Linked Lists

Since a variable referencing an object just holds the address of the object in memory, we can link multiple objects together to form dynamic lists or other structures. In our case we will make a list. The advantage of forming lists this way is we will only use exactly the amount of storage space that is needed to hold the number of items in the list, unlike an array (or even ArrayList) which could waste some memory.

We define a record (called a node) that has at least two class variables: 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 assume that our list is a list of integers.

```
public class Node
{
 public Node()
 {
   num = 0;
   next = null;
 }
 public Node(int num)
 {
    this.num = num;
    next = null;
 }
 public int getNum()
 {
   return num;
 }
 public Node getNext()
 {
   return next;
 }
 public void setNum(int n)
 {
   num = n;
 }
 public void setNext(Node next)
 {
   this.next = next;
 }

 private int num;  // payload for the node
 private Node next; // pointer to the next node in the list
}
```

To form a dynamic linked list, we link variables of Node together to form a chain using the next data variable. We can start by making two Node objects:

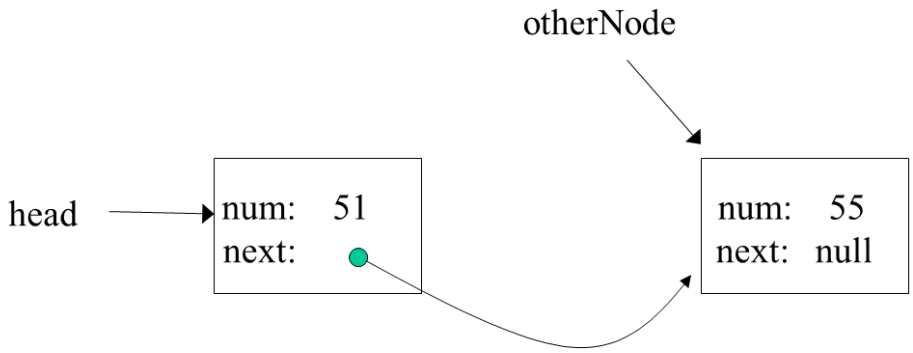
```
public static void main(String[] args)
{
      Node head = null; // First thing in the list
      Node otherNode = null;

      head = new Node(10);
      otherNode = new Node(20);
}
```

Right now we have two separate Nodes. We can link them together to form a linked list by having the next field of one pointing the address of the next node:

```
head.setNext(otherNode);
```

We now have a picture that looks like:



We just built a linked list consisting of two elements! The end of the list is signified by the *next* field holding **null**.

We can get a third node and store its address in the next member of the second node. This process continues until the list is complete. The following code fragment reads and stores integer numbers into a list until the input is –1.

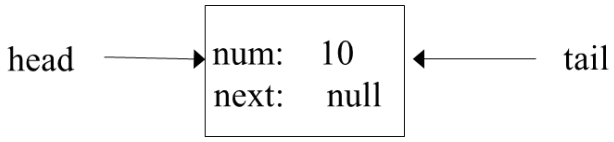
```
public static void main(String[] args)
 {
       Scanner keyboard = new Scanner(System.in);
       Node head = null; // First thing in the list
       Node tail = null; // Last thing in the list
       Node otherNode = null;
       Node tempNode = null;
       int num;

       head = new Node();
       tail = head;
       System.out.println("Enter value for the first node");
```

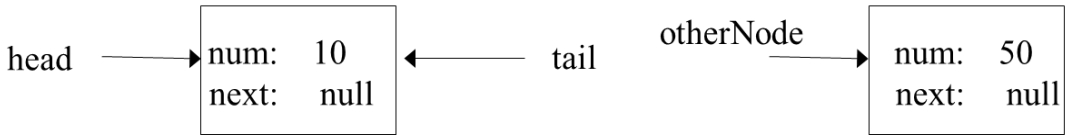
```
        num = keyboard.nextInt();
        head.setNum(num);

        System.out.println(
           "Enter values for remaining nodes, -1 to stop.");
        num = keyboard.nextInt();
        while (num != -1)
        {
                otherNode = new Node(num); // new node with num
                tail.setNext(otherNode);  // link to end of list
                tail = otherNode; // new tail
                num = keyboard.nextInt(); // get next value
        }
}
```

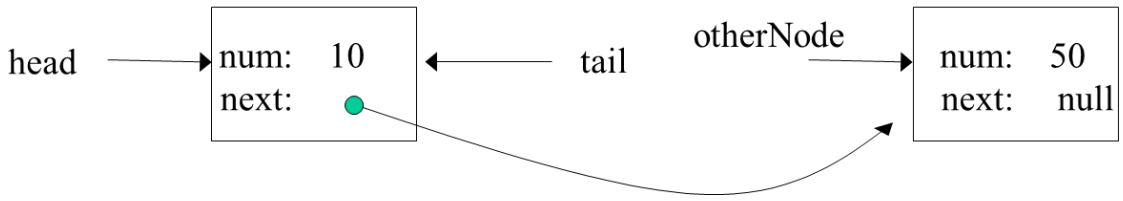
This program (it is incomplete, we'll finish it below) first allocates memory for head and inputs a value into it. It then sets tail equal to head. tail will be used to track the end of the list while head will be used to track the beginning of the list. For example, let's say that initially we enter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



