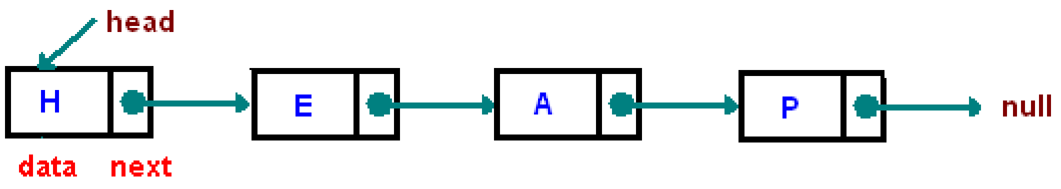
Linked Lists

Introduction

One disadvantage of using arrays to store data is that arrays are static structures and therefore cannot be easily extended or reduced to fit the data set. Arrays are also expensive to maintain new insertions and deletions. In this chapter we consider another data structure called Linked Lists that addresses some of the limitations of arrays.

A linked list is a linear data structure where each element is a separate object.



Each element (we will call it a **node**) of a list is comprising of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the **head** of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the head is a null reference.

A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list.

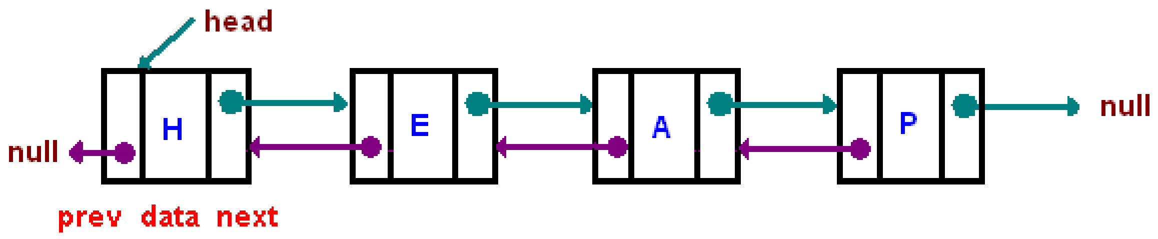
One disadvantage of a linked list against an array is that it does not allow direct access to the individual elements. If you want to access a particular item then you have to start at the head and follow the references until you get to that item.

Another disadvantage is that a linked list uses more memory compare with an array - we extra 4 bytes (on 32-bit CPU) to store a reference to the next node.

Types of Linked Lists

A **singly linked list** is described above

A **doubly linked list** is a list that has two references, one to the next node and another to previous node.



Another important type of a linked list is called a **circular linked list** where last node of the list points back to the first node (or the head) of the list.

The Node class

In Java you are allowed to define a class (say, B) inside of another class (say, A). The class A is called the outer class, and the class B is called the **inner** class. The purpose of inner classes is purely to be used internally as helper classes. Here is the LinkedList class with the inner Node class

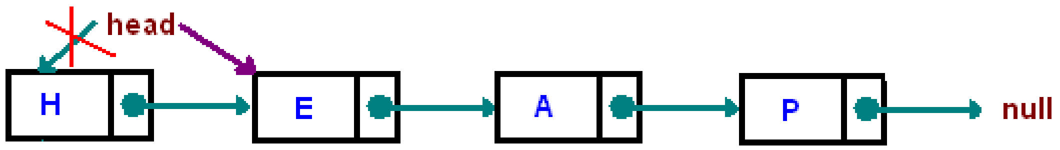
```
private static class Node<AnyType>
{
   private AnyType data;
   private Node<AnyType> next;

   public Node(AnyType data, Node<AnyType> next)
   {
      this.data = data;
      this.next = next;
   }
}
```

An inner class is a member of its enclosing class and has access to other members (inclusing private) of the outer class, And vise versa, the outer class can have a direct access to all members of the inner class. An inner class can be declared private, public, protected, or package private. There are two kind of inner classes: static and non-static. A static inner class cannot refer directly to instance variables or methods defined in its outer class: it can use them only through an object reference.

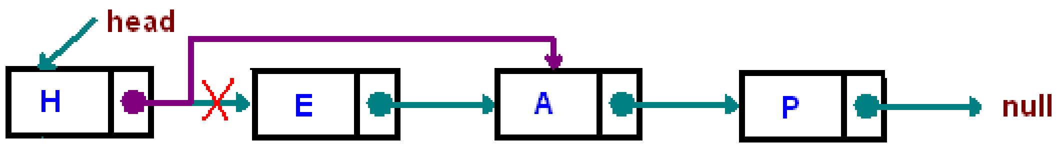
We implement the LinkedList class with two inner classes: static Node class and non-static LinkedListIterator class. See LinkedList.java for a complete implementation.

