

Introduction to Data Structures

Introduction - Suppose we have a phone book with 20,000,000 people and a new person is added and we want to add in alphabetically. A Linked List will allow us to do that *efficiently*.

I. Linked Lists

A) A linked list is a sequence of elements (nodes). Each element stores an object and a reference to the next element (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 like in arrays is not easy
- The class needed is called **LinkedList** and **import java.util.LinkedList** is needed

(Note: LinkedList automatically implements the List interface which an ArrayList also does so all the methods that an ArrayList has so does a LinkedList - add, get, set, size - but a LinkedList has others, as we will see)

B) Easy access to first and last elements with methods

```
void addFirst(Object obj)
```

```
void addLast(Object obj)
```

```
Object getFirst( )
```

```
Object removeFirst( )
```

```
ex) LinkedList ex1 = new LinkedList( );
```

```
ex1.getFirst( )
```

```
ex1.getLast( )
```

```
ex1.removeFirst( )
```

```
ex1.removeLast( )
```

C) Inserting an Element into a Linked List

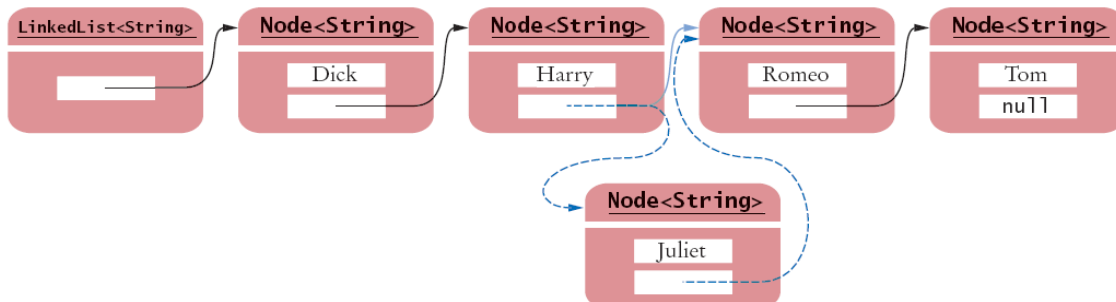


Figure 1 Inserting an Element into a Linked List

D) Using the listIterator method

- 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
ex.) `LinkedList letters = . . . ;`
`ListIterator iterator = letters.listIterator();`

A Conceptual View of the List Iterator

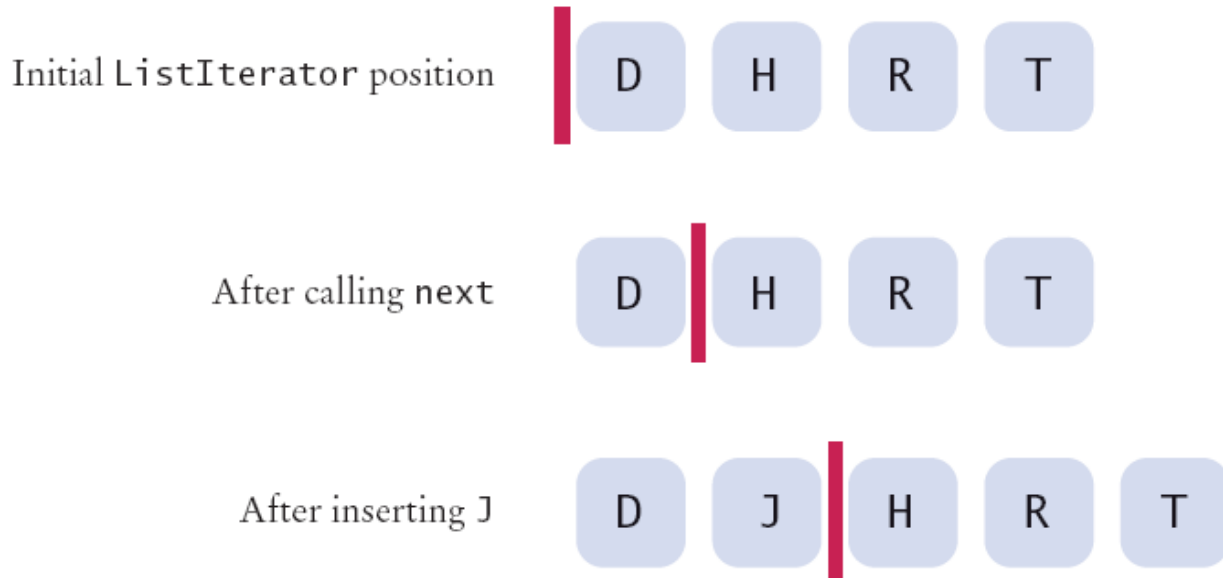


Figure 3 A Conceptual View of the List Iterator

*The iterator position is between two elements. It is like a link that connects the elements. (Technically a linked list is a doubly linked list because each element has a link to the previous element and one to the next element. To move the list position use: `hasPrevious()` and `previous()`.)

