Time Complexity of Algorithms

You are expected to:

- use big-O notation to categorize an algorithm as constant, linear, quadratic, logarithmic and exponential time
- given two or more algorithms, rank them in terms of their time efficiency

Complexity of Algorithms

- In the coming lectures, we'll be discussing different implementations of collections and comparing them with respect to certain operations.
- We need to have a good way to define the performance of an algorithm (or a piece of code).
- In this section, we examine a means of analyzing the performance of an algorithm. Usually we are interested in the algorithm's
  - ***time complexity***: time taken for an algorithm to run
  - ***space complexity:*** amount of memory required by it
- In this course we mainly interested in time complexity .

Time Complexity

- One approach to determining an algorithm's *time complexity* would be to count the number of CPU cycles (or CPU time) it takes the algorithm to to perform its operation
  - tedious and not a very practical approach
  - depends on the machine
- Instead we will count the number of ***simple statements*** (or ***steps***) which are executed by the algorithm for a given input value n (time will be a function of n).
- By *simple statement* we mean a statement whose running time does not depend on n:
  - an assignment (without function calls)
  - a comparison between variables, etc.

  For instance, a loop that executes n times would contribute n times the number of steps of the body

Big-O Notation

- We are not interested in an exact count of steps. Instead we want to know how fast the time grows as n grows. So, we use the following approximation
- **Definition:** Let T and f be a functions of n. We say that T is $O(f(n))$ (pronounced "big-O f(*n*)" or "O f(*n*)") if:

  $T(n) \le c\, f(n)$ for any $n > n_0$

  where c and n_0 are constants.
- **Example:** Suppose that T is the time taken for an algorithm to sort an array of length n and that:

  $T(n) \le c\, n^2$

  for all n then we say that the algorithm is $O(n^2)$.

Example 1

```
int count = 0;
int sum = 0;
while( count < N )
{
      sum += count;
      count++;
}
System.out.print( " The sum is :  ");
System.out.println( sum );
```

- The time complexity of it depends on N.
- So
  T(N) = 2 + (3)N + 2
  = 4 + 3N
- and
  T(N) <= 4N + 3N
  <= 7N
  <= c N (c=7)
- Therefore
  T(N) is O(n)

Linear Algorithms

- Algorithms like the previous one are called "***linear algorithms***"
- This means that the time taken to execute the algorithm $T(n)$ *for large values of* n is $O(n)$
- It also means that the time for the algorithm grows linearly as n grows
- Let's suppose that we double *n*. How does this affect the time taken to execute the algorithm?

Example 2

```
int count = 0;
int sum = 0;
while( count < N )
{
      int index = 0;
      while( index < N )
      {
         sum += index * count;
         index++;
      }
      count++;
}
```

- $T(N) = 2 + N(2+ N(3) +1)$
  $= 2 +N(3N+3)$
  $= 3N^2 + 3N + 2$
- Then
  $T(N) <= 3N^2 + 3N^2 + 2N^2$
  or
  $T(N) <= 8N^2$
- Therefore:
  $T(N)$ is $O(N^2)$

Quadratic Algorithms and More

- Algorithms like the previous one are called "***quadratic algorithms***"
- This means that the time taken to execute the algorithm $T(n)$ *for large values of* n is $O(n^2)$
- It also means that the time for the algorithm grows by n^2 as n grows
- In general we are interested in the following algorithm types:

Algorithm Type	**T(n)**
Constant	$O(1)$
Linear	$O(n)$
Logarithmic	$O(\log n)$
Polynomial	$O(n^k)$ where k is an int constant
Exponential	$O(k^n)$ where k is an int constant

Complexity Example (Sorting)

Selection Sort

- Sorts an array by repeatedly finding the smallest element of the unsorted tail region and moving it to the front
- Slow when run on large data sets
- Example: sorting an array of integers

11	9	17	5	12

Sorting an Array of Integers

- Find the smallest and swap it with the first element

5	9	17	11	12

- Find the next smallest. It is already in the correct place

5	9	17	11	12

- Find the next smallest and swap it with first element of unsorted portion

5	9	11	17	12

- Repeat

5	9	11	12	17

- When the unsorted portion is of length 1, we are done

5	9	11	12	17

ch14/selsort/SelectionSorter.java

```
/**
   This class sorts an array, using the selection sort
   algorithm
*/
public class SelectionSorter
{
   /**
      Constructs a selection sorter.
      @param anArray the array to sort
   */
   public SelectionSorter(int[] anArray)
   {
      a = anArray;
   }

   /**
      Sorts the array managed by this selection sorter.
   */
   public void sort()
   {
```

Continued

ch14/selsort/SelectionSorter.java (cont.)

```
   for (int i = 0; i < a.length - 1; i++)
   {
      int minPos = minimumPosition(i);
      swap(minPos, i);
   }
}

/**
   Finds the smallest element in a tail range of the array.
   @param from the first position in a to compare
   @return the position of the smallest element in the
   range a[from] . . . a[a.length - 1]
*/
private int minimumPosition(int from)
{
   int minPos = from;
   for (int i = from + 1; i < a.length; i++)
      if (a[i] < a[minPos]) minPos = i;
   return minPos;
}
```

