# Chapter 9

# Data Structures: Linked Lists

CSC 113
King Saud University
College of Computer and Information Sciences
Department of Computer Science

Dr. S. HAMMAMI

# Objectives

**After you have read and studied this chapter, you should be able to:**

- Understand the concept of a dynamic data structure.
- Be able to create and use dynamic data structures such as linked lists.
- Understand the stack and queue ADTs.
- Various important applications of linked data structures.
- Know how to use inheritance to define extensible data structures.
- Create reusable data structures with classes, inheritance and composition.

# Outline

1. **Introduction**
2. **Self-Referential Classes**
   - 2.1. Definition
   - 2.2. Generic Node Class
   - 2.3. Example
   - 2.4. Implementation of Generic Class Node
   - 2.5. Connecting two nodes
   - 2.6. Examples

3. **Linked Lists**
   - 3.1. Definition
   - 3.2. Graphical representation
   - 3.3. Performance
   - 3.4. Single Linked List
   - 3.5. Basics Methods of Linked List: Implementation
   - 3.6. Examples

# 1. Introduction

- A *data structure* is organizes information so that it efficient to access and process.
- An *array* is a *static* structure -- it can’t change size once it is created.
- A *vector* is a *dynamic* structure -- it can grow in size after creation.
- In this chapter we study several dynamic data structures -- *lists*, *queues*, and *stacks*.

# 2. Self-Referential Classes: Definition

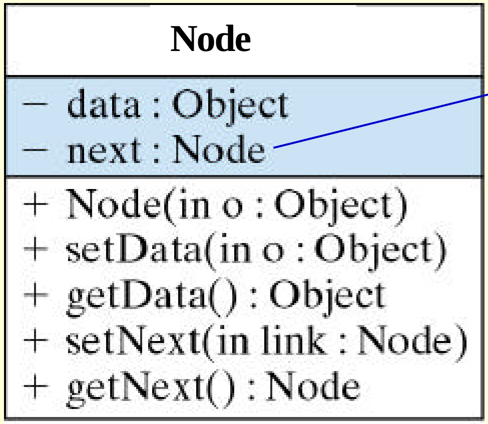
- **Self-referential class**

  **Contains an instance variable that refers to another object of the same class type**

  **That instance variable is called a link**

  **A `null` reference indicates that the link does not refer to another object**

# 2. Self-Referential Classes (cont)

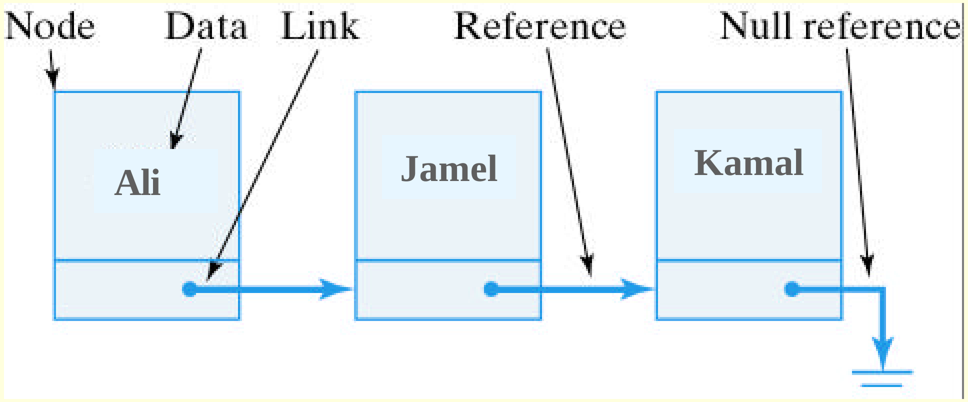


**Node**

− data : Object
− next : Node

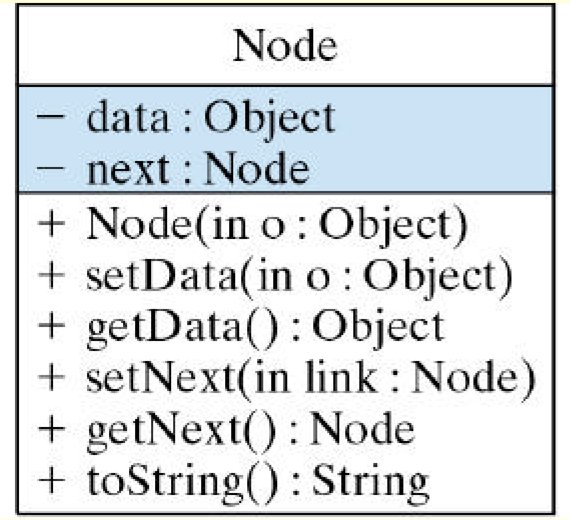
+ Node(in o : Object)
+ setData(in o : Object)
+ getData() : Object
+ setNext(in link : Node)
+ getNext() : Node

A *link* to another Node object.



# Basic Node: The Generic Node Class

- A node in a linked list contains data elements and link elements.



Node

− data : Object
− next : Node

+ Node(in o : Object)
+ setData(in o : Object)
+ getData() : Object
+ setNext(in link : Node)
+ getNext() : Node
+ toString() : String

# Generic Node Class: Implementation

```
public class Node {
    private Object data;  // Stores any kind of data
    private Node next;

    public Node(Object obj) {  // Constructor
        data = obj;
        next = null;
    }

  // Link access methods
    public void setNext( Node nextPtr ) {
        next = nextPtr;
    }

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