- Initially, the iterator points before the first element
- The next method moves the iterator:
`iterator.next();`
Note: `next()` throws a `NoSuchElementException` if you are already past the end of the list
- `hasNext()` returns true if there is a next element and should be used before using `next()`
`if (iterator.hasNext())`
`iterator.next();`

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  
}
```

E) Adding and Removing from a LinkedList

1) The add method:

- Adds an object after the iterator
- Moves the iterator position past the new element
ex.) iterator.add("Juliet");

2) The remove method

- Removes and returns the object that was returned by the last call to next or previous

```
ex.) // 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

See ex.) next page

AP Programming - Chapter 19 Lecture

```
import java.util.LinkedList;
import java.util.ListIterator;
// A program that demonstrates the LinkedList class
public class ListTester
{
    public static void main(String[] args)
    {
        LinkedList<String> staff = new LinkedList<String>( ); // new 5.0 syntax
        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

        // 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) // new 5.0 syntax
            System.out.print(name + " ");
    }
}
```

Output: Dick Harry Juliet Nina Tom

- F) Implementation of a simplified version of the LinkedList class
(You will see how the list operations manipulate the links as the list is modified.)
To keep it simple, we will implement a singly linked list and the example will supply direct access only to the first list element, not the last one which would be very similar. Also our example will not use a type parameter but will store raw Object values and insert casts when retrieving them.

Recall: a LinkedList class

- 1) Holds a reference variable called **first** that points to the first node - it is an instance field for the LinkedList class that holds the first link (pointer) to the first element (node)
- 2) Has a method to get the first element

Node will be the object type in our example that stores an object and a reference to the next node:

```
public class LinkedList
{
    . . .
    private class Node(Object x, Node y)
    {
        data = x
        next = y;
    }
    public Object data;
    public Node next;
}
```

Methods of linked list class and iterator class have frequent access to the Node instance variables

To make it easier to use:

- We make Node a private inner class of LinkedList
- We do not make the instance variables private
It is safe to leave the instance variables public since it is an inner class
- None of the list methods returns a Node object

1) getFirst() method:

```
public class LinkedList
{
    public LinkedList( )
    {
        first = null;
    }

    public Object getFirst( )
    {
        if (first == null)
            throw new NoSuchElementException( );
        return first.data; // or return first.getValue( ) could have been used
    }
    ...
    private Node first;
}
```

- 2) Adding a new first node i.e. adding a new element to the head of the linked list
When a new node is added to the list
- a) It becomes the head of the list
 - b) The old list head becomes its next node

```
public class LinkedList
{
    ...
    public void addFirst(Object obj)
    {
        Node newNode = new Node( ); ①
        // or Node newNode = new Node(obj, first); then next 2 lines not needed
        newNode.data = obj;
        newNode.next = first;        ②
        first = newNode;            ③
    }
    ...
}
```