Continued

ch14/selsort/SelectionSorter.java (cont.)

```
   /**
      Swaps two entries of the array.
      @param i the first position to swap
      @param j the second position to swap
   */
   private void swap(int i, int j)
   {
      int temp = a[i];
      a[i] = a[j];
      a[j] = temp;
   }

   private int[] a;
}
```

ch14/selsort/SelectionSortDemo.java

```java
import java.util.Arrays;

/**
   This program demonstrates the selection sort algorithm by
   sorting an array that is filled with random numbers.
*/
public class SelectionSortDemo
{
   public static void main(String[] args)
   {
      int[] a = ArrayUtil.randomIntArray(20, 100);
      System.out.println(Arrays.toString(a));

      SelectionSorter sorter = new SelectionSorter(a);
      sorter.sort();

      System.out.println(Arrays.toString(a));
   }
}
```


File ArrayUtil.java

Typical Output:

```
[65, 46, 14, 52, 38, 2, 96, 39, 14, 33, 13, 4, 24, 99, 89, 77,
73, 87, 36, 81] [2, 4, 13, 14, 14, 24, 33, 36, 38, 39, 46, 52,
65, 73, 77, 81, 87, 89, 96, 99]
```

Question

Why do we need the `temp` variable in the `swap` method? What would happen if you simply assigned `a[i]` to `a[j]` and `a[j]` to `a[i]`?

Answer: Dropping the `temp` variable would not work. Then `a[i]` and `a[j]` would end up being the same value.

Question

What steps does the selection sort algorithm go through to sort the sequence 6 5 4 3 2 1?

Answer:

1	5	4	3	2	6

1	2	4	3	5	6

1	2	3	4	5	6

Analyzing the Performance of the Selection Sort Algorithm

- In an array of size *n*, count how many times an array element is visited
  - *To find the smallest, visit n elements + 2 visits for the swap*
  - *To find the next smallest, visit (n - 1) elements + 2 visits for the swap*
  - *The last term is 2 elements visited to find the smallest + 2 visits for the swap*

Analyzing the Performance of the Selection Sort Algorithm

- The number of visits:
  - $n + 2 + (n - 1) + 2 + (n - 2) + 2 + . . .+ 2 + 2$
  - *This can be simplified to* $n^2/2 + 5n/2 - 3$
  - $5n/2 - 3$ *is small compared to* $n^2/2$ *– so let's ignore it*
  - *Also ignore the 1/2 – it cancels out when comparing ratios*

Analyzing the Performance of the Selection Sort Algorithm

- The number of visits is of the order n^2
- Using big-Oh notation: The number of visits is $O(n^2)$
- Multiplying the number of elements in an array by **2** multiplies the processing time by **4**
- Big-Oh notation "$f(n) = O(g(n))$"
  expresses that f grows no faster than g
- To convert to big-Oh notation:
  locate fastest-growing term, and ignore constant coefficient

Question

If you increase the size of a data set tenfold, how much longer does it take to sort it with the selection sort algorithm?

Answer: It takes about 100 times longer.

Learning Goals Review

You are expected to:

- use big-O notation to categorize an algorithm as constant, linear, quadratic, logarithmic and exponential time
- given two or more algorithms, rank them in terms of their time efficiency

Java Collections: the `List` Interface

You will be expected to:

- program to the generic List interface including read and use the List API (e.g., use Lists in ways similar to arrays)
- program using the ListIterator interface (be able to read and use the ListIterator API)
- explain the difference between `Iterator` and `ListIterator`
- compare and contrast ArrayList and LinkedList implementations of the List interface

Reading :

- 2nd Ed: 20.1
- 3rd/4th Eds: 15.1

The List Interface

- A ***list*** is an ordered collection that can contain duplicates. Lists
  are also called ***sequences.*** ***Example:*** ArrayLists.
- The List interface extends Collection and adds methods for:
  - **positional access:** can access
    - current position (using an iterator)
    - i-th element $(0 \leq i < size)$
  - **positional search**
    - returns the position of a given object
  - **special iteration**
    - defines special iterators for moving forwards *or backwards*
  - **subrange operations**
    - create sub-lists
    - add/delete elements at a given position

The List Interface (cont'd)

```
public interface List<E> extends Collection<E> {
  // Positional Access
  E get(int index);
  E set(int index, E element);  // Optional
  void add(int index, E element);  // Optional
  E remove(int index);          // Optional
  boolean addAll(int index, Collection c); // Optional
  // Search
  int indexOf(Object o);
  int lastIndexOf(Object o);
  // Iteration
  ListIterator<E> listIterator();
  ListIterator<E> listIterator(int index);
  // Sublist
  List<E> subList(int fromIndex, int toIndex);
```

The `List` Interface (cont'd)

- New methods in addition to those inherited from `Collection`:
  - `add(i,e)` adds at position `i`
  - `addAll(i,c)` adds the given collection `c` starting at position `i`
  - `remove(i)` removes object at position `i`

- Methods whose behaviour is specified to be different than in the `Collection` interface (overridden):
  - `add(e)` – adds `e` at the end of the list
  - `addAll(c)` – adds collection `c` at the end of the list
  - `remove(o)` – removes first occurrence of o

The `List` Interface (cont'd)

- The `subList` method returns a *view* of this list between `fromIndex` (inclusive) and `toIndex` (exclusive).
- Any non-structural changes to the sublist are reflected in this list and vice versa.
- You must not make structural changes (i.e., add or remove) to the original underlying list while using the sublist.
- Structural changes to the sublist *are* reflected in the backing list.