```
// Data access methods
public void setData(Object obj)
{
    data = obj;
}

public Object getData() {
    return data;
}

public String toString() {
    return data.toString();
}
```

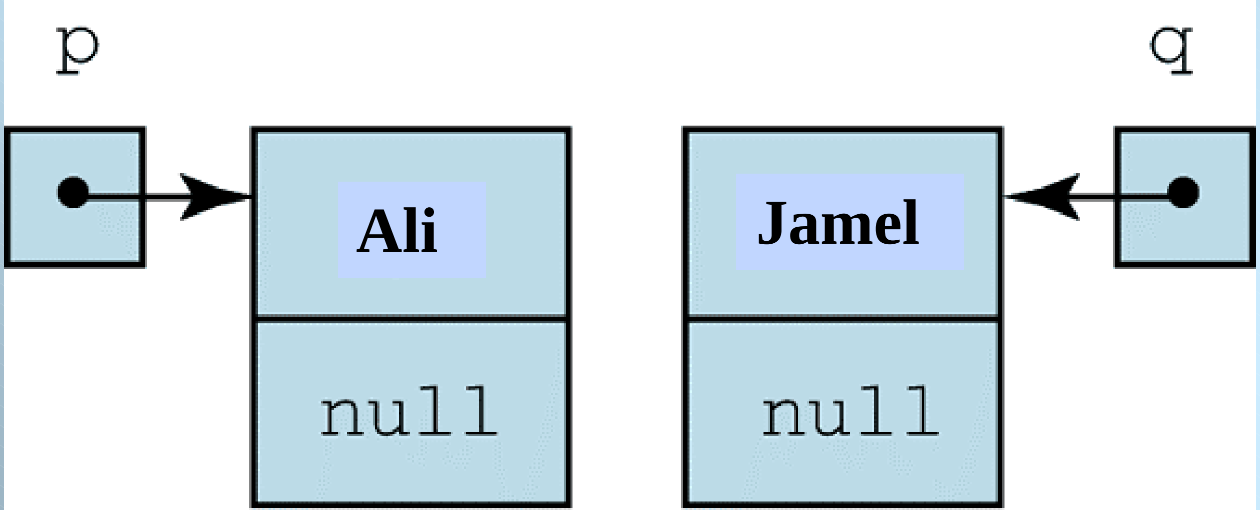
# Connecting two nodes

The statements