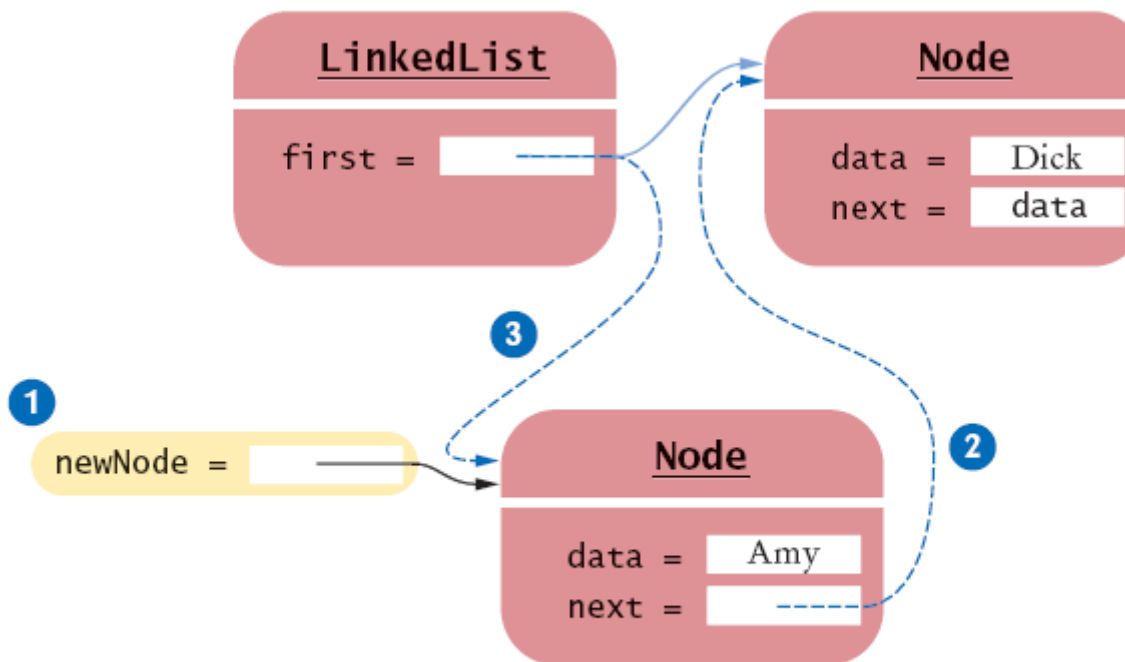


Figure 4 Adding a Node to the Head of a Linked List

2) Removing the new first node

When the first element is removed

- a) The data of the first node are saved and later returned as the method result
- b) The successor of the first node becomes the first node of the shorter list
- c) The old node will be garbage collected when there are no further references to it

```
public class LinkedList
{
    ...
    public Object removeFirst( )
    {
        if (first == null)
            throw new NoSuchElementException( );
        Object obj = first.data; // or Object obj = first.getValue( );
        first = first.next;      ❶ // or first = first.getNext( );
        return obj;
    }
    ...
}
```

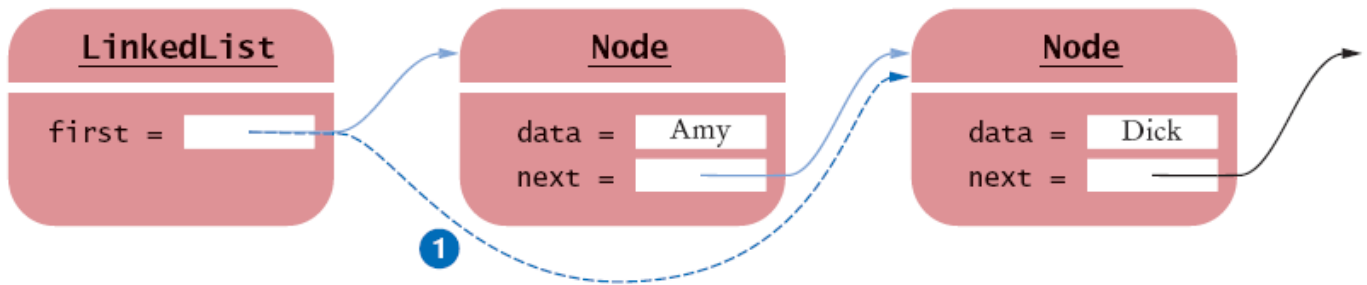


Figure 5 Removing the First Node from a Linked List

3) Adding a node somewhere in the middle of the linked list

The most complex operation is the addition of a node. The add method does:

- a) inserts the new node after the current position
- b) sets the successor of the new node to the successor of the current position

