**CS201**
**ArrayLists, Generics, and Dynamic Data Structures (Chapters 14, 15)**

A data structure is a software construct used to organize our data in a particular way. Some common data structures include lists, stacks, queues, and heaps. Dynamic data structures are those data structures that can grow as needed while a program is running. First we will look at a useful class provided by Java called an **arraylist**. This class uses a feature called **generics** to allow us to create lists of arbitrary data types. Then we will see how to create our own data structures, in particular, how to create dynamically linked lists.

## ArrayLists

The ArrayList class provided by Java is essentially a way to create an array that can grow or shrink in size during the execution of a program. If you recall the section on arrays, once we declared an array to be a particular size we were not able to change that size. For example, in our trivia game we declared the array to be of size 5. This means we have a maximum of 5 questions and can never exceed that amount. This is because Java allocates a specific amount of memory to hold exactly the number of items we initially specified. We could get around that problem by creating a new array of the size we like and copy the elements we want into it, but this approach is time consuming. The arraylist class does the dirty work for us to give us the same effect as a dynamic array.

If arraylists do the same thing as arrays but are dynamic, why not use them all the time? We could (and some people do) but there are two good reasons to use arrays over arraylists:

1. Arrays are more efficient than arraylists
2. The elements in an arraylist must be objects (which contributes to #1 for simple items)
3. We can't use the convenient [] notation on arraylists

If we wanted an arraylist of int's, then we instead we would have to make an arraylist of class Integer, which is a wrapper class for objects of type int (or we could make our own class).

Here is how we use a arraylist. To access the arraylist code we can import the class via:

```
import java.util.ArrayList;
```

To create a variable of type ArrayList that is a list of type DataType use the following:

```
ArrayList<DataType> varName = new ArrayList<DataType>();
```

If you know how many items the arraylist will likely need, execution can be a bit faster by specifying an initial capacity. Note however that we can still change the size later

during runtime if we want to. The example below initializes the arraylist with 50 elements:

```
ArrayList<Integer> a = new ArrayList<Integer>(50);
```

Here are methods to manipulate data in an arraylist:

public boolean add (BaseType newElement)
Adds a new object to the end of the arraylist.

Recall that all classes are derived from class Object, therefore all classes are supported as parameters for this method.

public void add(BaseType newElement, int index)
Inserts the newElement at position index and pushes everything else after this object down by one in the arraylist.

public void set(int index, BaseType newElement)
Sets an existing element at position index to newElement. Index starts at 0.

An element at this index must previously exist (i.e. can't use to add to a new position)

public BaseType get(int index)
Returns the object at position index
Index starts at 0.

public BaseType remove(int index)
Deletes the element at position index and moves everything else after this object up by one in the arraylist. Returns the object removed.

public int indexOf(Object target)
Returns the index of the first element equal to target or -1 if not found. This method invokes the equals() method of the object, so if your object does not implement and override equals() inherited from class object, most likely this method will not work properly!

public void remove(BaseType element)
Removes element from the arraylist by first searching for it using the equals method. Requires that equals be overridden.

public int size()
Returns the number of elements placed in the arraylist

Here is a simple example. Let's say we would like to make an arraylist of integers. To drive home the point that the arraylist only works with objects, we'll make our own Integer class (but we could have used the built-in Integer class too).

```
import java.util.ArrayList;

// Simple integer class with just two constructors.
// A more robust version would make the int m_value private with
// accessor methods instead

class MyIntClass
{
 public int m_value;

 public MyIntClass()
 {
      m_value = 0;
 }
 public MyIntClass(int i)
 {
      m_value = i;
 }
}

// This class illustrates common uses of the arraylist
class ArrayL
{
      public static void main(String[] args)
      {
            ArrayList<MyIntClass> v = new ArrayList<MyIntClass>();
            int i;

            // First add 4 numbers to the arraylist
            for (i=0; i<4; i++) {
                  v.add(new MyIntClass(i));
            }
            // Print out size of the arraylist, should be 4
            System.out.println("Size: " + v.size() + "\n");

            // Print arraylist
            System.out.println("Original arraylist:");
            PrintArraylist(v);

            // Remove the second element
            v.remove(2);
            // Print out the elements again
            System.out.println("After remove:");
            PrintArraylist(v);

            // Insert the number 100 at position 1
            v.add(1, new MyIntClass(100));
            // Print out the elements again
            System.out.println("After insert:");
            PrintArraylist(v);
      }
```

```
	// This method prints out the elements in the arraylist
	public static void PrintArraylist(ArrayList<MyIntClass> v)
	{
		MyIntClass temp;
		int i;

		for (i=0; i<v.size(); i++) {
			temp =  v.get(i);
			System.out.println(temp.m_value);
		}
		System.out.println();
	}
}
```

