Introduction to Algorithms and Data Structures

Lecture 10 - An Introduction to Self-Referential Objects and Linked Lists

Values and Memory

- When we declare a variable x by writing

```
int    x;
```

we are allocating a location in memory for an integer value x. Any reference to x in the program refers to the integers stored at that location, e.g.,

```
x = 5;
```

How do Java Objects Work?

- When we declare an object x by writing

  ```
  Integer    x;
  ```

  we are allocating a location in memory to store a memory address – at which the object x will be located if we allocate memory for it.
- To allocate storage for the object, we write

  ```
  x = new Integer();
  ```

Objects and Self-References

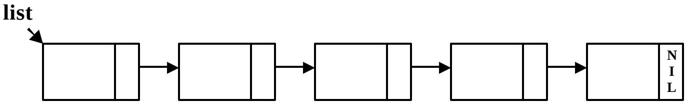
- We can define a class of objects in which one of the properties is another object of the same class:

  ```
  public class SelfReference  {
     private int myData;
     private double yourData;
     SelfReference someoneElsesData;
  }
  ```

- Every object of the class SelfReference contains the address at which another object is stored.

What is a Linked List?

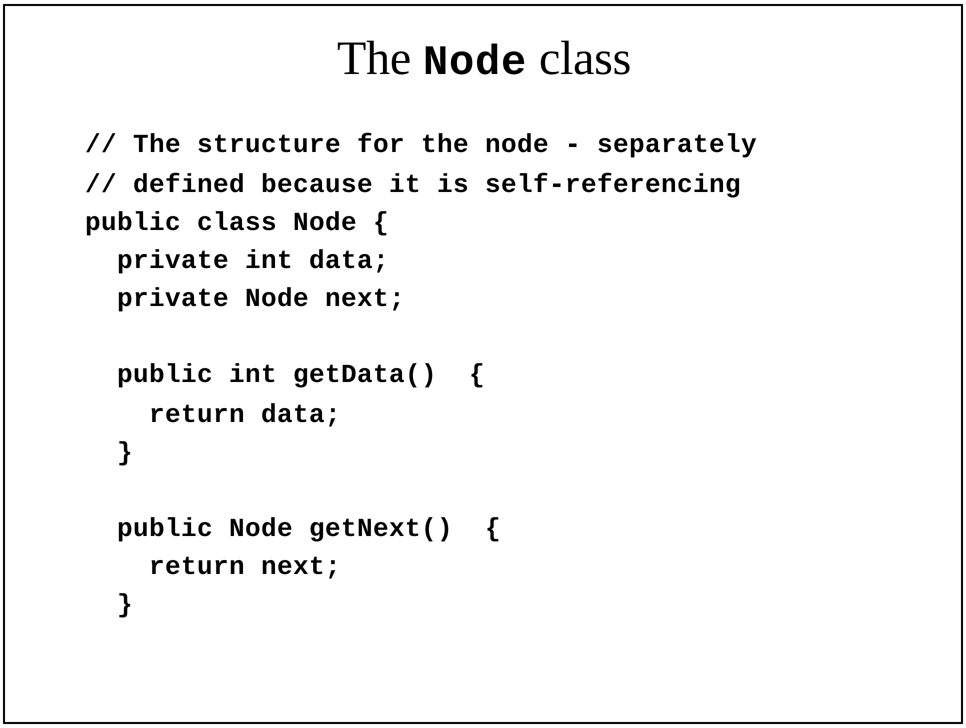
- A linked list is a collection of data items that include a pointer to the next data item in the collection.
- Each item in the collection is called a node and contains both data as well as a pointer to the next node on the list.