```
Node p = new Node(“Ali");
Node q = new Node(“Jamel");
```

allocate storage for two objects of type **Node** referenced by **p** and **q**. The node referenced by **p** stores the string **“Ali"**, and the node referenced by **q** stores the string **“Jamel"**. The **next** fields of both nodes are **null**.

*Nodes referenced by p and q*



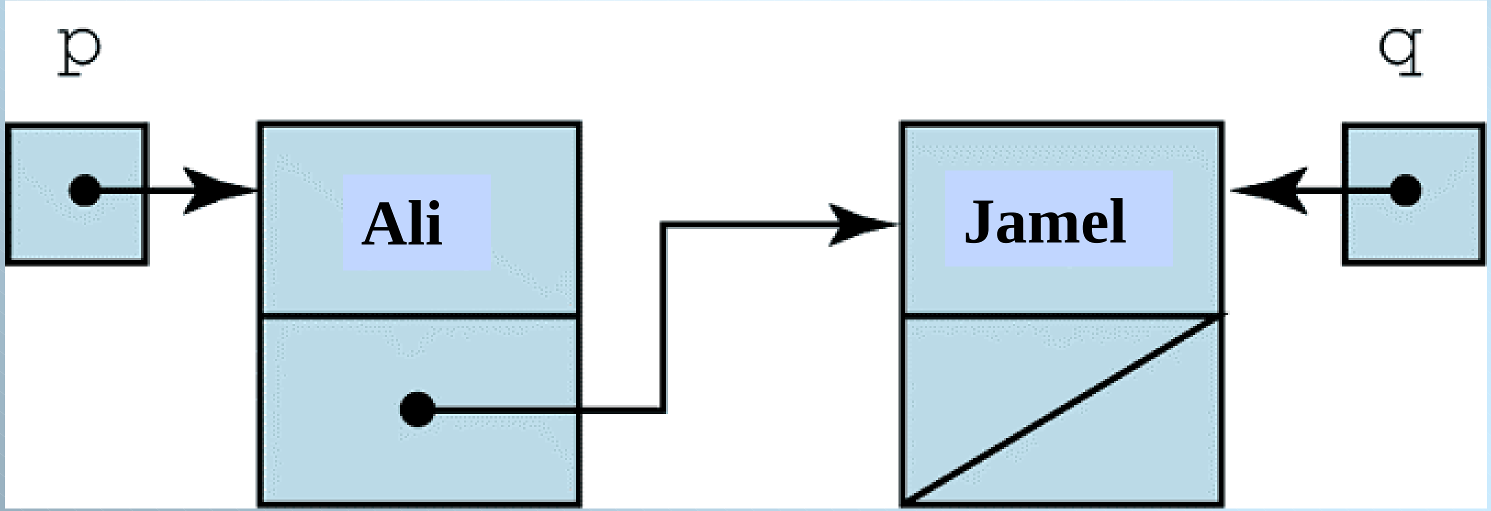
# Connecting two nodes (Cont)

The statement

**p.next = q;**

stores the address of node **q** in the link field of node **p**, thereby connecting node **p** to node **q**, and forming a linked list with 2 nodes. The diagonal line in the **next** field of the second list node indicates the value **null**.

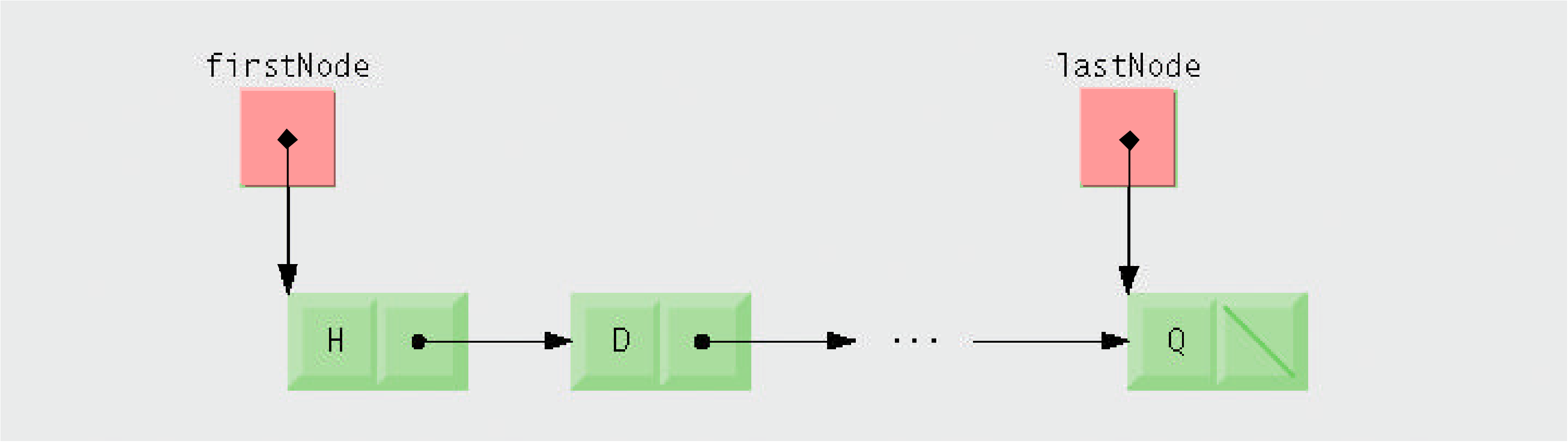
***Linked list with two nodes***



# 3. Linked Lists: Definition

- Linked list
  - – Linear collection of nodes
    - A *linked list* is based on the concept of a **self-referential object** -- an object that refers to an object of the same class.
  - – A program typically accesses a linked list via a reference to the first node in the list
    - A program accesses each subsequent node via the link reference stored in the previous node
  - – Are dynamic
    - The length of a list can increase or decrease as necessary
    - Become full only when the system has insufficient memory to satisfy dynamic storage allocation requests

# Linked list graphical representation.



# Linked Lists: Performance

- An array can be declared to contain more elements than the number of items expected, but this wastes memory. Linked lists provide better memory utilization in these situations. Linked lists allow the program to adapt to storage needs at runtime.

- Insertion into a linked list is fast—only two references have to be modified (after locating the insertion point). All existing node objects remain at their current locations in memory.

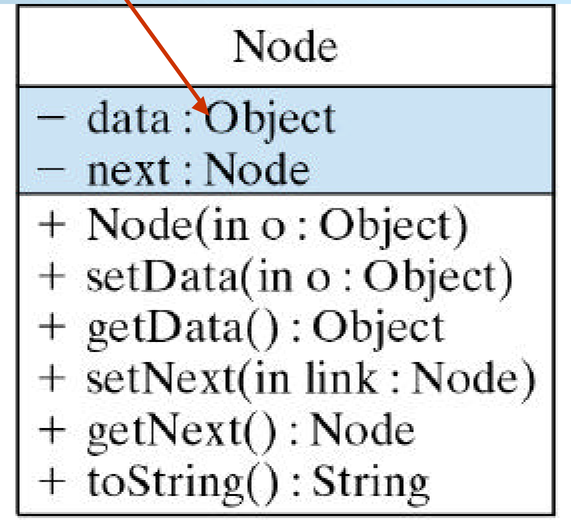
- Insertion and deletion in a sorted array can be time consuming—all the elements following the inserted or deleted element must be shifted appropriately.

# Single Linked List & Doubly Linked List

- Singly linked list
  - Each node contains one reference to the next node in the list (Example)
- Doubly linked list
  - Each node contains a reference to the next node in the list and a reference to the previous node in the list (Example)
  - `java.util`'s `LinkedList` class is a doubly linked list implementation

# The Generic List Class: Implementation

The data field is an Object reference, so it can refer to any object.



Node

− data : Object
− next : Node

+ Node(in o : Object)
+ setData(in o : Object)
+ getData() : Object
+ setNext(in link : Node)
+ getNext() : Node
+ toString() : String

**List**

**head: Node** // firstNode

**tail: Node** // lastNode

**Name: String**

**List ()**

**List(name: String)**

**insertAtFront(o: Object)**

**insertAtBack(o: Object)**

**removeFromFront()**

**removeFromBack()**

**isEmpty(): Boolean**

**size(): int**

# The Generic List Class: Implementation (Cont)

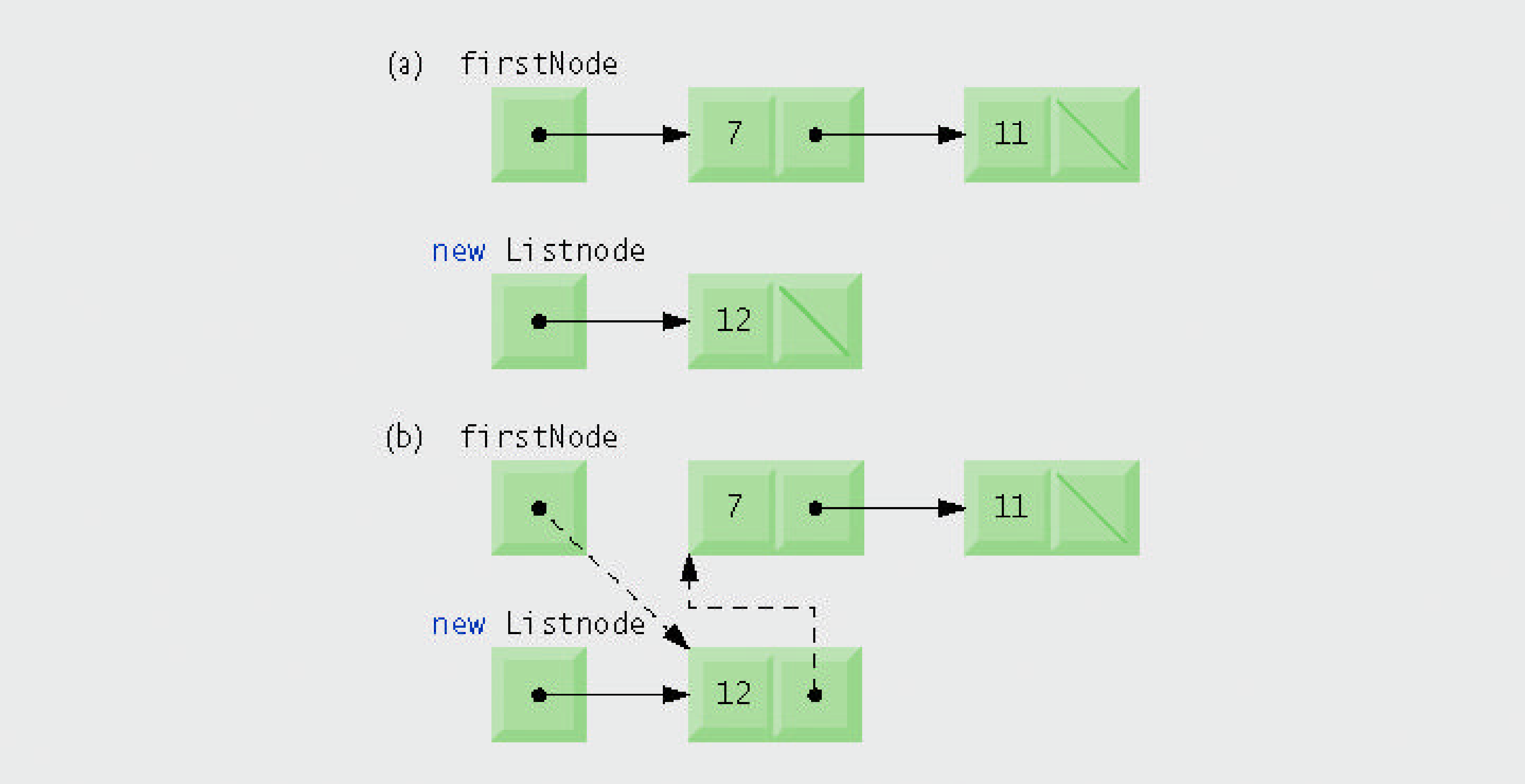
```
public class List {
    private Node head;
    private Node tail;
    public List() {
        head = null;
    }

