LinkedLists:

Arrays: A collection of variables of the same type stored in contiguous memory.

Benefits: Very fast access to any arbitrary element.

Downside: Can not change size after created and can't insert or delete without needing to shift values.

Linked Lists:

Benefits: Can easily increase or decrease size as needed. Can easily insert or delete at any location.

Downside: Fast access to only a few elements.

A linked list is formed of "nodes". Each node contains an element and a pointer to the next node of the list.

The first node of the list is called the "head" of the list.

+---+---+ +---+---+ +---+---+

head -> | 1 | \*-+--> | 2 | \*-+--> | 3 | \*-+--x

+---+---+ +---+---+ +---+---+

NOTE: LinkedList is a class of the Java API. However, we are going to implement our own class so you can understand how linked lists work.

The LinkedList class of the API is identical to our class except that each box also has an array pointing to the box that comes before it.

Such a linked list is sometimes called a "double linked list".

What type should the element be? Whatever type we want to store in the list. This seems to imply that we will need to

create a different linked list class for each possible type we want to store, but starting in Java 5, we can specify a

"generic type". A generic type is a place holder (usually a single capital letter) that indicates that the type will

be specified later.

public class LLNode<T> {

indicates that there is a generic type associated with LLNode. When we create an instance of the LLNode, we will specify

what the type T is.

(A class can have as many generic types as you wish associated with it. For example, if had a class called Box with two

generic types, we would declare the class as: public class Box<K,T> { where K and T are the two letters used as

placeholders for the two generic types.)

Inside LLNode, we can use T just like any other type:

public class LLNode<T> {

private T element;

private LLNode<T> next;

public LLNode(T element, LLNode<T> next) {

this.element = element;

this.next = next;

}

public T getElement() {

return element;

}

public LLNode<T> getNext() {

return next;

}

public void setNext(LLNode<T> next) {

this.next = next;

}

}

The use of LLNode<T> in the setNext and the constructor and the next field forces Java to require that the LLNode that next points to

must hold the same type as this node holds. That way we can force every node of the list to hold the same type. This was impossible before

generics. Instead, we would need to either specify a separate list class for each type or we would have to store Object and then use lots

of instanceof expressions and typecasts to enforce that only one type of Object is stored.

When we create an instance of LLNode, we will specify what the type will be. For example, we can store JFrame:

LLNode<JFrame> node = new LLNode<JFrame>(new JFrame(), null);

or we can store String:

LLNode<String> node2 = new LLNode<String>("Hi", null);

Let us create a list that has

+------+---+ +-------+---+

node2 -> | "Hi" | \*-+--> | "Bye" | \*-+--x

+------+---+ +-------+---+

LLNode<String> node3 = new LLNode<String>("Bye", null);

node2.setNext(node3);

Generic Types and Widening/Narrowing:

Is this legal?

LLNode<String> node = new LLNode<JFrame>(new JFrame(), null);

No! We said that node will store String, but instead we gave it a LLNode that stores JFrame.

Generics and Primitive Types

The generic is used to specify an arbitrary non-primitive type. If you do not specify the generic, for example, you use

LLNode n;

Java will give you a warning message and use Object for the generic type.

We still can mix generics and primitives by using Java's wrapper classes.

Wrapper Classes:

For each primitive type (including void), Java provides a wrapper class that allows you to store the primitive inside an object. This allows us to use

primitives where the code is expecting Objects. For the most part, the wrapper class is the same name as the primitive, but with a capital letter:

Double d = new Double(4.3);

except for Integer and Character.

Starting with Java 5, Java will automatically convert between the wrapper class and the primitive, when needed and when it is clear what type is needed.

So, the following are legal:

Integer x = 5;

int y = new Integer(10);

Integer p = 4;

p = p + 1;

Warning: despite how it is written, things are not exactly as they seem.

The line "Integer p = 4" creates a new object of type Integer and stores the value 4 in it.

In the next line, the expression "p + 1" requires an int so the Integer p is replaced by the int value 4.

The second part of that line: "p = " requires that the int value 5 be placed in an Integer, and so a new Integer object is created containing 5, and p stores the address of that object,

The result is that p now has the address of a different Integer object.

Now, let us create the linked list node:

+---+---+ +---+---+

node -> | 1 | \*-+--> | 2 | \*-+--x

+---+---+ +---+---+

LLNode<Integer> node = new LLNode<Integer>(1, new LLNode<Integer>(2, null));

Note that, while we used primitive int values, Java will automatically wrap those primitives into Integer objects, and store those in the linked list.

Likewise, it will automatically unwrap the values when we retrieve them.

int z = node.getElement();

z = z + node.getNext().getElement();