The `List` Interface (cont'd)

- **Example**: clear / remove all the items from a list between index 1 (inclusive) and 4 (exclusive):

```
myList.subList( 1, 4 ).clear();
```

- **Example**: a function that swaps two list elements:
  - `public static <T> void swap(List<T> list, int i, int j)`
  - `{`
  - `}`

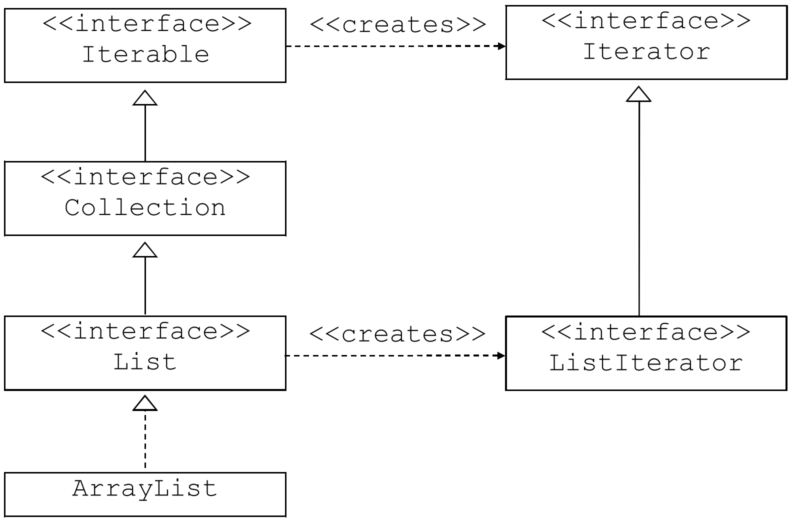
- Why do we have `<T>` at the start of the method declaration?

`ListIterator`

- In addition to a general `Iterator`, a list can also create a more specialized `ListIterator`.

- A `ListIterator` is an example of a *bi-directional* iterator. You can traverse the list either forwards or backwards.

Java Collection Framework



The `ListIterator` Interface

```
public interface ListIterator<E>
                     extends Iterator<E>
{
  boolean hasPrevious();
  E previous();

  int nextIndex();
  int previousIndex();

  void set(E o);      // Optional
  void add(E o);      // Optional
}
```

ListIterator

- Can create an iterator that is positioned at
  - the first element
  - a specified position
- `next()` returns items <u>in the order they are in the list</u>
- List iterators can move forwards or backwards
  - `nextIndex()` returns the index of the next item
  - `previousIndex()` returns the index of the previous element
- Best way is to think that the iterator points *between* the items:

	↑	item 0	↑	item 1	↑	item 2	↑	item 3	↑
index:	0		1		2		3		4

ListIterator (cont')

- The `remove` and `set` method affect the last element that was returned by a call to `next` or `previous`.
  - Cannot be called if remove() or add() have been called since last call to next() or previous(). Throws IllegalStateException.
- The `add` method adds a new element after the one that will be returned by a call to `previous` and before the one that will be returned by `next`.
  - Cannot be called if add() has been called since last call to next() or previous(). Throws IllegalStateException.
- The restrictions we discuss for general iterators are also applicable to ListIterator's. See the Java API for more details on restrictions.

ListIterator (cont')

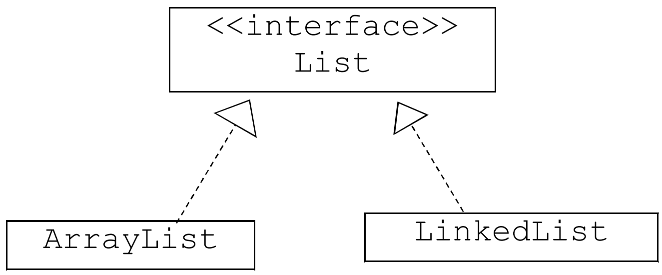
- After performing an `add`, a subsequent call to `previous` will return the element just added and a subsequent call to `next` is unaffected.
- Example:

In-Class Exercise I

- 1. Write a method that takes a List<String> parameter and prints out each item in the list
- 2. Write a second method that takes a List<String> parameter and prints out each item in reverse
- Indicate the time complexity of your methods

List Implementations

- The Java collections framework provides the following implementations of a `List` interface (among others):
  - `ArrayList`
  - `LinkedList`



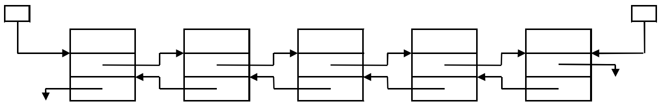
List Implementations

- We've already considered the `ArrayList`. As you can probably guess, if we were to look under the hood, we'd find that its implementation is based on an array.

- The `LinkedList` class provides a different implementation of the `List` interface.

The `LinkedList` Class

- Rather than using an array to store data, the `LinkedList` class stores data in a collection of nodes that are chained or linked together similar to the one shown below:



- One advantage of this implementation is that the list creates only as many nodes as are needed to store the data in the list. The number of nodes increases/decreases every time we add/remove data.