    public boolean isEmpty() {
        return head == null;
    }

    public void print() {  }

    public void insertAtFront( Object newObj ) { }

    public void insertAtBack( Object newObj ) { }

    public Object removeFromFirst() { }

    public Object removeFromLast() { }
    ………
} // List
```

# Linked List: insertAtFront

- **Method insertAtFront's steps**
  - Call isEmpty to determine whether the list is empty
  - If the list is empty, assign firstNode and lastNode to the new ListNode that was initialized with insertItem
    - The ListNode constructor call sets data to refer to the insertItem passed as an argument and sets reference nextNode to null
  - If the list is not empty, set firstNode to a new ListNode object and initialize that object with insertItem and firstNode
    - The ListNode constructor call sets data to refer to the insertItem passed as an argument and sets reference nextNode to the ListNode passed as argument, which previously was the first node

# Linked List: insertAtFront (Cont)

Graphical representation of operation `insertAtFront`



# Linked List: insertAtFront (Cont)

**Code of `insertAtFront`**

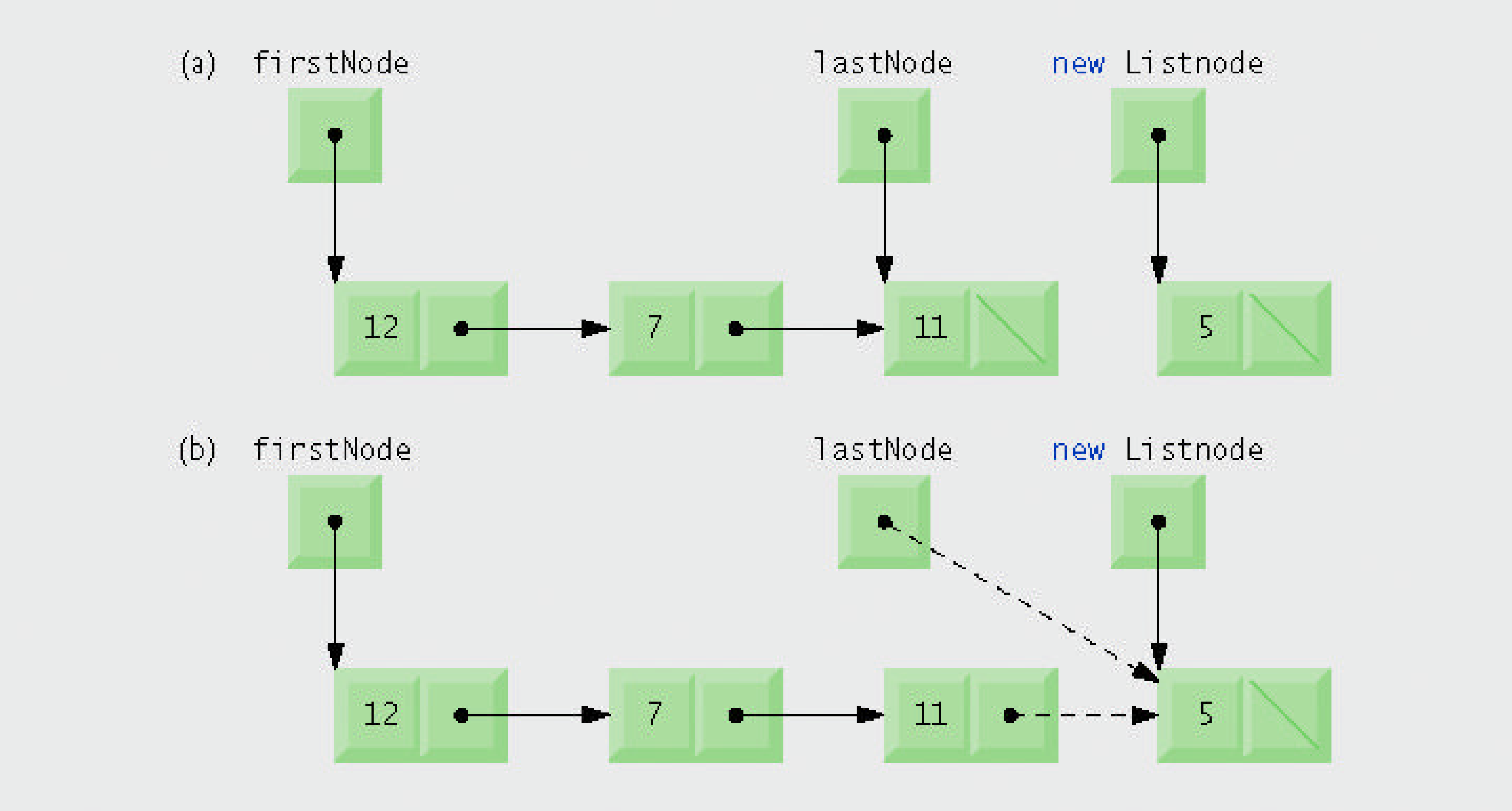
```
public void insertAtFront(Object obj) {

   Node newnode =  new Node(obj);

   newnode.setNext(head);

   if(isempty())
       head=tail= newnode;

   else

       head = newnode;

} // insertAtFront()
```

# Linked List: insertAtBack

- **Method insertAtBack's steps**
  - Call isEmpty to determine whether the list is empty
  - If the list is empty, assign firstNode and lastNode to the new ListNode that was initialized with insertItem
    - The ListNode constructor call sets data to refer to the insertItem passed as an argument and sets reference nextNode to null
  - If the list is not empty, assign to lastNode and lastNode.nextNode the reference to the new ListNode that was initialized with insertItem
    - The ListNode constructor sets data to refer to the insertItem passed as an argument and sets reference nextNode to null

# Linked List: insertAtBack (Cont)

Graphical representation of operation insertAtBack.



# Linked List: insertAtBack (Cont)

## Code of insertAtBack

