the first node in the list
     */
    private ListNode head;

    /**
     * The pointer to the last node in the list
     */
    private ListNode tail;

    /**
     * The number of elements in the list
     */
    private int count;

    //-----
}
```



```

// Constructors
//-----

/**
 * Default constructor
 */
public STLinkedList( )
{
    clear( );
}

//-----
//      Public Methods:
//
//      void      add      ( int, Object )
//      void      add      ( Object      )
//
//      void      clear     (              )
//      void      contains  (              )
//
//      Object    get       ( int          )
//
//      int       indexOf   ( Object       )
//      boolean   isEmpty   (              )
//
//      void      remove    ( int          )
//      void      remove    ( Object       )
//
//      void      set       ( int, Object  )
//      int       size      (              )
//
//-----

/**
 * Adds an item to end of the list.
 *
 * @param item an item to add.
 */
public void add( Object item )
{
    //creates a new ListNode
    ListNode newNode = new ListNode( item );

    if ( count == 0 ) {
        head = tail = newNode;
    }
    else {
        tail.next = newNode;
        tail = newNode;
    }

    count++;
}

/**
 * Adds an item to the list at index.
 */

```

```

    * @param index the position the item is added
    * @param item  an item to add.
    */
public void add( int index, Object item )
{
    if (index >=0 && index <= count ) {

        ListNode ptr = head;

        ListNode newNode = new ListNode( item );

        if ( index == 0 ) { //adding the first node

            newNode.next = head;
            head = newNode;
        }
        else {

            for (int i = 1; i < index; i++) {
                ptr = ptr.next;
            }

            newNode.next = ptr.next;
            ptr.next      = newNode;
        }

        //adjust tail if the new node added is
        //the last node in the list
        if ( index == count ) {
            tail = newNode;
        }

        count++;
    }
}

/**
 * Resets the list to be an empty list.
 */
public void clear( )
{
    head = tail = null;

    count = 0;
}

/**
 * Determines if the passed item in the list
 * or not. If it is, return true. Otherwise,
 * return false;
 */
public boolean contains( Object item )
{
    boolean result = true;

    int      loc      = indexOf( item );

```

```

        if ( loc == NOT_FOUND ) {
            result = false;
        }

        return result;
    }

    /**
     * Retrieves the object at the index position.
     * If index is invalid, null is returned.
     */
    public Object get( int index )
    {
        Object      item = null;

        ListNode    ptr  = head;

        if (index >= 0 && index < count) {
            for (int i = 0; i < index; i++) {
                ptr = ptr.next;
            }

            item = ptr.item;
        }

        return item;
    }

    /**
     * Finds the index position of the specified item.
     * If the item is not found, then NOT_FOUND is
     * returned.
     */
    public int indexOf( Object item )
    {
        int      loc      = 0;

        ListNode ptr      = head;

        while ( loc < count &&
                !ptr.item.equals( item ) ) {
            loc++;
            ptr = ptr.next;
        }

        if (loc == count) {
            loc = NOT_FOUND;
        }

        return loc;
    }

    /**
     * Determines whether the list is empty or not.
     * @return true if the list is empty; false otherwise.
     */
    public boolean isEmpty( )

```

```

{
    return ( count == 0 );
}

/**
 * Removes the element at the index position.
 *
 * @param index the position of the element to be removed
 */
public void remove( int index )
{
    if (index >=0 && index < count ) {

        ListNode ptr = head;

        for (int i = 1; i < index; i++) {
            ptr = ptr.next;
        }

        ptr.next = ptr.next.next;

        if (ptr.next == null) { //very last node was removed
            tail = ptr;      //we have a new last node
        }

        count--;
    }
}

/**
 * Removes the designated element
 *
 * @param item the item to remove from the list
 */
public void remove( Object item )
{
    ListNode ptr = head;
    ListNode trail = null;

    while ( ptr != null && !ptr.item.equals( item ) ) {

        trail = ptr;
        ptr = ptr.next;
    }

    if ( ptr != null ) {

        trail.next = ptr.next;
    }

    if ( trail.next == null ){ //very last node was removed
        tail = trail;      //we have a new last node
    }

    count--;
}

/**
 * Replaces the current element at the index position

```

```

    * with the passed element.
    *
    * @param index the position of the object to be replaced
    * @param item  the new item to replace the current element
    */
    public void set( int index, Object item )
    {
        if ( index >= 0 && index < count ) {

            ListNode ptr = head;

            for (int i = 0; i < index; i++) {
                ptr = ptr.next;
            }

            ptr.item = item;

        }
        //do nothing if index is out of valid range
    }

    /**
     * Returns the number of elements in the list
     *
     * @return the number of elements in the list
     */
    public int size( )
    {
        return count;
    }

    //-----
    //
    //      Inner Class: ListNode
    //
    //-----

    //Inner Class: ListNode
    class ListNode
    {
        private Object    item;

        private ListNode  next;

        public ListNode( Object item )
        {
            this.item = item;
            this.next = null;
        }
    }
}