Examples
Let us assume the singly linked list above and trace down the effect of each fragment below. The list is restored to its initial state before each line executes

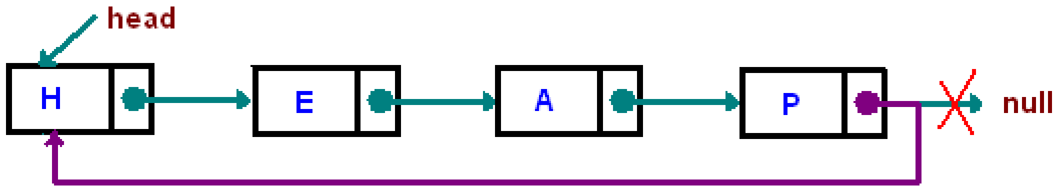
1. `head = head.next;`



2. `head.next = head.next.next;`



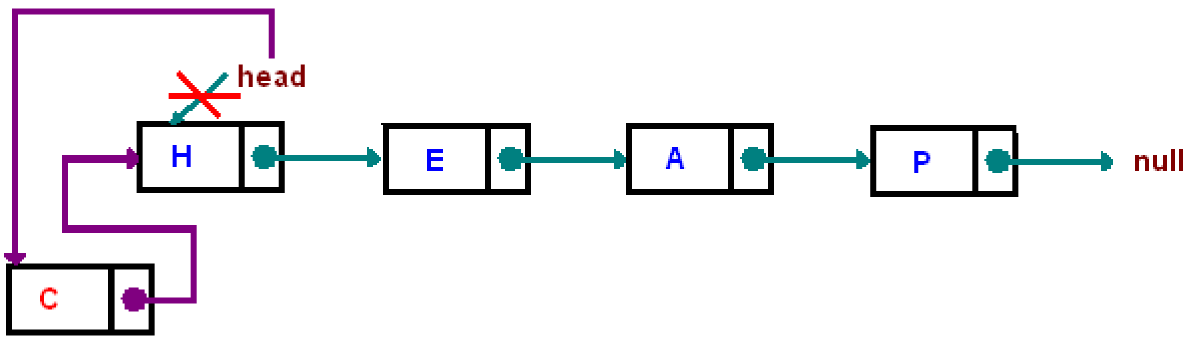
3. `head.next.next.next.next = head;`



Linked List Operations

addFirst

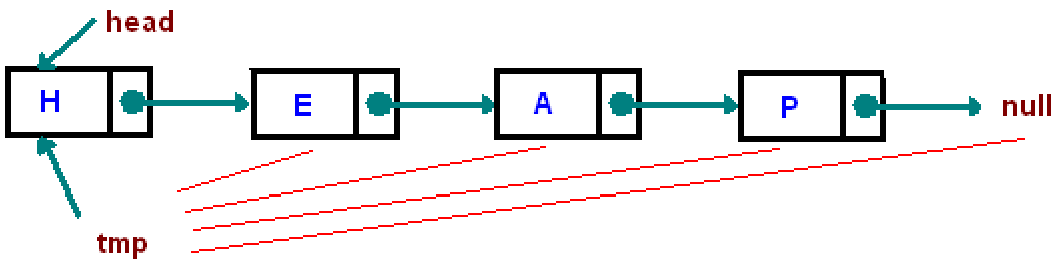
The method creates a node and prepends it at the beginning of the list.



```
public void addFirst(AnyType item)
{
   head = new Node<AnyType>(item, head);
}
```

Traversing

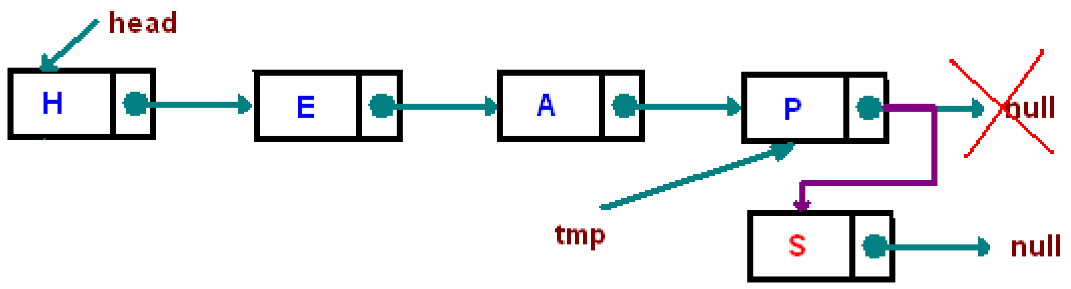
Start with the head and access each node until you reach null. Do not change the head reference.



```
Node tmp = head;

while(tmp != null) tmp = tmp.next;
```