Using Linked Lists

- A linked list consists of a number of nodes, each of which has a reference to the next node
- Adding and removing elements in the middle of a linked list is efficient
- Visiting the elements of a linked list in sequential order is efficient
- Random access is not efficient

Inserting an Element into a Linked List

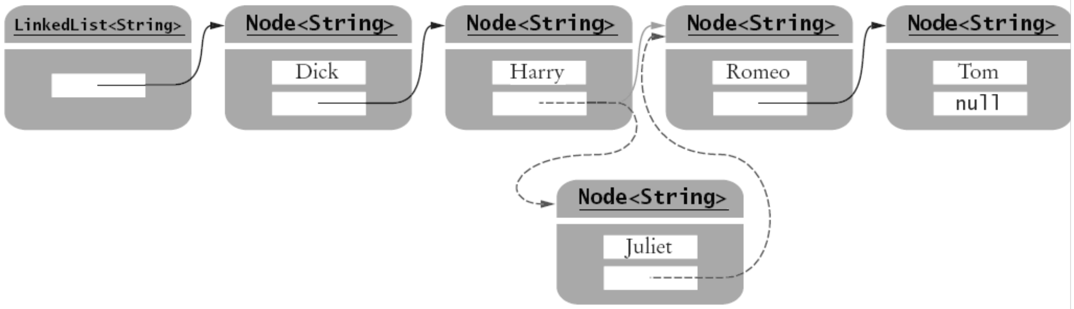


Figure 1 Inserting an Element into a Linked List

Java's `LinkedList` class

- Generic class
  - *Specify type of elements in angle brackets:* `LinkedList<Product>`
- Package: `java.util`
- Easy access to first and last elements with methods
  void addFirst(E obj)
  void addLast(E obj)
  E getFirst()
  E getLast()
  E removeFirst()
  E removeLast()

List Iterator

- `ListIterator` type
- Gives access to elements inside a linked list
- Encapsulates a position anywhere inside the linked list
- Protects the linked list while giving access

A List Iterator

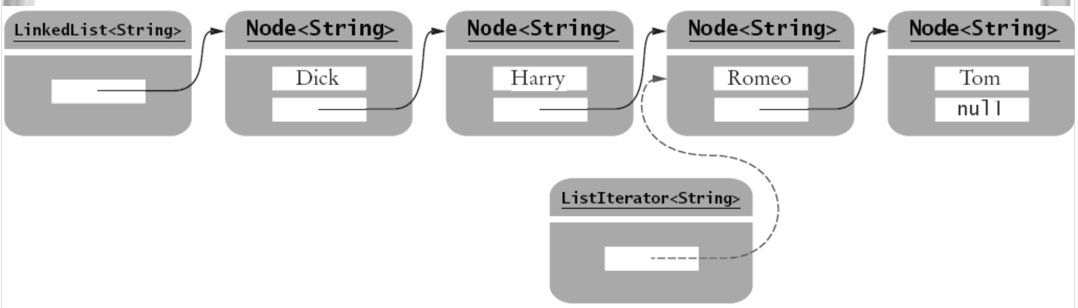


Figure 2 A List Iterator

A Conceptual View of the List Iterator

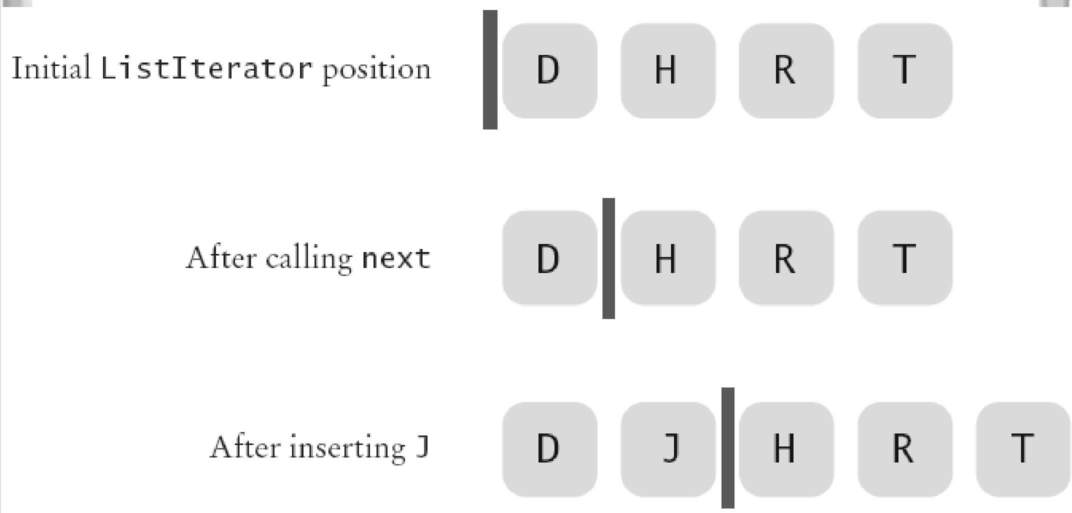


Figure 3 A Conceptual View of the List Iterator

List Iterator

- Think of an iterator as pointing between two elements
  - *Analogy: like the cursor in a word processor points between two characters*
- The `listIterator` method of the `LinkedList` class gets a list iterator

```
LinkedList<String> employeeNames = . . .; ListIterator<String>
iterator =
      employeeNames.listIterator();
```

List Iterator

- Initially, the iterator points before the first element
- The `next` method moves the iterator

```
iterator.next();
```

- `next` throws a `NoSuchElementException` if you are already past the end of the list
- `hasNext` returns true if there is a next element

```
if (iterator.hasNext())
iterator.next();
```


List Iterator

- The `next` method returns the element that the iterator is passing

```
while iterator.hasNext()
{
   String name = iterator.next();
   Do something with name
}
```

- Shorthand:

```
for (String name : employeeNames)
{
   Do something with name
}
```

Behind the scenes, the for loop uses an iterator to visit all list elements

List Iterator

- `LinkedList` is a *doubly linked list*
  - *Class stores two links:*
    - One to the next element, and
    - One to the previous element
- To move the list position backwards, use:
  - *`hasPrevious`*
  - *`previous`*