Let's try using the indexOf method:

```
		// Index of -1
		System.out.println("Position of 1");
		System.out.println(v.indexOf(new MyIntClass(1)));
```

Adding this gives an output of:

```
Position of 1
-1
```

This is not right, the list has a node with the value of 1 in position 2. We are getting -1 back because the `indexOf` method invokes the `equals` method for the BaseType object to determine which item in the arraylist matches the target. However, we didn't define one ourselves, so our program is really using the `equals` method defined for `Object` (which is inherited automatically for any object we make). However, this definition of equals only checks to see if the addresses of the objects are identical, which they are not in this case. What we really need to do is see if the integer values are the same by defining our own `equals` method. Add to the `MyIntClass` object:

```
public boolean equals(Object otherIntObject)
 {
	MyIntClass otherInt = (MyIntClass) otherIntObject;
	if (this.m_value == otherInt.m_value)
		return true;
	return false;
 }
```

If we run the program now, it will output:

```
Position of 1
2
```

A few comments of note: First, we had to define `equals` with an input parameter of type `Object`. This is because this is how the method is defined in the `Object` class which we need to override (and is invoked by the `indexOf` method). This means we have to typecast the object back to our class of `MyIntClass`. Once this is done we can compare the ints.

## For Each Loop

Iterating through an arraylist is such a common occurrence that Java includes a special type of loop specifically for this purpose. Called the for-each loop, it allows you to easily iterate through every item in the arraylist. The syntax looks like this:

```
for (BaseType varName : ArrayListVariable)
{
        Statement with varName
}
```

Here is a modified version of the PrintArrayList method from the previous example, but using the for each loop:

```
// This method prints out the elements in the arraylist
public static void PrintArraylist(ArrayList<MyIntClass> v)
{
      for (MyIntClass intObj : v)
      {
            System.out.println(intObj.m_value);
      }
      System.out.println();
}
```

Future Java courses will cover data structures called Collections (e.g. sets, maps, hash tables); you can iterate through these collections using the same for-each loop.

## Short Introduction to Generics

The ArrayList class used something new. It took as input a data type, in our case a MyIntClass definition, and used that to create the list. This type of class is called a **generic class** or a **parameterized class**. We can define our own generic classes if we wish.

We won't go into much detail here, but will just give a short example. Let's say we would like to make a class that can hold a Set of data. We can define a Set class that uses an ArrayList to store its data internally. In our add method we ensure there are no duplicate entries in the arraylist since most sets typically have no duplicates:

```
import java.util.ArrayList;

class MyGenericSet<T>         // Add <T> here to define a generic class
{
      private ArrayList<T> data;

      public MyGenericSet()
      {
            data = new ArrayList<T>();
      }
```

```
	public void add(T item)		// Use T for the generic data type
	{
		if (!data.contains(item))
			data.add(item);
	}

	public boolean contains(T item)
	{
		return (data.contains(item));
	}

	public void remove(T item)
	{
		data.remove(item);
	}
}
```

Here is a main method to test it, using sets of integers and strings:

```
	public static void main(String[] args)
	{
		MyGenericSet<String> set1 = new MyGenericSet<String>();
		set1.add("java");
		set1.add("c++");
		set1.add("php");
		System.out.println(set1.contains("java"));
		System.out.println(set1.contains("english"));

		MyGenericSet<Integer> set2 = new MyGenericSet<Integer>();
		set2.add(3);
		set2.add(55);
		System.out.println(set2.contains(20));
		System.out.println(set2.contains(55));
	}
```

**Linked Structures**

Classes can also be linked together to form dynamic lists. Recall that a variable that has a datatype of a class really corresponds to a pointer in memory to the address where the class object is stored. This is called a pointer or a reference variable. We define a class (called a node when used in a dynamic structure) that has at least two members: next (a pointer or reference to the next node in the list) and component (the type of the items on the list). For example, let's make a class to construct a list of integers.