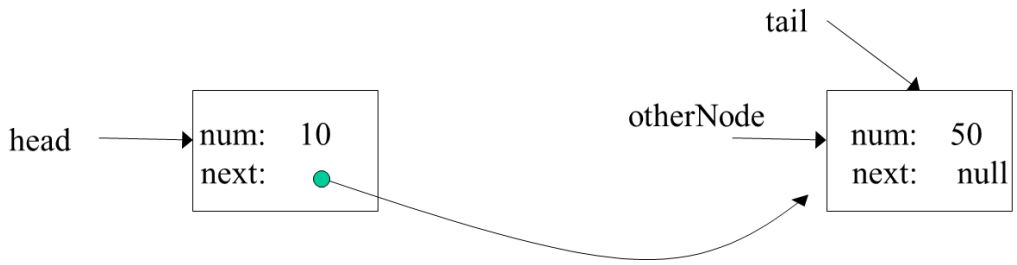
Upon entering the loop, let's say that we enter 50 which is stored into num. First we create a new node, referenced by otherNode, and store data into it:



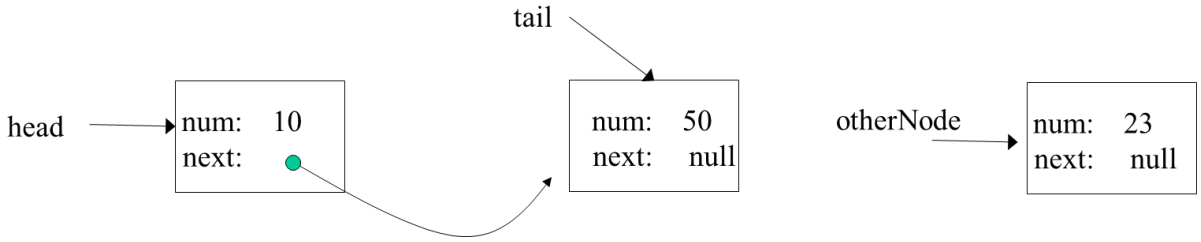
Then we link tail's next to otherNode:



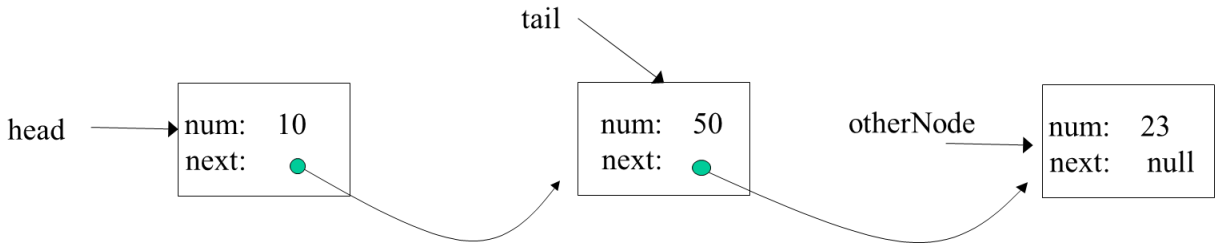
Finally we update tail to point to otherNode since this has become the new end of the list:



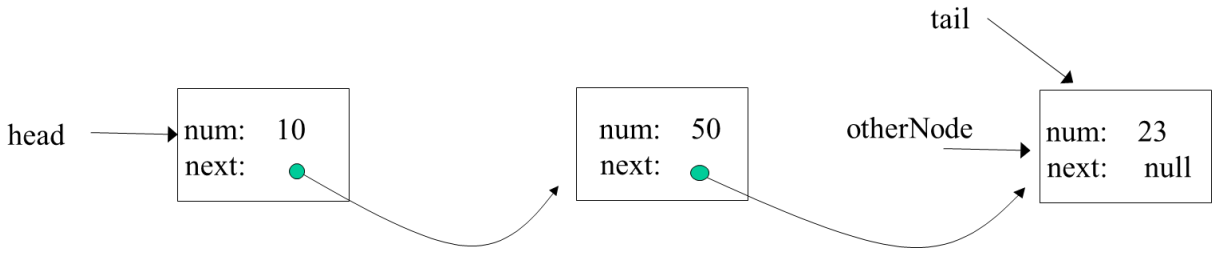
Let's say that the next number we enter is 23. We will repeat the process, first allocated a new node pointed to by otherNode, and filling in its values:



Then we link tail's next to otherNode:



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



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.

```
System.out.println("Printing out the list");
tempNode = head;
while (tempNode != null)
{
        System.out.println(tempNode.getNum());
        tempNode = tempNode.getNext();
}
```

tempNode is initialized to head, the first node. If tempNode 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 tempNode, and update tempNode to point to the next node in the list. tempNode is null when the last number has been printed, and we exit the loop.

Because the types of a list node can be any data type, we can create an infinite variety of lists. Pointers also can be used to create very complex data structures such as stacks, queues, and trees that are the subject of more advanced computer science courses.

To try:

- Create a class that uses a linked list internally to provide the following functionality:

  MyStringList

addString(String s)	- adds s to the front of the list
deleteString(String s)	- removes first s from the list
boolean contains(String s)	- true if s is in the list
void output()	- print out everything in the list

  This would be a good candidate to make into a template so we could make a list of any type instead of only strings!

- Something to think about and perhaps try: create a doubly-linked list, which has references to the previous and to the next Node in the list; this should make it easier to delete from the list, but there is more to update when we add to the list.