```
public void insertAtBack(Object obj) {
    if (isEmpty())
        head = tail = new Node(obj);
    else {
        Node current = head;                // Start at head of list
        while (current.getNext() != null)   // Find the end of the list
            current = current.getNext();
        current.setNext(new Node(obj));    // Insert the newObj
    }
} // insertAtRear
```

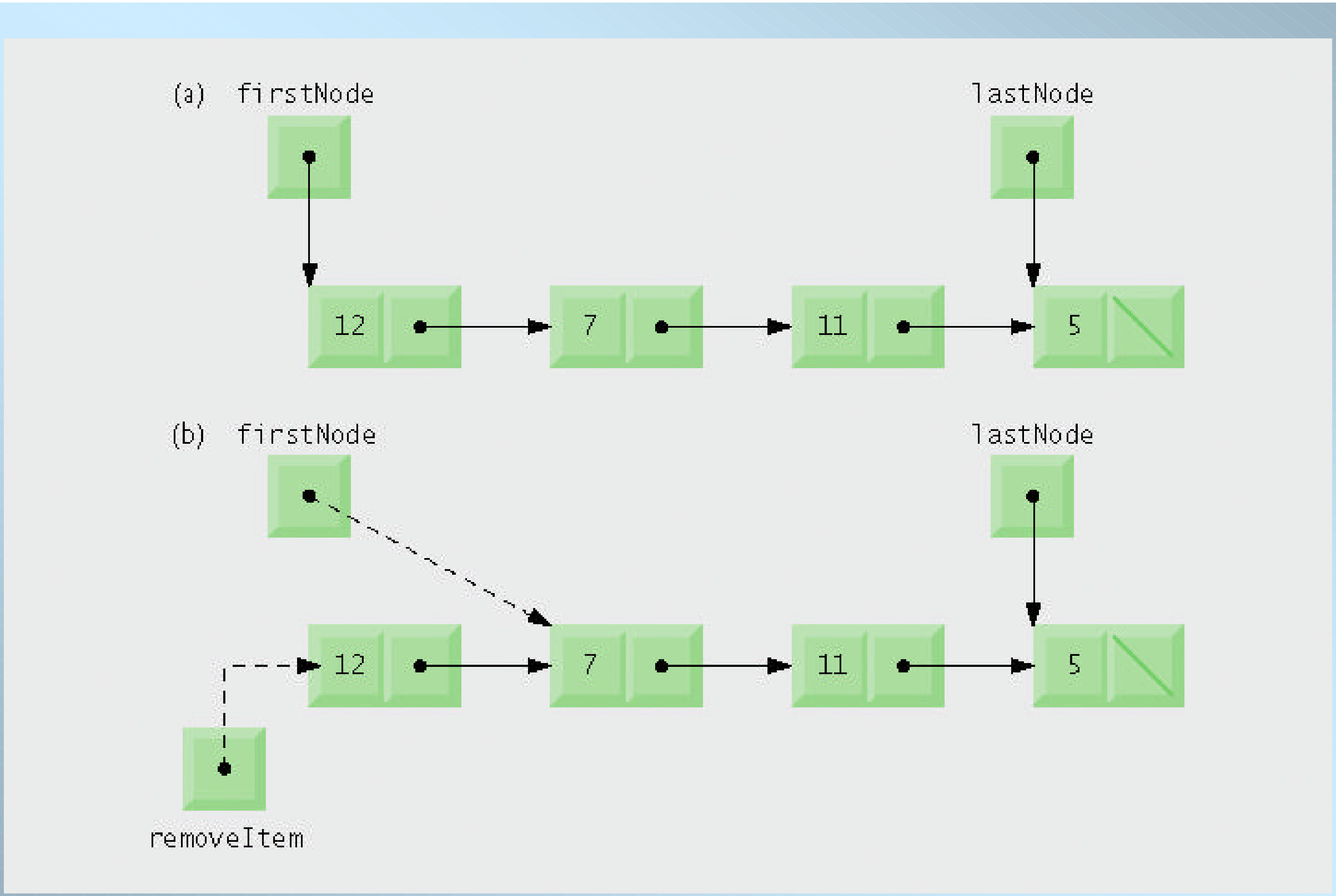
**Other solution using the tail**