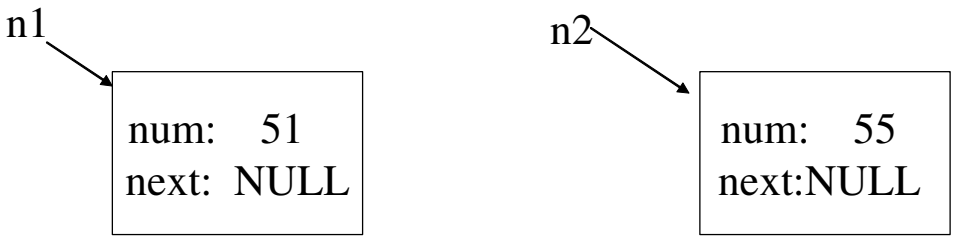
```
	public class IntNode
	{
		int num;
		IntNode next = null;

		public IntNode(int i) { num = i; }		// Constructor

		// We could add other methods here too
		// but we'll leave it at this to keep it simple for now
	}
```

To form dynamic lists, we link variables of type IntNode together to form a chain using the member variable called "next". Since this variable is of type IntNode, it is really a pointer to an IntNode object in memory. This could be a pointer to any IntNode!

Let's start to create our list by creating two separate IntNode's. We also make a variable called head, which is not set to any new object yet, but is referencing null. This will be used to remember the beginning of our list.

```
IntNode n1 = new IntNode(51);
IntNode n2 = new IntNode(55);
IntNode head = null;
```

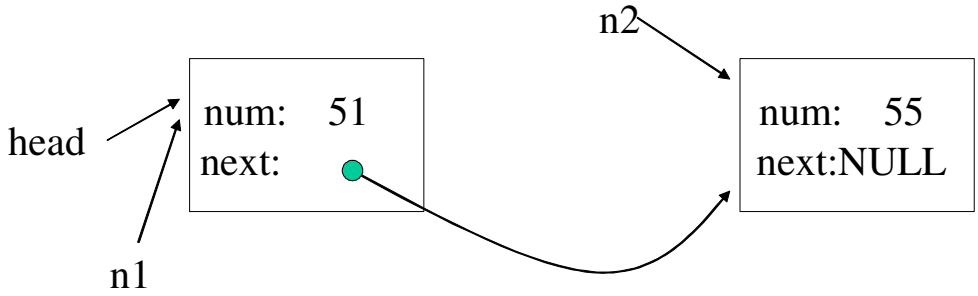
In memory we have something that looks like the following:




Next, let's connect the nodes together

```
head = n1;
n1.next = n2;
```

Here is the new picture:

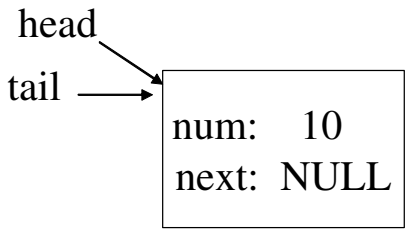



We just built a linked list consisting of two elements! The end of the list is signified by the *next* field holding NULL. Note that even if the variables n1 and n2 go away, we can still access the nodes they were pointing to by following the links from head.