addLast

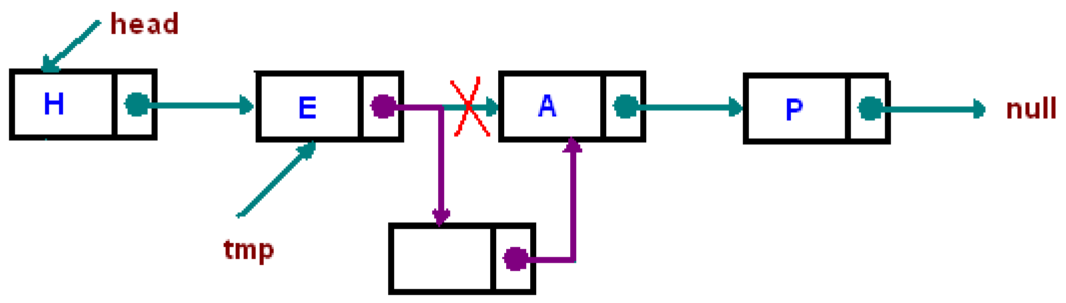
The method appends the node to the end of the list. This requires traversing, but make sure you stop at the last node



```
public void addLast(AnyType item)
{
   if(head == null) addFirst(item);
   else
   {
      Node<AnyType> tmp = head;
      while(tmp.next != null) tmp = tmp.next;

      tmp.next = new Node<AnyType>(item, null);
   }
}
```

Inserting "after"

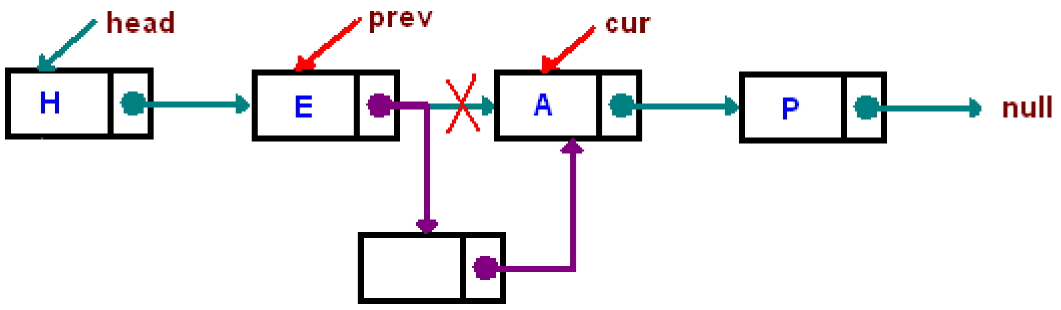
Find a node containing "key" and insert a new node after it. In the picture below, we insert a new node after "e":



```
public void insertAfter(AnyType key, AnyType toInsert)
{
   Node<AnyType> tmp = head;
   while(tmp != null && !tmp.data.equals(key)) tmp = tmp.next;

   if(tmp != null)
      tmp.next = new Node<AnyType>(toInsert, tmp.next);
}
```

Inserting "before"

Find a node containing "key" and insert a new node before that node. In the picture below, we insert a new node before "a":



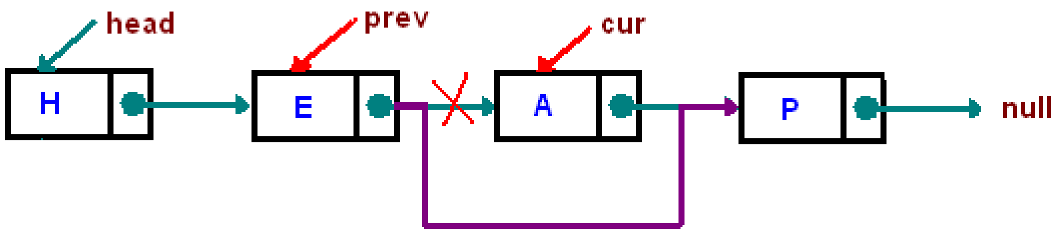
For the sake of convenience, we maintain two references `prev` and `cur`. When we move along the list we shift these two references, keeping `prev` one step before `cur`. We continue until `cur` reaches the node before which we need to make an insertion. If `cur` reaches null, we don't insert, otherwise we insert a new node between `prev` and `cur`.