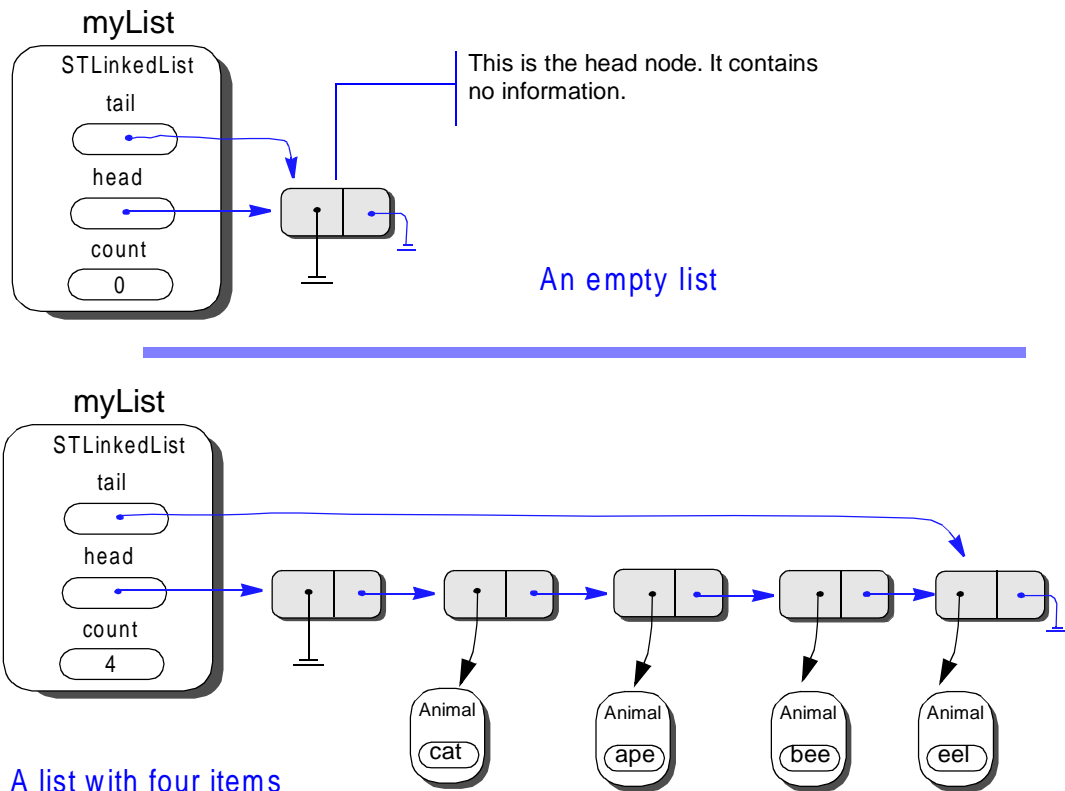
```

4 The Linked Implementation with the Head Node

In both versions of the add and remove methods of the linked implementation, two sets of code are written—one for the end case and another for the general case. To determine which set of code to execute, we include an if test. This testing is different from like testing whether a given integer is odd or even, where you have a 50-50 chance. If you add 1000 elements, for example, the end case occurs just once and all of the remaining 999 adds are the general case. It seems like a waste to test 1000 times to detect one case to respond differently. Is there a way to eliminate this testing?

The technique we can use to eliminate the testing for the end case is to insert one extra “dummy” node at the head of a list. By including the head node, one set of code can be used to handle both the end and general cases. Figure 9 shows the lists with the head node.

FIGURE 9 Sample lists with the head node.



With the head node in a list, the first version of add can be written as

```
public void add( Object item )
{
    //creates a new ListNode
    ListNode newNode = new ListNode( item );

    tail.next = newNode;
    tail = newNode;

    count++;
}
```

This code works for both the end and general cases so there's no need to test for which case.

The second version of add and the two versions of remove can also be written without the if test that determines the case. Please refer to the accompanying source file STLinkedList2.java for more details.

5 Dynamic versus Static Memory Allocation

dynamic memory
allocation

static memory
allocation

Often we hear the term *dynamic memory allocation* associated with the linked implementation. Dynamic allocation of memory means memory is allocated to a program while the program is being executed. In contrast to dynamic memory allocation, there is *static memory allocation* where the amount of memory to be allocated to the program is determined at the time the program is compiled. In many programming languages, memory for arrays can only be allocated statically. With Java, however, arrays are allocated dynamically (Note: One way to tell whether you are allocating dynamically or not is the use of reserved word `new`. If you use the `new` operator, then it is allocated dynamically.)

Static memory allocation is often not flexible enough to accommodate varying needs of a program. For example, one program may require memory space to store 100 data items in one execution and 1000 data items in another execution. What happens when the program manipulates 1000 data items only rarely, and for the majority of executions, the program uses 50 data items? If the programming language you are using allows only the static allocation of memory for arrays, then you have no choice but to declare the array of size 1000. This means you are wasting a large amount of memory for the majority of executions. Worst, what happens when over 1000 data items are entered into the program? It would crash. For such programming languages, dynamic allocation of linked lists makes much more sense if the program has to process a varying size of data items. Although linked lists require more memory to store the same amount of information than arrays because the links in the linked lists take up four bytes of memory each, flexibility of dynamic memory allocation outweighs

the slight disadvantage of larger memory usage for the applications whose memory needs vary greatly.

Since Java allows dynamic allocation for arrays also, the distinction between the array implementation and linked implementation may not be as stark in Java as in other languages. Still, finer granularity of memory allocation and the use of pointers in the linked implementation make the update operations perform better. We will discuss briefly the performances of the array and linked implementations in the next section.

6 The Performances of the Array and Linked Implementations

How do the performances of two different implementations stack up against each other? In general, the array implementation is better in locating an object by position while the linked implementation is better in adding and removing an object. With the array implementation, a node can be located easily with the indexed expression if the node's position is given. With the linked implementation, on the other hand, one must traverse the links for i times to locate the node at the i 'th position. In contrast, with the array implementation, nodes after the i 'th position must be shifted by one position to the left or right when a node at the i 'th position is removed or a new node is added at the i 'th position, respectively. With the linked implementation, on the other hand, one just updates the link field of one or two nodes. For the other operations, the performances of two implementations are roughly equivalent.