Basic List Operations

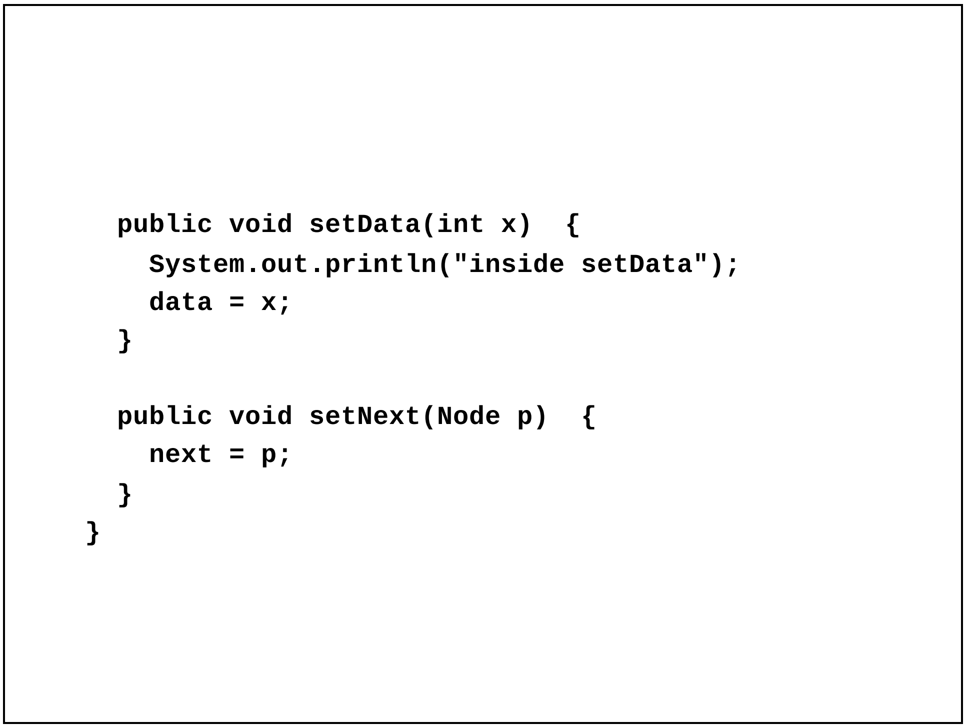
The basic operations performed on the list include:

- – Creating a new node
- – Inserting a node at the front of the list
- – Inserting a node at the end of the list
- – Inserting a new node after an existing node on the list
- – Determining if a value is stored on the list
- – Removing a node from the list



The **Node** class

```
// The structure for the node - separately
// defined because it is self-referencing
public class Node {
  private int data;
  private Node next;

  public int getData()  {
    return data;
  }

  public Node getNext()  {
    return next;
  }
```



```
  public void setData(int x)  {
    System.out.println("inside setData");
    data = x;
  }

  public void setNext(Node p)  {
    next = p;
  }
}
```

The **`LinkedList`** class

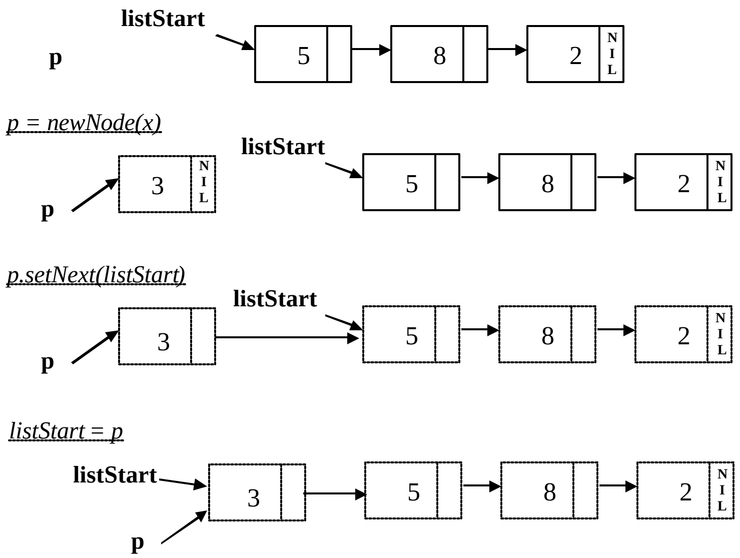
```
public class LinkedList {
   private Node listStart;

   // list() - The default constructor - Starts
   //          the list as empty
  public LinkedList()  {
    listStart = null;
  }
```

```
  // list() - An initializing constructor that
  //          creates a node and places in it the
  //          initial value
  public LinkedList(int x)  {
    listStart = new Node();
    listStart.setData(x);
    listStart.setNext(null);
  }
```

```
// newNode() - Creates a new node with a zero
//             as data by default
public Node newNode()  {
  Node p = new Node();
  p.setData(0);
  p.setNext(null);
  return p;
}

// newNode() - Creates a new node with the
//    parameter x as its value
public Node newNode(int x)  {
  Node p = new Node();
  p.setData(x);
  p.setNext(null);
  return p;
}
```