```
public void insertAtBack(Object obj) {
      if(isempty())
          head = tail = new Node(obj);
      else{
          Node newnode =  new Node(obj);
          tail.setNext(newnode);
          tail=newnode;
      }
}
```

# Linked List: removeFromFront

- **Method removeFromFront's steps**
  - Throw an EmptyListException if the list is empty
  - Assign firstNode.data to reference removedItem
  - If firstNode and lastNode refer to the same object, set firstNode and lastNode to null
  - If the list has more than one node, assign the value of firstNode.nextNode to firstNode
  - Return the removedItem reference

# Linked List: removeFromFront



Graphical representation of operation removeFromFront.

# Linked List: removeFromFront

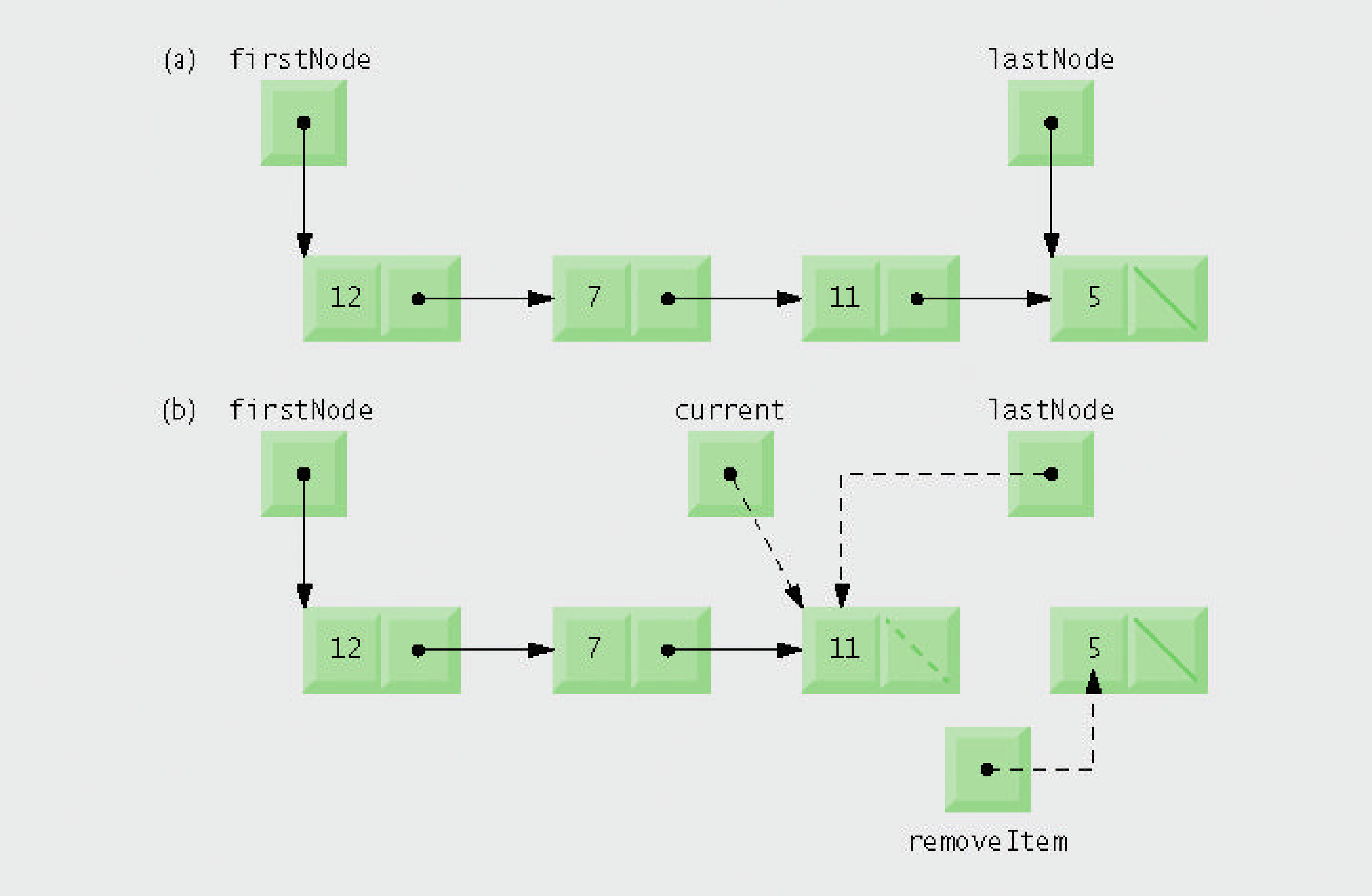
## Code of removeFromFront

```
public Object removeFromFrontt() {
    if (isEmpty())
        return null;
    Node first = head;
    if head == tail
          head = tail = null;
    head = head.getNext();
    return first.getData();
}
```

# Linked List: removeFromBack

- **Method removeFromBack's steps**
  - Throws an EmptyListException if the list is empty
  - Assign lastNode.data to removedItem
  - If the firstNode and lastNode refer to the same object, set firstNode and lastNode to null
  - If the list has more than one node, create the ListNode reference current and assign it firstNode
  - "Walk the list" with current until it references the node before the last node
    - The while loop assigns current.nextNode to current as long as current.nextNode is not lastNode

# Linked List: removeFromBack



Graphical representation of operation removeFromBack.

- –Assign `current` to `lastNode`
- –Set `current.nextNode` to `null`
- –Return the `removedItem` reference

# Linked List: removeFromBack

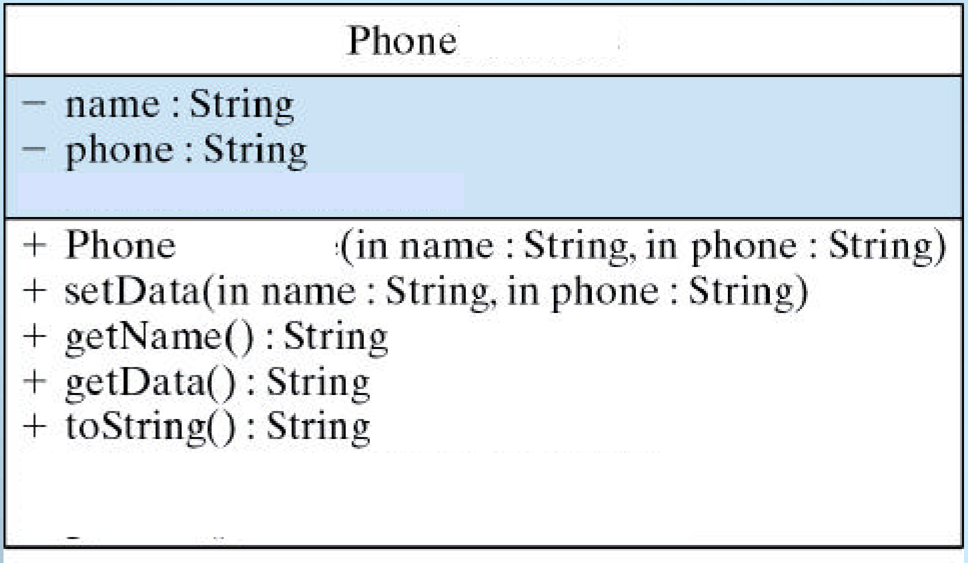
## Code of removeFromBack

```
public Object removeFromBack() {
    if (isEmpty())  // Empty list
        return null;

    Node current = head;
    if (current.getNext() == null) { // Singleton list
        head = tail = null;
        return current.getData();
    }

    Node previous = null;            // All other cases
    while (current.getNext() != null) {
        previous = current;
        current = current.getNext();
    }
    previous.setNext(null);
    tail = previous;
    return current.getData();
} // removeLast()
```

# Linked List: size