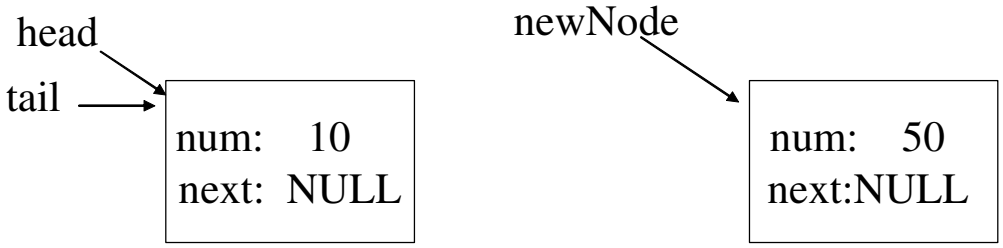
We can get a third node and have the next variable of the second node point to it. This process continues until the list is complete. Since we are constructing the list node by node, our list can grow as large as we like up to the amount of free memory. The following code fragment reads and stores integer numbers into a list until the input is –1:

```
int i;
Scanner keyboard = new Scanner(System.in);
IntNode head = null, tail = null, temp = null, newNode = null;

System.out.println("Enter value for first node.");
head = new IntNode(keyboard.nextInt());
tail = head;            // Track end of the list

System.out.println("Enter values for other nodes, -1 to stop.");
i = keyboard.nextInt();
while (i != -1) {
      // Make new node
      newNode = new IntNode(i);
      // Add it to the end of the list
      tail.next = newNode;
      // Set tail to the new tail
      tail = newNode;
      // Get next value
      i = keyboard.nextInt();
}
```

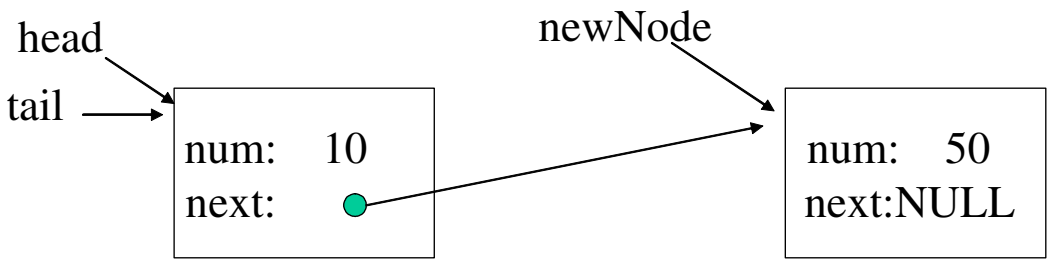
This program (it is incomplete, we'll finish it below) first inputs a number and allocates memory for head and stores the value into it. It then sets tail equal to head. tail will be used to track the end of the list while head will be used to track the beginning of the list. For example, let's say that initially we enter the value 10:



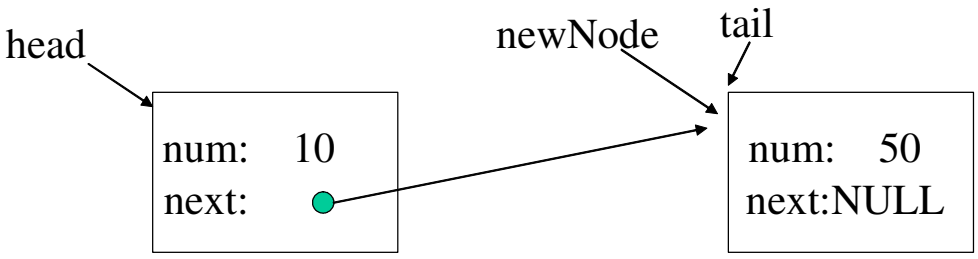

Before entering the loop, let's say that we enter 50 which is stored into i. First we create a new node, pointed to by newNode, and store the value of i into it:



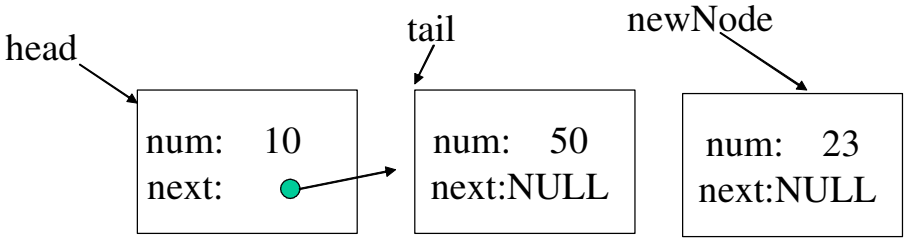
Then we link tail.next to newNode:



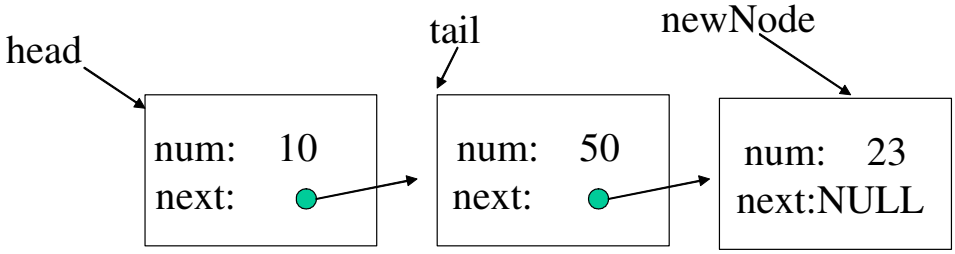

Finally we update tail to point to newNode since this has become the new end of the list:



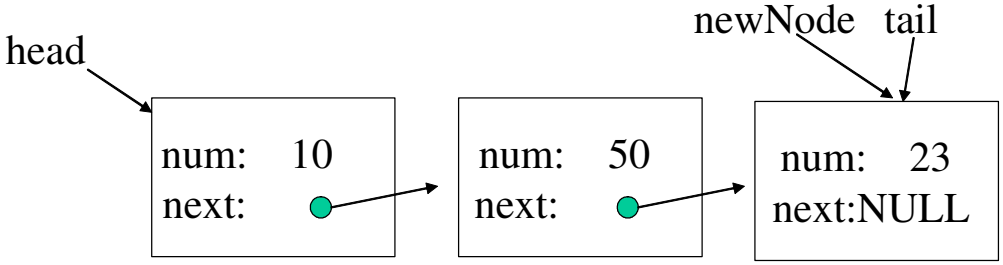

Let's say that the next number we enter is 23. We will repeat the process, first allocated a new node pointed to by newNode, and filling in its values:




Then we link tail.next to newNode:




Finally we update tail to point to the new end of the list, newNode:



The process shown above continues until the user enters –1. Note that this allows us to enter an arbitrary number of elements, up until we run out of memory! This overcomes limitations with arrays where we need to pre-allocate a certain amount of memory (that may turn out later to be too small).

Lists of dynamic variables are traversed (nodes accessed one by one) by beginning with the first node and accessing each node until the next member of a node is NULL. The following code fragment prints out the values in the list.

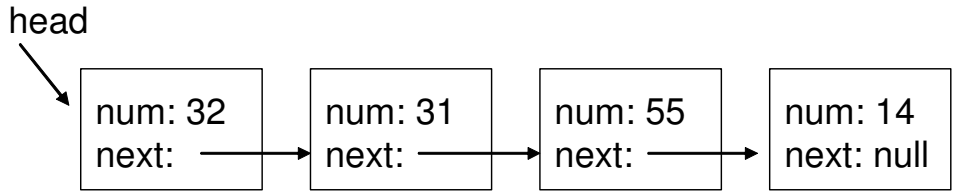
```
// List is entered, now print it out
System.out.println("\nList contains:\n");
temp = head;
while (temp != null)  {
      System.out.println(temp.num);
      temp = temp.next;
}
```

temp is initialized to head, the first node. If temp is NULL, the list is empty and the loop is not entered. If there is at least one node in the list, we enter the loop, print the member component of temp, and update temp to point to the next node in the list. temp is NULL when the last number has been printed, and we exit the loop.

Note that we are allocating memory for each node, but we never tell Java to de-allocate that memory when we're done. Java has a method called garbage collection that watches when any allocated memory has nothing referencing it. Periodically Java will "collect" all of this allocated-but-unused memory and mark it as free and available for use. As mentioned before, with many programming languages the programmer is required to explicitly de-allocate any allocated memory to prevent "memory leaks" (what happens when a program grows and grows and keeps using more and more memory until eventually the program crashes).