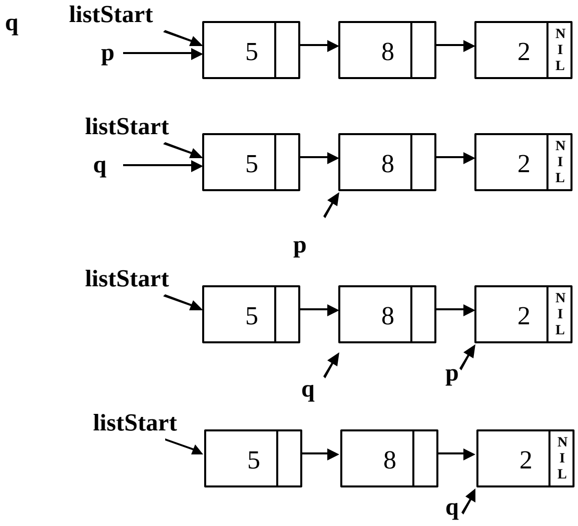
```
// addFront() - Inserts a new node containing x
//              at the front of the list
public void addFront(int x)  {
  Node p = newNode(x);
  p.setNext(listStart);
  listStart = p;
}
```

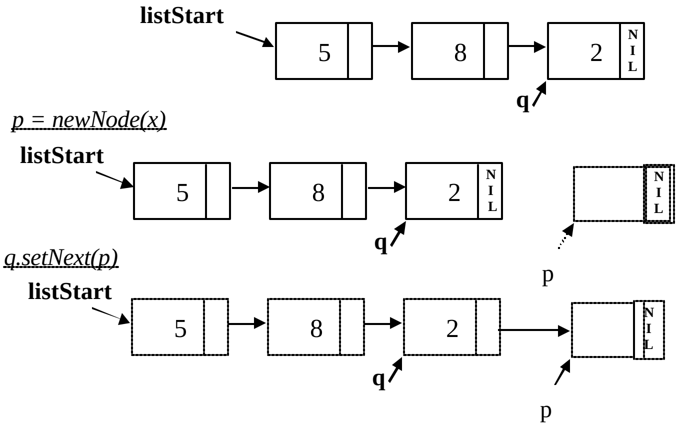
Tracing `addfront`



```
// addRear() -  Inserts a new node containing x
//              at the rear of the list
public void addRear(int x)  {
  Node p, q;

  // Scan through the list to find the end
  // q points to the last node
  for (p = listStart, q = null; p != null;
                     q = p, p = p.getNext())
    ;

  // Invariant - p must be NULL so we use it to
  //  hold a pointer to the new node
  p = newNode(x);
  q.setNext(p);
}
```

Tracing **`addrear`**



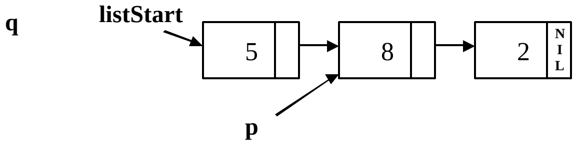


p = newNode(x)

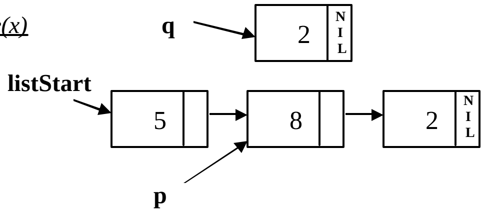
q.setNext(p)

```
// insertAfter() -  Insert value x in a new
//                  node to be inserted
//                  after p
public void insertAfter(int x, Node p)  {
  Node  q = newNode(x);
  q.setNext(p.getNext());
  p.setNext(q);
}
```

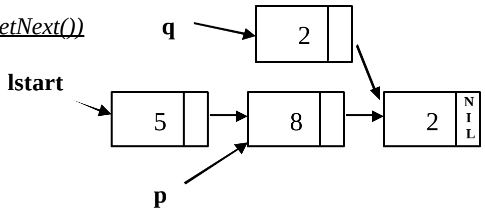
Tracing `insertafter`



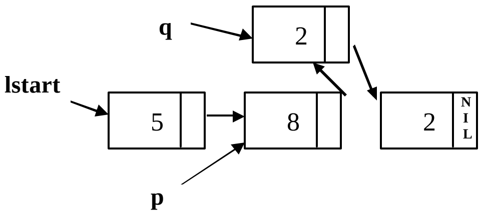
q = newNode(x)



q.setNext(p.getNext())



p.setNext(q)