Adding and Removing from a LinkedList

- The `add` method:
  - *Adds an object after the iterator*
  - *Moves the iterator position past the new element*

```
iterator.add("Juliet");
```

Adding and Removing from a LinkedList

- The `remove` method
  - *Removes and*
  - *Returns the object that was returned by the last call to `next` or `previous`*

```
 //Remove all names that fulfill a certain condition
while (iterator.hasNext())
{
String name = iterator.next();
if (name fulfills condition)
   iterator.remove(); }
```

- Be careful when calling `remove`:
  - *It can be called only once after calling `next` or `previous`*
  - *You cannot call it immediately after a call to `add`*
  - *If you call it improperly, it throws an `IllegalStateException`*

Sample Program

- `ListTester` is a sample program that
  - *Inserts strings into a list*
  - *Iterates through the list, adding and removing elements*
  - *Prints the list*

ch15/uselist/ListTester.java

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

/**
   A program that tests the LinkedList class
*/
public class ListTester
{
   public static void main(String[] args)
   {
      LinkedList<String> staff = new LinkedList<String>();
      staff.addLast("Dick");
      staff.addLast("Harry");
      staff.addLast("Romeo");
      staff.addLast("Tom");

      // | in the comments indicates the iterator position

      ListIterator<String> iterator
            = staff.listIterator(); // |DHRT
      iterator.next(); // D|HRT
      iterator.next(); // DH|RT
```

Continued

ch15/uselist/ListTester.java (cont.)

```java

      // Add more elements after second element

      iterator.add("Juliet"); // DHJ|RT
      iterator.add("Nina"); // DHJN|RT

      iterator.next(); // DHJNR|T

      // Remove last traversed element

      iterator.remove(); // DHJN|T

      // Print all elements

      for (String name : staff)
         System.out.print(iterator.next() + " ");
      System.out.println();
      System.out.println("Expected: Dick Harry Juliet Nina Tom");
   }
}
```

ch15/uselist/ListTester.java (cont.)

Output:

```
Dick Harry Juliet Nina Tom
Expected: Dick Harry Juliet Nina Tom
```


LinkedList

- Let's look at how LinkedList is actually implemented, since it differs from ArrayList
- This is actually a simplified version of LinkedList from Big Java

Adding a New First Element

- When a new node is added to the list
  - *It becomes the head of the list*
  - *The old list head becomes its next node*

Adding a New First Element

```
public void addFirst(Object obj)
{
   Node newNode = new Node(); ❶
   newNode.data = obj;
   newNode.next = first; first = newNode;
}
```

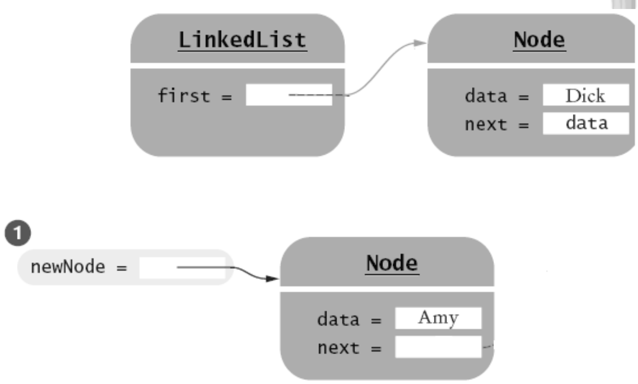


Figure 4 Adding a Node to the Head of a Linked List

Adding a New First Element

```
public void addFirst(Object obj)
{
   Node newNode = new Node(); 1
   newNode.data = obj;
   newNode.next = first; 2
   first = newNode;
}
```

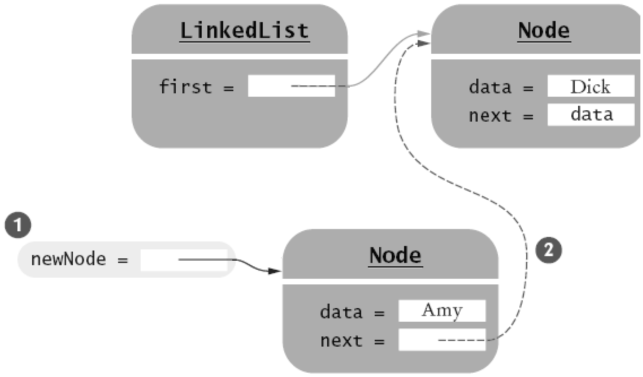


Figure 4 Adding a Node to the Head of a Linked List

Adding a New First Element

```
public void addFirst(Object obj)
{
   Node newNode = new Node(); 1
   newNode.data = obj;
   newNode.next = first; 2
   first = newNode; 3
}
```

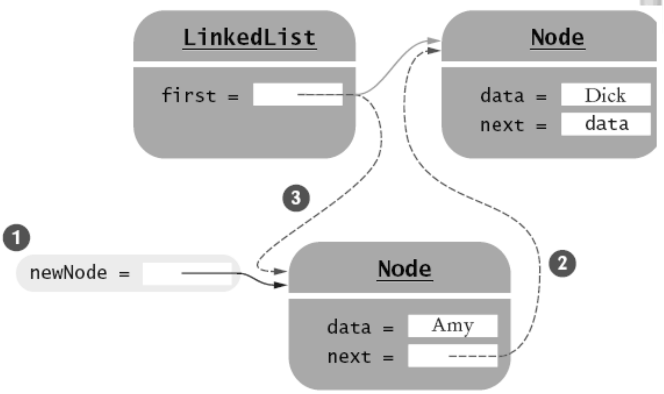


Figure 4 Adding a Node to the Head of a Linked List

Removing the First Element

- When the first element is removed
  - *The data of the first node are saved and later returned as the method result*
  - *The successor of the first node becomes the first node of the shorter list*
  - *The old node will be garbage collected when there are no further references to it*

Removing the First Element