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

        newNode.next = position.next; 1
        position.next = newNode; 2
        position = newNode; 3
    }
    previous = position; 4
}
```

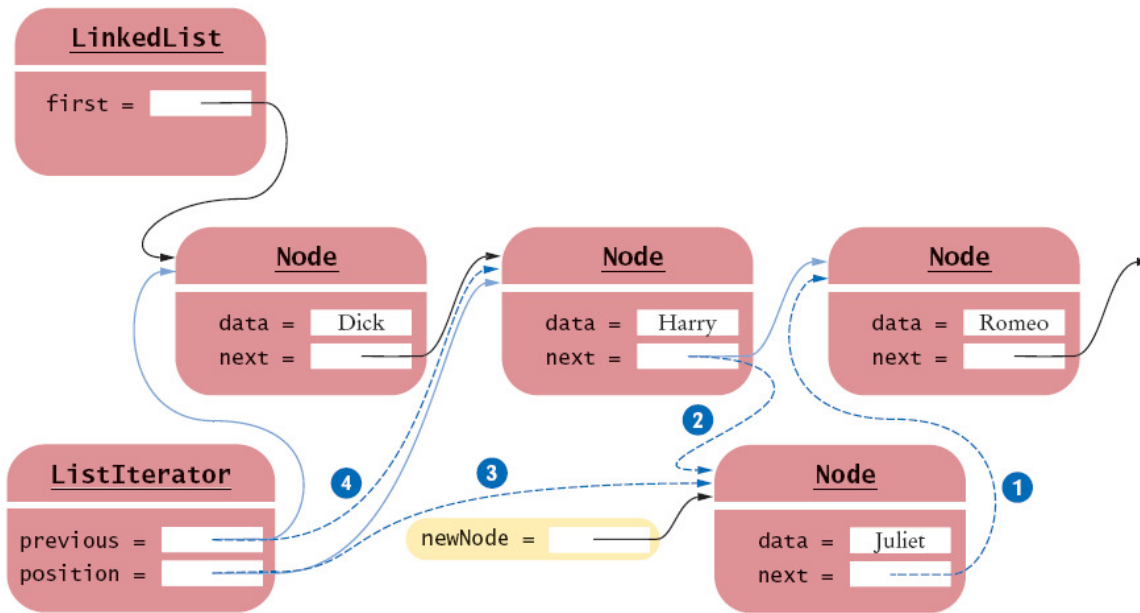


Figure 7 Adding a Node to the Middle of a Linked List

AP Programming - Chapter 19 Lecture

- 3) Adding a node somewhere in the middle of the linked list without using a built in iterator

```
Node first = null; // list is initialized to be empty
Node newNode = new Node(name, null);
    // Our new Node is created to hold the new string value (name).
if (first == null) // check to see if we are inserting into an empty list.
{
    first = newNode;
}
// if the list is not empty, check to see if the new node should be
// inserted in the front of the list.
String firstValue = (String) first.getValue( );
String newValue = (String) newNode.getValue( );
if (firstValue.compareTo(newValue) >= 0) // new node goes first
{
    newNode.setNext(first);
    first = newNode;
}

// see if node belongs in the middle or at the end of the list
String newValue = (String) newNode.getValue( );
ListNode temp = first;
ListNode follower = temp;

// checking for end and for order
while (temp.getNext( ) != null &&
        newValue.compareTo((String) temp.getValue( )) > 0)
{
    follower = temp;
    temp = temp.getNext( );
}

// does it belong in the middle?
if (newValue.compareTo((String) temp.getValue( )) < 0)
{
    follower.setNext(newNode);
    newNode.setNext(temp);
}

else // or does it belong at the end?
{
    temp.setNext(newNode);
}
```

II. Stacks - an abstract data type i.e. a collection of items with "last in first out" retrieval

Allows insertion and removal of elements only at one end traditionally called the *top* of the stack.

- New items are added to the top of the stack called ***push***
- Items are removed at the top of the stack called ***pop***

This is called *last in, first out* or LIFO order

ex.1) Think of a stack of books



Figure 12 A Stack of Books

ex.1) stack code:

```
// Uses an array to implement a stack
Stack<String> s = new Stack<String>( ); // 5.0 syntax
s.push("A");
s.push("B");
s.push("C");
while (s.size( ) > 0)
    System.out.print(s.pop( ) + " ");
```

Output: C B A

Note: peekTop() is a method in Stack that would return the first (top) element

III. Queue: collection of items with "first in first out" retrieval

- Add items to one end of the queue (the tail)
- Remove items from the other end of the queue (the head)

Queues store items in a *first in, first out* or FIFO fashion i.e. items are removed in the same order in which they have been added

ex.) Think of people lining up - People join the tail of the queue and wait until they have reached the head of the queue



Figure 13 A Queue

ex.1) Queue code:

```
public class LinkedListQueue
{
    // Constructs an empty queue that uses a linked list.
    public LinkedListQueue( )
    {
        list = new LinkedList( );
    }

    // Adds an item to the tail of the queue param x the item to add
    public void add(Object x)
    {
        list.addLast(x);
    }

    // Removes an item from the head of the queue & return the removed item
    public Object remove( )
    {
        return list.removeFirst( );
    }

    // Gets the number of items in the queue & return the size
    int size( )
    {
        return list.size( );
    }

    private LinkedList list;
}
```

Note: peekFront() is a method in Queue that would return the front element

Note: enqueue() is a method in Queue that would insert an element at the end

Note: dequeue() is a method in Queue that would remove the front element

1) Stacks and Queues: Uses in Computer Science

Queue

- Event queue of all events, kept by the Java GUI system
- Queue of print jobs

Stack

Run-time stack that a processor or virtual machine keeps to organize the variables of nested methods

2) Stacks and Queues can be implemented (i.e. they are interfaces) by arrays, ArrayLists, and LinkedLists