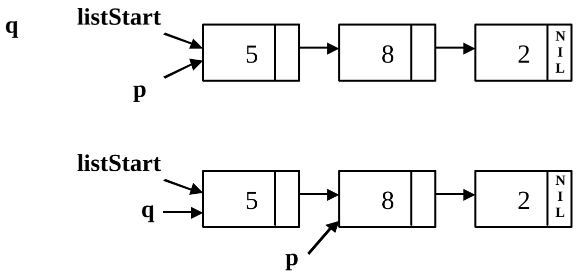
```
// isXThere() - Is there a node on the list
//              containing x?
public boolean isXThere(int x)  {
  Node p = listStart;

  if (p == null)
    return false;
  else {
  // Scan through the list looking for x
  while (p != null && p.getData() != x)
     p = p.getNext();
    // Invariant - either p contains x or we have
    // gone through the entire list
    if (p == null)
      return(false);
    else
      return true;
    }
  }
```

```
// find() - Get the node containing x
public Node find(int x)  {
  Node p;
  // Scan through the list looking for x
  for (p = listStart; p != null && p.getData()!= x;
                      p = p.getNext())
    ;

  if (p != null)
    // p contains x
    return p;
  else
    // We searched through the whole list and
    // x wasn't there
    return(null);
}
```

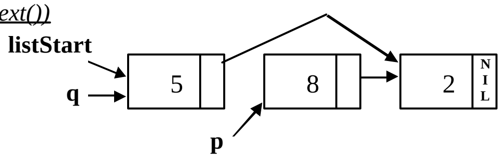
```
// removenode() - Remove the node containing x
//                from the list
public void removeNode(int x)  {
  Node p, q;

  // Scan through the list - is x there?
  for (p = listStart, q = null;
             p != null && p.getData() != x;
             p = p.getNext())
    q = p;
```

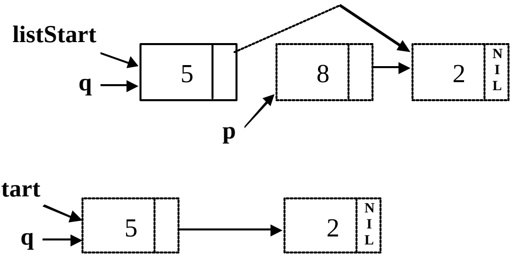
```
  // If so, remove it
  if (p!= null)  {
    if (q == null)
      // x is at the front
      // Re-adjust the pointer to the
      // front of the list
      listStart = p.getNext();
    else
      // Splice it out of the list
      q.setNext(p.getNext());

  }
}
```

Tracing **`removenode`**



q.setNext(p.getNext())





```
// writeLinkedList() - Write the data contents
//                     of every node on the
//                     list
public void writeLinkedList()  {
  Node p;

  for  (p = listStart; p != null;
                       p = p.getNext())
    System.out.println(p.getData());
}
```

Rewriting List Operations Using Recursion

- Lists can also be traversed recursively.
- The simple case is when the list’s reference is null.
- In other case, you do what needs to be deon with the first node and recursively act on the rest of the list (sometimes acting on the last node as well).

```
  // travLinkedList() -  Traverse a list
  //                     recursively using the
  //                     function trav
  public void travLinkedList()  {
    if (listStart != null)
      trav(listStart);
    System.out.println();
  }
```

```
  // trav() - The auxiliary traversal
  //          function that is used
  //          recursively.
  private void trav(Node p)  {
    if (p != null) {
      System.out.print(p.getData() + "\t");
      trav(p.getNext());
    }
  }
}
```

The **`TestLinkedList`** class

```
public class TestLinkedList  {
  public static void main(String[] args)  {
    LinkedList  myLinkedList = new LinkedList();

    myLinkedList.writeLinkedList();

    myLinkedList.addFront(12);
    myLinkedList.writeLinkedList();
    System.out.println();

    myLinkedList.addRear(1);
    myLinkedList.removeNode(12);
    myLinkedList.writeLinkedList();
  }
}
```