```
public Object removeFirst()
{
   if (first == null)
      throw new NoSuchElementException();
   Object obj = first.data;
   first = first.next;
   return obj;
}
```

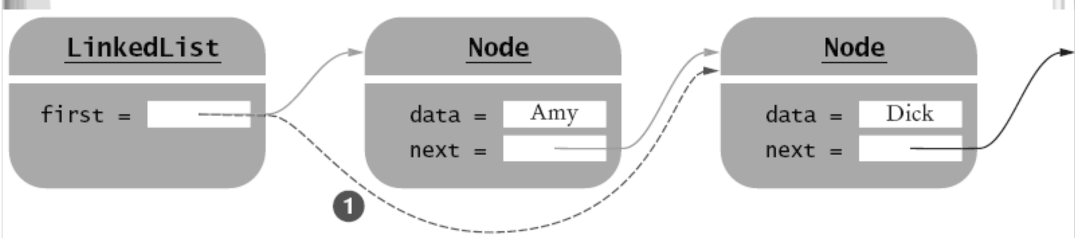


Figure 5 Removing the First Node from a Linked List

Linked List Iterator

- We define `LinkedListIterator`: private inner class of `LinkedList`
- Implements a simplified `ListIterator` interface
- Has access to the `first` field and private `Node` class
- Clients of `LinkedList` don't actually know the name of the iterator class
  - *They only know it is a class that implements the `ListIterator` interface*

LinkedListIterator

The `LinkListIterator` class

```
public class LinkedList
{
   . . .
   public ListIterator listIterator()
   {
      return new LinkedListIterator();
   }

   private class LinkedListIterator implements
         ListIterator
   {
      public LinkedListIterator()
      {
         position = null;
         previous = null;
      }
```

Continued

LinkedListIterator (cont.)

```
      . . .
      private Node position;
      private Node previous;
   }
    . . .
}
```


The Linked List Iterator's `next` Method

- `position`: reference to the last visited node
- Also, store a reference to the last reference before that
- `next` method: `position` reference is advanced to `position.next`
- Old position is remembered in `previous`
- If the iterator points before the first element of the list, then the old `position` is `null` and `position` must be set to `first`

The Linked List Iterator's `next` Method

```
public Object next()
{
   if (!hasNext())
      throw new NoSuchElementException();
   previous = position; // Remember for remove
   if (position == null)
      position = first;
   else position = position.next;
   return position.data;
}
```


The Linked List Iterator's `hasNext` Method

- The `next` method should only be called when the iterator is not at the end of the list
- The iterator is at the end
  - *if the list is empty (`first == null`)*
  - *if there is no element after the current position (`position.next == null`)*

The Linked List Iterator's `hasNext` Method

```
private class LinkedListIterator implements ListIterator
{
   . . .
   public boolean hasNext()
   {
      if (position == null)
         return first != null;
      else
         return position.next != null;
   }
   . . .
}
```


The Linked List Iterator's `remove` Method

- If the element to be removed is the first element, call `removeFirst`
- Otherwise, the node preceding the element to be removed needs to have its `next` reference updated to skip the removed element
- If the `previous` reference equals `position`:
  - *this call does not immediately follow a call to next*
  - *throw an `IllegalArgumentException`*
- It is illegal to call `remove` twice in a row
  - *remove sets the `previous` reference to `position`*

The Linked List Iterator's `remove` Method

```
public void remove()
{
   if (previous == position)
      throw new IllegalStateException();
   if (position == first)
   {
      removeFirst();
   }
   else
   {
      previous.next = position.next; ❶
   }
   position = previous;
}
```

Continued

The Linked List Iterator's `remove` Method (cont.)

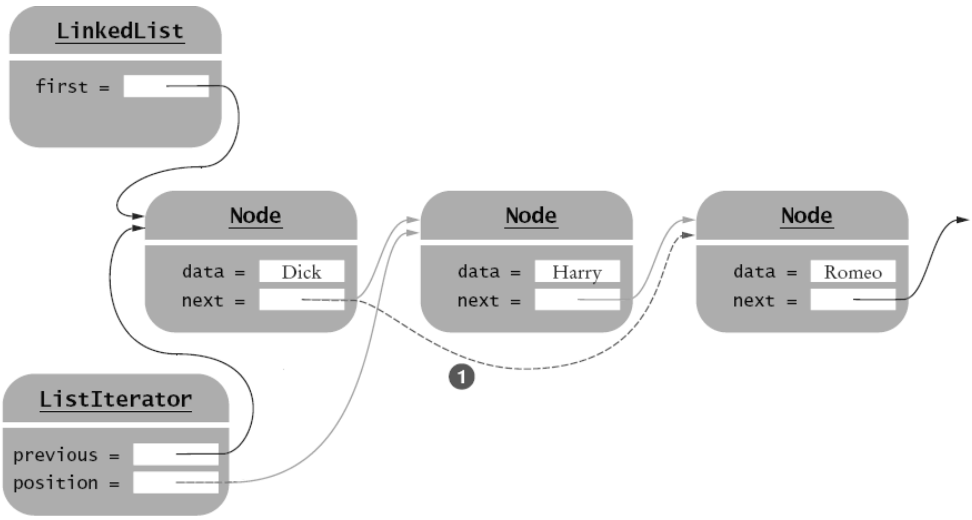


Figure 6 Removing a Node from the Middle of a Linked List

The Linked List Iterator's `remove` Method

