1. Linked Lists :: Motivation

An array is the simplest data structure in many programming languages, including Java, but arrays have limitations which preclude their use in all situations where we need to store a list of elements. For example, suppose we declare and create an array to store the information for each student in a class:

```
Student[] roster = new Student[size];
```

The first problem with an array is that we have to specify the size of the array at the time we declare it. What if we don't know the size? Suppose we are going to open a file and read student information from the file, creating *Student* objects and storing the information in *roster*.

The standard way of doing this using arrays is to create an array which is larger than what it actually needs to be. This guarantees that as we read student information from the file and add it to *roster* that we will not run out of room.

Okay, so we can make that work. It wastes some memory, but memory is no big deal since everyone has 500 GB these days. But now consider what would happen when a student drops the course. We have to remove the *Student* object from *roster*:

thus creating a hole. A similar problem would occur when adding a new *Student* to *roster*. Ultimately, it turns out that arrays simply have too many limitations for applications which require inserting and removing elements in the middle of the list.

1. Linked Lists :: Introduction

A **linked list** is a dynamic data structure (the size of the list grows and shrinks as we add and remove elements) which permits efficient insertion and removal of elements in the middle of the list.

The data stored in the list are organized as **nodes** where each node stores an element and a **link** to the next node in the list:

The link to the next node in the list is traditionally called *next*. This particular type of linked list is referred to as a **singly linked list** because each node stores a single link. An alternative arrangement is to form a **doubly linked list** where each node stores both a link to the succeeding node (*next*) and the preceding node (*previous* or simply *prev*):

1. Linked Lists :: Introduction

The advantage of the linked list becomes apparent when we consider how we insert and remove a node in the middle of the list:

The primary disadvantage of the linked list is that **random accesses** to the list are slow:

However, if accesses are primarily sequential, then the linked list is a very efficient data structure.