**Searching A Linked List**

Let's say that we have constructed the following linked list:




If we would like to search the list for some target, we merely traverse the list starting from the head:

```
temp = head;
boolean found = false;
while ((temp != null) && (!found))  {
      if (temp.num == target)
      {
            System.out.println("Found match, num = " + temp.num);
            found = true;
      }
      else
      {
            temp = temp.next;
      }
}
if (!found) System.out.println("No match found.");
```

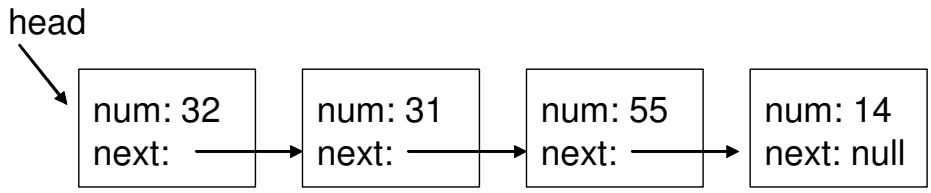
This code snippet merely walks through the linked list starting from the beginning until a match is found, at which point we stop by setting the found flag to true.

**Deleting From A Linked List**

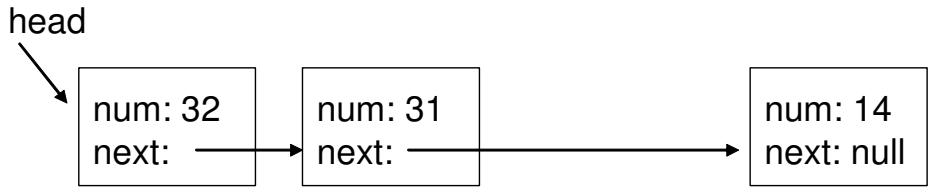
What if we would like to delete a node from the list? We could start by modifying our code to search the list:

```
temp = head;
boolean found = false;
while ((temp != null) && (!found))  {
      if (temp.num == target)  {
            // We want to delete this node
            found = true;
      }
      else  {
            temp = temp.next;
      }
}
```

This is a start, but what does it mean to delete a node? Let's say we want to delete the node with value 55 from the linked list below:



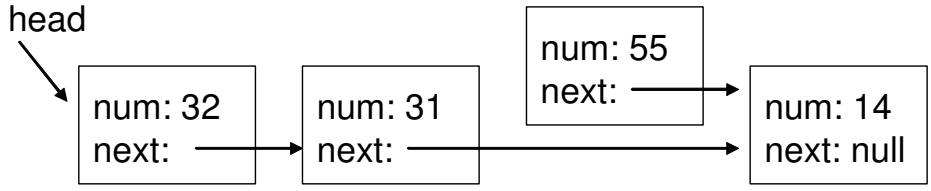

This would result in:




To delete the node with 55 what we really want to do is set the next pointer for the **previous** node to the next of the node we want to delete! But with the code we had written so far, we didn't keep track of the previous node. Here is modified code that tracks the previous:

```
currentPtr = head;
previousPtr = null;
boolean found = false;
while ((temp != null) && (!found))  {
      if (temp.num == target)
      {
            // To delete this one, set the previous next
            // to the current next
            previousPtr.next = currentPtr.next;
            found = true;
      }
      else
      {
            previousPtr = currentPtr; // Set previous to current
            currentPtr = currentPtr.next;  // Advance current
      }
}
```

Note that we have really created the following picture in deleting node 55:




Since the node with the value 55 was referencing the last node, we actually have two nodes referencing the same last node. But starting from the head of the list, there is no

way to access the node with 55. So this node is effectively deleted. During garbage collection, Java will reclaim the memory that was allocated to this node so that it can be reused for something else.

Before we exit this section, the code we have just written has a subtle flaw. Does it work correctly if we try to delete the last node in the list? What if we try to delete the first node in the list? If there is a problem, how could we fix it?