```
public void remove()
{
   If (previous == position)
      throw new IllegalStateException();
   if (position == first)
   {
      removeFirst();
   }
   else
   {
      previous.next = position.next; 1
   }
   position = previous; 2
}
```

Continued

The Linked List Iterator's `remove` Method (cont.)

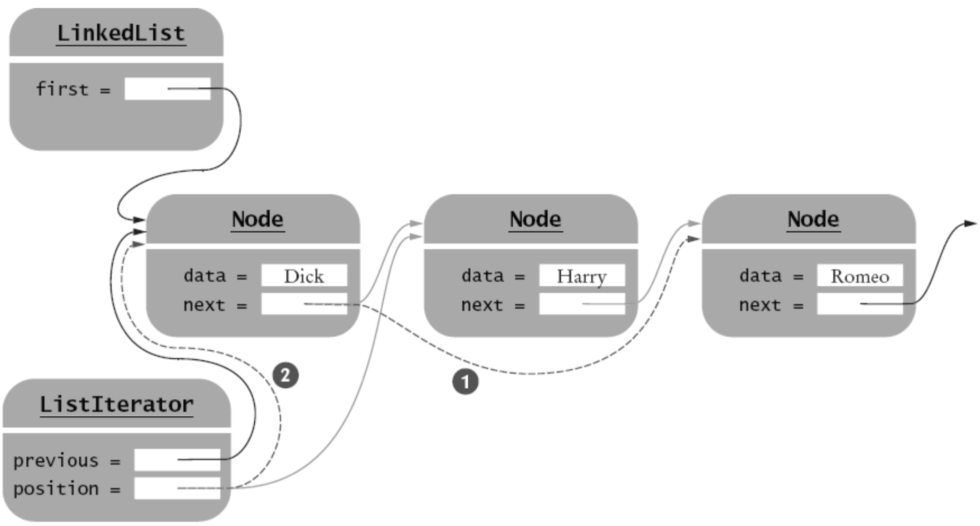


Figure 6 Removing a Node from the Middle of a Linked List

The Linked List Iterator's `set` Method

- Changes the data stored in the previously visited element
- The `set` method

```
public void set(Object obj)
{
   if (position == null)
      throw new NoSuchElementException();
    position.data = obj;
}
```


The Linked List Iterator's `add` Method

- The most complex operation is the addition of a node
  - *add inserts the new node after the current position*
  - *Sets the successor of the new node to the successor of the current position*

The Linked List Iterator's `add` Method

```
public void add(Object obj)
{
   if (position == null)
   {
      addFirst(obj);
      position = first;
   }
   else
   {
      Node newNode = new Node();
      newNode.data = obj;
      newNode.next = position.next; ❶
      position.next = newNode;
      position = newNode;
   }
   previous = position;
}
```

Continued

The Linked List Iterator's `add` Method (cont.)

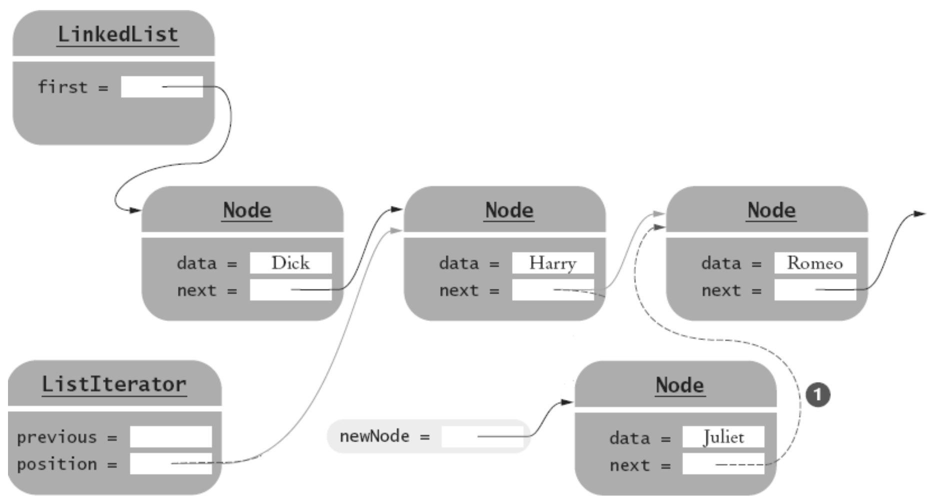


Figure 7 Adding a Node to the Middle of a Linked List

The Linked List Iterator's `add` Method