```
public int size()  {
    if(isempty())  return 0;
    Node current = head;;
    int c=1;
    while(current.getNext()!=null){
             current=current.getNext();
             c++;
    }
    return c;
}
```

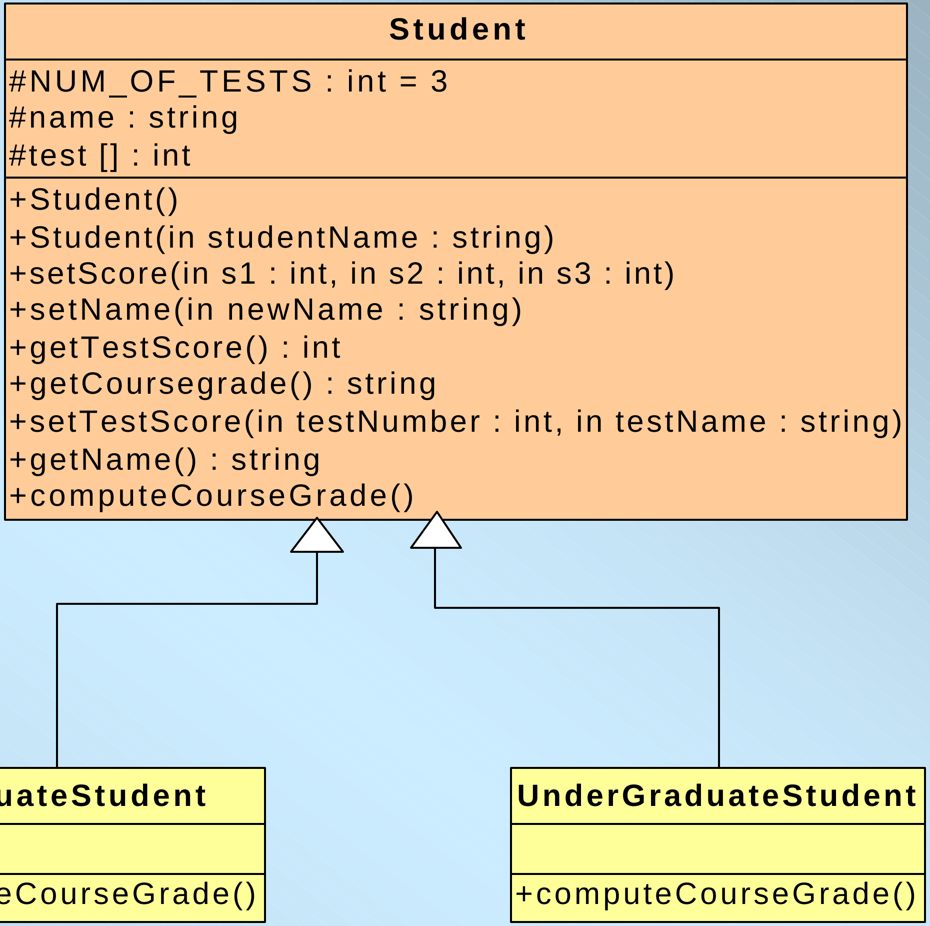
# Example: Create list and insert heterogeneous nodes



Phone

− name : String
− phone : String

+ Phone :(in name : String, in phone : String)
+ setData(in name : String, in phone : String)
+ getName() : String
+ getData() : String
+ toString() : String



**Student**

#NUM_OF_TESTS : int = 3
#name : string
#test [] : int

+Student()
+Student(in studentName : string)
+setScore(in s1 : int, in s2 : int, in s3 : int)
+setName(in newName : string)
+getTestScore() : int
+getCoursegrade() : string
+setTestScore(in testNumber : int, in testName : string)
+getName() : string
+computeCourseGrade()

**GraduateStudent**

+computeCourseGrade()

**UnderGraduateStudent**

+computeCourseGrade()

# Testing the List ADT

```
Public class Test {
  public static void main( String argv[] ) {
                // Create list and insert heterogeneous nodes
    List list = new List();

    Student s1 =new Student("Saad");

    s1.setScore(10,20,15);

    s1.computeCourseGrade();


    list.insertAtFront(s1);
    list.insertAtFront(new Phone(“Ali M", "997-0020"));
    list.insertAtFront(new Integer(8647));
    list.insertAtFront(new String("Hello World"));


    System.out.println("Generic List"); // Print the list
    list.print();
                 // Remove objects and print resulting list
    Object o;
    o = list.removeFromBack();
    System.out.println(" Removed " + o.toString());
    System.out.println("Generic List:");
    list.print();
    o = list.removeFromFirst();
    System.out.println(" Removed " +o.toString());
    System.out.println("Generic List:");
    list.print();
   } // main()
}
```

# Example: Node with data Student

```
public class Node
{
   public Student data;
   public Node nextNode;

   public Node(Student object )
   {
      this( object, null );
   }

   public Node(Student object, Node node)
   {
      data = object;
      nextNode = node;
   }

   public Student getData()
   {
      return data;
   }

   public  Node getNext()
   {
      return nextNode;
   }
} // end class Node
```

# Class Student

```
class Student
{
  private final static int NUM_OF_TESTS = 3;
  private   String           name;
  private   int[]              test;
  private   String           courseGrade;
  public Student( )
  { this ("No Name"); }
  public Student(String studentName)
  {
    name = studentName;
    test = new int[NUM_OF_TESTS];
    courseGrade = "****";
  }
  public void setScore(int s1, int s2, int s3)
   {
     test[0] = s1; test[1] = s2; test[2] = s3;
    }
  public String getCourseGrade( )
  {
      return courseGrade;
  }
  public String getName( ) { return name; }
```

```
public void computeCourseGrade()
   {if (getTotal() >= 50)
     courseGrade = "Pass";
    else { courseGrade = "NoPass";
    }
public int getTestScore(int testNumber) {
  return test[testNumber-1];  }
public void setName(String newName) {
    name = newName;  }
public void setTestScore(int tN, int tS)
   {         test[tN-1]=tS;  }
public int getTotal()
{ int total = 0;
  for (int i = 0; i < NUM_OF_TESTS; i++) {
    total += test[i]; }
  return total;
}
public void display()
{System.out.print("The student "+ name +" has
                 "+getTotal()+ " marks");
 System.out.println("  and  Course grade = "+
                          courseGrade);
}
}
```

# Class List

```
// class List definition
public class List
{  private Node firstNode;
   private Node lastNode;
   private String name;
   public List()
   { this( "list" );
   }
   public List(String listName )
   {  name = listName;
      firstNode = lastNode = null;
   }
   public void insertAtFront(Student stud )
   {if ( isEmpty() )
      firstNode = lastNode = new Node(stud);
    else
     firstNode = new Node(stud, firstNode );
   }
public void insertAtBack(Student stud)
 {if ( isEmpty() )
    firstNode = lastNode = new Node(stud);
  else
   lastNode=lastNode.nextNode = new
Node(stud);
 }
```

```
public Student removeFromFront()
   { Student st = firstNode.data;
     if ( firstNode == lastNode )
         firstNode = lastNode = null;
      else
         firstNode = firstNode.nextNode;
      return st;
   }
public Student removeFromBack()
   {Student st = lastNode.data;
    if ( firstNode == lastNode )
         firstNode = lastNode = null;
    else
      { Node current = firstNode;
        while ( current.nextNode != lastNode )
            current = current.nextNode;
        lastNode = current;
         current.nextNode = null;
      }
    return st;
   }
```

# Class List