Examine this implementation

```
public void insertBefore(AnyType key, AnyType toInsert)
{
   if(head == null) return null;
   if(head.data.equals(key))
   {
      addFirst(toInsert);
      return;
   }

   Node<AnyType> prev = null;
   Node<AnyType> cur = head;

   while(cur != null && !cur.data.equals(key))
   {
      prev = cur;
      cur = cur.next;
   }
   //insert between cur and prev
   if(cur != null) prev.next = new Node<AnyType>(toInsert, cur);
}
```

Deletion

Find a node containing "key" and delete it. In the picture below we delete a node containing "A"



The algorithm is similar to insert "before" algorithm. It is convinient to use two references `prev` and `cur`. When we move along the list we shift these two references, keeping `prev` one step before `cur`. We continue until `cur` reaches the node which we need to delete. There are three exceptional cases, we need to take care of:

1. list is empty
2. delete the head node
3. node is not in the list

```
public void remove(AnyType key)
{
   if(head == null) throw new RuntimeException("cannot delete");

   if( head.data.equals(key) )
   {
      head = head.next;
      return;
   }

   Node<AnyType> cur  = head;
   Node<AnyType> prev = null;

   while(cur != null && !cur.data.equals(key) )
   {
      prev = cur;
      cur = cur.next;
   }

   if(cur == null) throw new RuntimeException("cannot delete");

   //delete cur node
   prev.next = cur.next;
}
```

Iterator

The whole idea of the iterator is to provide an access to a private aggregated data and at the same moment hiding the underlying representation. An iterator is Java is an object, and therefore it's implementation requires creating a class that implements the *Iterator* interface. Usually such class is implemented as a private inner class. The *Iterator* interface contains the following methods:

- AnyType next() - returns the next element in the container
- boolean hasNext() - checks if there is a next element
- void remove() - (optional operation).removes the element returned by next()

In this section we implement the Iterator in the LinkedList class. First of all we add a new method to the LinkedList class:

```
public Iterator<AnyType> iterator()
{
   return new LinkedListIterator();
}
```

Here `LinkedListIterator`is a private class inside the LinkedList class

```
private class LinkedListIterator implements Iterator<AnyType>
{
   private Node<AnyType> nextNode;
```

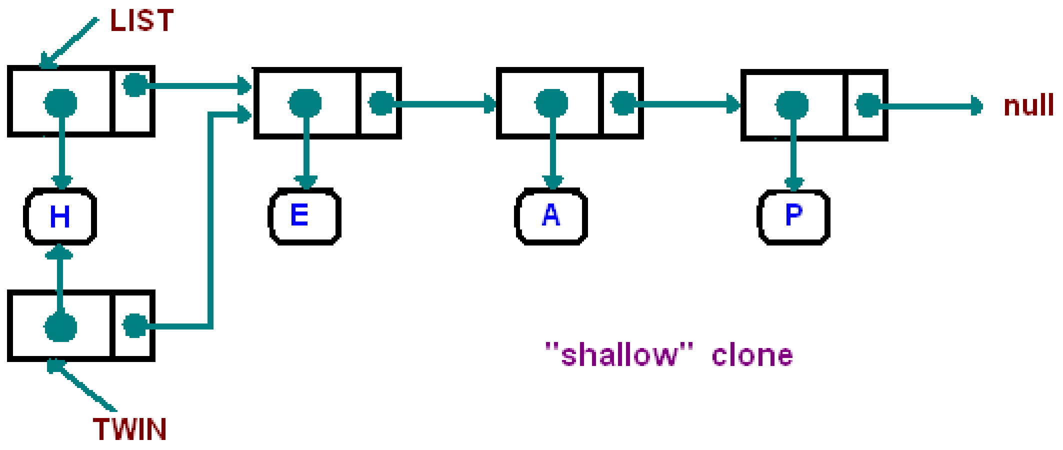
```
           public LinkedListIterator()
           {
              nextNode = head;
           }
           ...
        }
```

The `LinkedListIterator` class must provide implementations for `next()` and `hasNext()` methods. Here is the `next()` method:

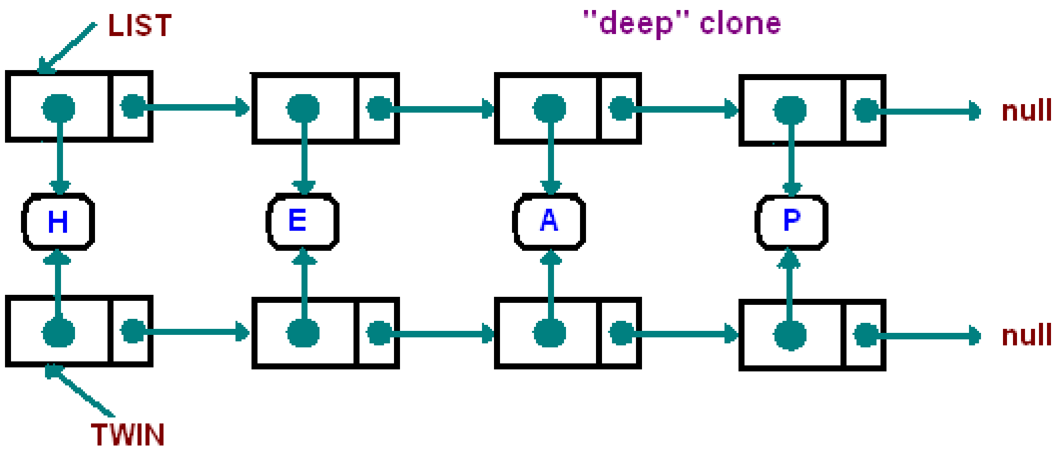
```
        public AnyType next()
        {
           if(!hasNext()) throw new NoSuchElementException();
           AnyType res = nextNode.data;
           nextNode = nextNode.next;
           return res;
        }
```

Cloning

Like for any other objects, we need to learn how to clone linked lists. If we simply use the clone() method from the Object class, we will get the following structure called a "shallow" copy:



The Object's clone() will create a copy of the first node, and share the rest. This is not exactly what we mean by "a copy of the object". What we actually want is a copy represented by the picture below



Since out data is immutable it's ok to have data shared between two linked lists. There are a few ideas to implement linked list copying. The simplest one is to traverse the original list and copy each node by using the addFirst() method. When this is finished, you will have a new list in the reverse order. Finally, we will have to reverse the list:

```
public Object copy()
{
   LinkedList<AnyType> twin = new LinkedList<AnyType>();
   Node<AnyType> tmp = head;
   while(tmp != null)
   {
      twin.addFirst( tmp.data );
      tmp = tmp.next;
   }

   return twin.reverse();
}
```

A better way involves using a tail reference for the new list, adding each new node after the last node.

```
public LinkedList<AnyType> copy3()
{
   if(head==null) return null;
   LinkedList<AnyType> twin = new LinkedList<AnyType>();
   Node tmp = head;
   twin.head = new Node<AnyType>(head.data, null);
   Node tmpTwin = twin.head;

   while(tmp.next != null)
   {
      tmp = tmp.next;
      tmpTwin.next = new Node<AnyType>(tmp.data, null);
      tmpTwin = tmpTwin.next;
   }

   return twin;
}
```

Applications

Polynomial Algebra

The biggest integer that we can store in a variable of the type int is $2^{31} - 1$ on 32-but CPU. You can easily verify this by the following operations:

```
int prod=1;
for(int i = 1; i <=; 31; i ++)
        prod *= 2;
System.out.println(prod);
```

This code doesn't produce an error, it produces a result! The printed value is a *negative* integer $-2147483648 = -2^{31}$. If the value becomes too large, Java saves only the low order 32 (or 64 for longs) bits and throws the rest away.

In real life applications we need to deal with integers that are larger than 64 bits (the size of a long). To manipulate with such big numbers, we will be using a linked list data structure. First we observe that each integer can be expressed in the decimal system of notation.

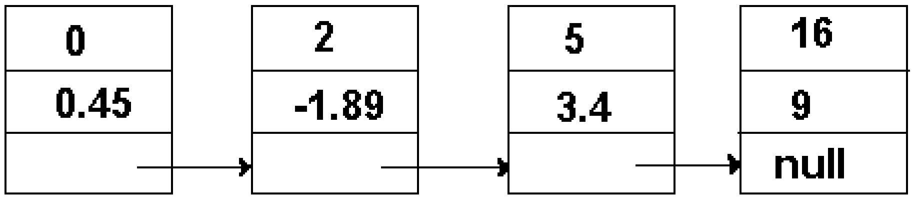
$937 = 9*10^2 + 3*10^1 + 7*10^0$

$2011 = 2*10^3 + 0*10^2 + 1*10^1 + 1*10^0$

Now, if we replace a decimal base 10 by a character, say 'x', we obtain a univariate polynomial, such as

$$0.45 - 1.89\,x^2 + 3.4\,x^5 + 9\,x^{16}$$

We will write an application that manipulates polynomials in one variable with real coefficients.Among many operations on polynomials, we implement addition, multiplication, differentiation and evaluation. A polynomial willbe represented as a linked list, where each node has an integer degree, a double coefficient and a reference to the next term. The final node will have a null reference to indicate the end of the list. Here is a linked link representation for the above polynomial:



The terms are kept in order from smallest to largest exponent. See Polynomial.java for details.

Victor S.Adamchik, CMU, 2009