```
public void add(Object obj)
{
   if (position == null)
   {
      addFirst(obj);
      position = first;
   }
   else
   {
      Node newNode = new Node();
      newNode.data = obj;
      newNode.next = position.next; ❶
      position.next = newNode;     ❷
      position = newNode;
   }
   previous = position;
}
```

Continued

The Linked List Iterator's `add` Method (cont.)

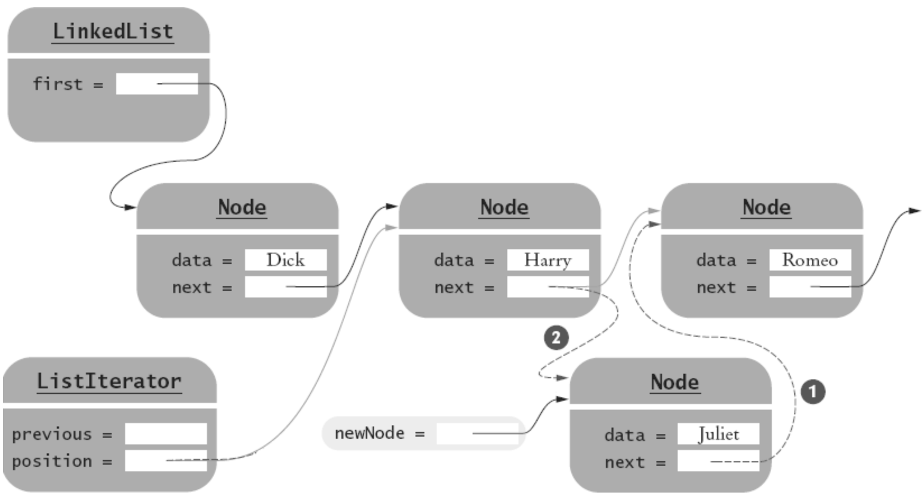


Figure 7 Adding a Node to the Middle of a Linked List

The Linked List Iterator's `add` Method

```
public void add(Object obj)
{
   if (position == null)
   {
      addFirst(obj);
      position = first;
   }
   else
   {
      Node newNode = new Node();
      newNode.data = obj;
      newNode.next = position.next;
      position.next = newNode;
      position = newNode;
   }
   previous = position;
}
```

Continued

The Linked List Iterator's `add` Method (cont.)

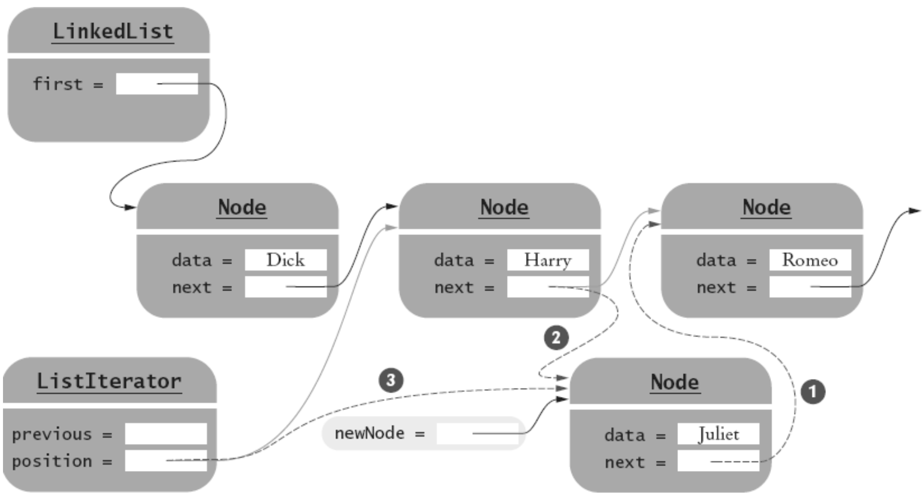


Figure 7 Adding a Node to the Middle of a Linked List

The `LinkedList` Class

- Let's look at some differences between `ArrayList` and `LinkedList`. Let's consider the time complexity of some common operations in the *worst case*:

	LinkedList	ArrayList
get(int index)	O(n)	O(1)
add(int i, E e)	O(n)	O(n)
add(E e)	O(1)	O(1)
remove(int index)	O(n)	O(n)
contains(Object o)	O(n)	O(n)
ListIterator -> add	O(1)	O(n)
ListIterator -> remove	O(1)	O(n)

Efficiency of Operations for Arrays and Lists

Operation	ArrayList	LinkedList
Random access	$O(1)$	$O(n)$
Linear traversal step	$O(1)$	$O(1)$
Add/remove an element	$O(n)$	$O(1)$

List Example

- Reverse a list

```
public static <E> List<E> reverse(List<E> list) {

   List<E> newlist = new ArrayList<E>();

   Iterator<E> itr = list.iterator();
     while (itr.hasNext()) {
       newlist.add(0, itr.next());
   }
   return newlist;
}
```

What is its complexity?

List Example

- A better way to write this method:

```
public static <E> List<E> reverse(List<E> list) {
```

In-Class Exercise II

- Write a method that takes an ArrayList<String>, and visits each item in order, printing out the item and then removing it
- Write a method that takes a LinkedList<String>, and visits each item in reverse order, printing out the item and then removing it
- Indicate the complexity of each method

Learning Goals Review

You will be expected to:
- program to the generic List interface including read and use the List API (e.g., use Lists in ways similar to arrays)
- program using the ListIterator interface (be able to read and use the ListIterator API)
- explain the difference between `Iterator` and `ListIterator`
- compare and contrast ArrayList and LinkedList implementations of the List interface

Midterm Exam

- Class contracts
  - Preconditions
  - Postconditions
  - Invariants

Midterm Exam

- Exceptions
  - Throwing
  - Catching
  - Propagating
  - Defining

Midterm Exam

- Testing
  - Unit testing
  - Blackbox testing
  - Equivalence classes
  - Test cases: typical values, boundary values

Midterm Exam

- Good and bad design
  - High cohesion, low coupling
  - Open-closed principle
  - Liskov substitution principle
  - Weakening precondition, strengthening postcondition

Midterm Exam

- Java collections
  - Interfaces: Iterable, Collection, List, Iterator
  - Classes: ArrayList
- Generic programming
  - Generic classes (defining and using)
  - Generic methods (defining and using)
  - Type parameters
  - Bounded wildcards