```
public boolean isEmpty()
{ return firstNode == null; } // End isEmpty
public void print()
{if ( isEmpty() )
 {System.out.println("The list" + name +"
                     is empty");
  return; }
  System.out.println( "\n" );
  System.out.println( "The list : "+ name+
                     " contains : " );
  Node current = firstNode;
  while ( current != null )
  {current.data.display();
   current = current.nextNode;
   }
} // End method print

public int maximumMarks()
{if ( isEmpty() )
 {System.out.println("The list" + name +" is
                     empty");
  return -1;}
```

```
 int max=firstNode.data.getTotal();
  Node current = firstNode.nextNode;
  while ( current != null )
  {if (max < current.data.getTotal())
    max =current.data.getTotal();
   current = current.nextNode;
  }
  return max;
} // End method maximumMarks
public double averageMarks()
{if ( isEmpty() )
 {System.out.println("The list" + name +"
            is empty");
   return 0.0;}
 int sum=0, counter=0;
 Node current = firstNode;
 while ( current != null )
{sum+=current.data.getTotal();
 counter++;
 current = current.nextNode;
}
 return 1.0*sum/counter;
} // End method averageMarks
```

# Class List

```
//=== this method computes the number of passed or NotPassed student
   public int numberOfPassedOrNotPassedStundent(String ss)
   {
     if ( isEmpty() )
     {
       System.out.println("The list" + name +" is empty");
       return -1;
      }
    int nb=0;
    Node current = firstNode;
    while ( current != null )
    {
     if(current.data.getCourseGrade().equals(Pass))
       nb++;
     current = current.nextNode;
    }
    return nb;
 }
} // end class List
```

# Testing the List ADT

```
public class ListStudentTest
{public static void main(String args[])
 { List ob = new List("csc");
   Student s1 =new Student("Saad");
   s1.setScore(10,20,15);
   s1.computeCourseGrade();

   Student s2 =new Student("Ali");
   s2.setScore(10,50,40);
   s2.computeCourseGrade();

   Student s3 =new Student("Nabil");
   s3.setScore(30,10,15);
   s3.computeCourseGrade();

   Student s4 =new Student("Sami");
   s4.setScore(32,14,44);
   s4.computeCourseGrade();

   ob.insertAtFront(s1);
   ob.insertAtFront(s2);
   ob.insertAtFront(s3);
   ob.insertAtFront(s4);
   ob.print();
```

```
System.out.println("number of passed Students is :
  "+ob.numberOfPassedOrNotPassedStundent("Pass"));
System.out.println("number of  not passed Students
is:"+ob.numberOfPassedOrNotPassedStundent("NoPass"));
System.out.println("The max is:"+ ob.maximumMarks());
System.out.println("The avrg : "+ ob.averageMarks());
ob.removeFromFront();
System.out.println("After remov- the first node :");
ob.print();
System.out.println("number of passed Students is :
     "+ob.numberOfPassedOrNotPassedStundent("Pass"));
System.out.println("number of  not passed Students is
  :"+ob.numberOfPassedOrNotPassedStundent("NoPass"));
System.out.println("The max is:"+ ob.maximumMarks());
System.out.println("The avrg:"+ ob.averageMarks());
 }
}
```

# Testing the List ADT

```
/*   output
   The list : csc contains :
    The student Sami  has 90 marks  and  Course grade = Pass
    The student Nabil  has 55 marks  and  Course grade = Pass
    The student Ali  has 100 marks  and  Course grade = Pass
    The student Saad  has 45 marks  and  Course grade = NoPass
    ====number of passed Students is : 3
    ====number of  not passed Students is : 1
    The maximum is : 100
    The average : 72.5
    After removing the first node :



    The list : csc contains :
    The student Nabil  has 55 marks  and  Course grade = Pass
    The student Ali  has 100 marks  and  Course grade = Pass
    The student Saad  has 45 marks  and  Course grade = NoPass
    ====number of passed Students is : 2
    ====number of  not passed Students is : 1
    The maximum is : 100
    The average : 66.66666666666667
 */
```

# The Generic List Class: Implementation with the element type that the Node will manipulate

## The Node Class: Implementation

```
public class Node<T>
{
   T data;
   Node nextNode;

   public Node( T object )
   {
      this( object, null );
   }

   public Node(T object, Node node)
   {
      data = object;
      nextNode = node;
   }
```

```
    public T getData()
   {
      return data;
   }

   public  Node getNext()
   {
      return nextNode;
   }
 } // end class Node
```

# The Generic List Class: Implementation with the element type that the Node will manipulate

```
public class List<V>
{ private Node<V> firstNode;
  private Node<V> lastNode;
  private String name;
public List()
  {  this( "list" );
  }
public List(String listName )
  { name = listName;
     firstNode = lastNode = null;
  }
public void insertAtFront(V insertItem )
{ if ( isEmpty() )
     firstNode = lastNode = new Node<V>( insertItem );
  else
     firstNode = new Node<V>( insertItem, firstNode );
  }
```

```
public void insertAtBack( V insertItem )
{  if ( isEmpty() )
      firstNode = lastNode = new Node<V>(
insertItem );
   else
     lastNode = lastNode.nextNode = new Node<V>(
insertItem );
   }
public V removeFromFront()
   { V removedItem = firstNode.data;
      if ( firstNode == lastNode )
         firstNode = lastNode = null;
      else
         firstNode = firstNode.nextNode;
     return removedItem;
   }
  public V getFromFront()
   {  return firstNode.data; }
```

# The Generic List Class: Implementation with the element type that the Node will manipulate

```
public V removeFromBack()
 { V removedItem = lastNode.data;
    if ( firstNode == lastNode )
        firstNode = lastNode = null;
    else {
        Node<V> current = firstNode;
       while ( current.nextNode != lastNode )
          current = current.nextNode;
        lastNode = current;
       current.nextNode = null;
      }
     return removedItem;
   }
  public boolean isEmpty()
  {   return firstNode == null;    }
```

```
 public void print()
  {
    if ( isEmpty() )
    {
      System.out.printf( "Empty %s\n", name );
      return;
    }

    System.out.printf( "The %s is: ", name );
    Node current = firstNode;

    while ( current != null )
    {
      System.out.printf( "%s ", current.data );
      current = current.nextNode;
    }

    System.out.println( "\n" );
  }

} // end class List
```