Linked List

Juan Zhai

juan.zhai@rutgers.edu

Java is Pass by Value

```
public static Node insert(Node front,  int data){
    //front stores the address 2000 when main calls insert()
    Node node = new Node(data, front);//suppose node stores the address
2800
    front = node; /*front stores 2800 because this assignment copies the
                                  address 2800 stored in node to front*/
    return front; //return front which stores the address 2800
}
public static void set(Node front, int data){
    front.data = data;  /*front stores the address 2800, this assignment
                              updates the info field in the address 2800*/
}
public static void main(String[] args){
     Node head = new Node(3, null); //suppose head stores the address 2000
     head = insert(head,6); /*before assignment, head stores the address
                      2000, after assignment, head stores the address 2800
                      so the list now has 6 and 3*/
    set(head, 5); /*head stores the address 2800, the info stored in 2800
                          is changed to 5, so the list now has 5 and 3*/
  }
```

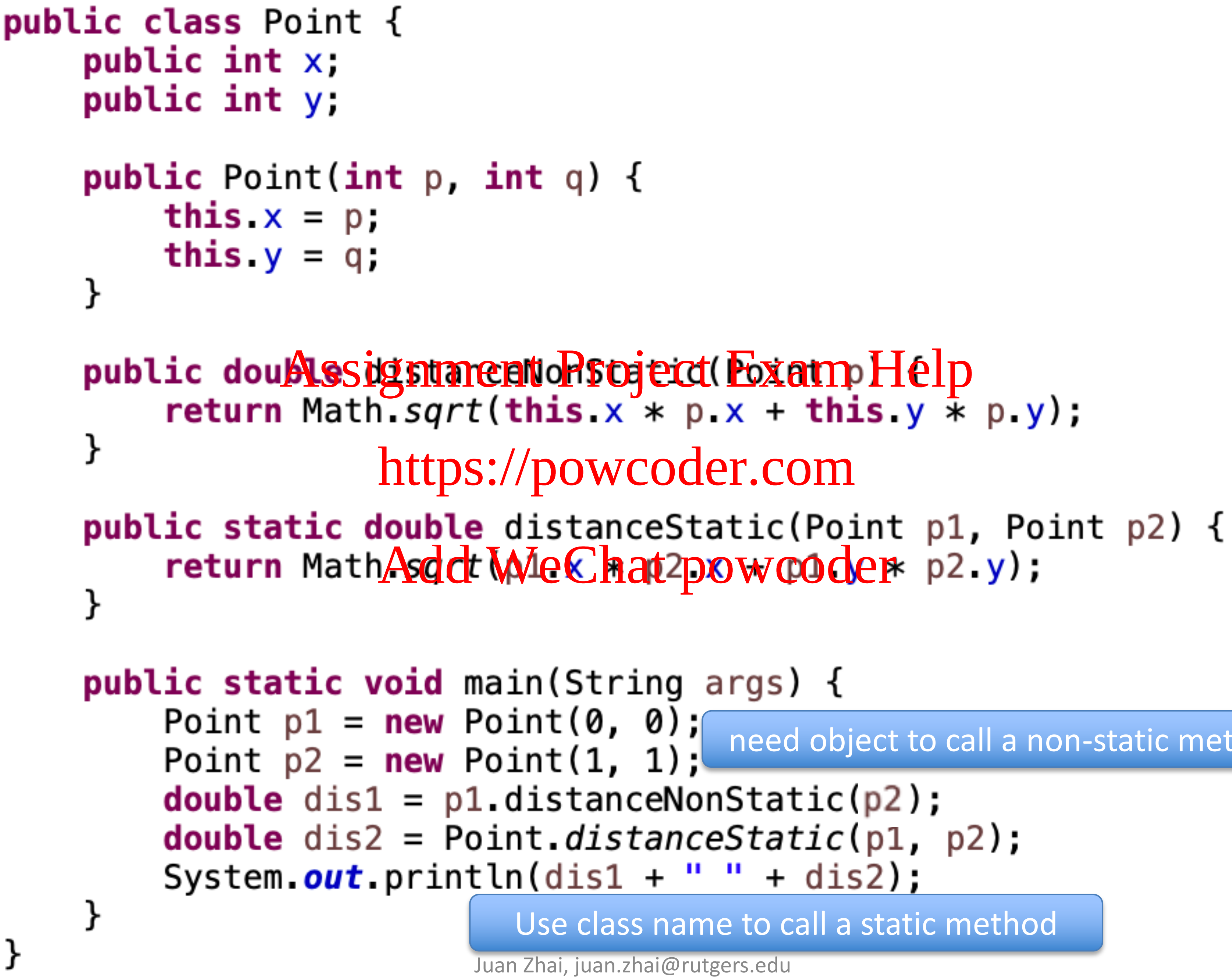
Object-oriented programming

- When we insert/delete nodes from a linked list, we need to return *front* since the address stored in *front* may be updated. By returning *front*, the caller will know where to get the latest linked list.
- What if we want the method to return a boolean value to tell us whether the operation is successfully performed or not?
- We are manipulating the linked list that is pointed to by *front*, is it possible that all the operations share *front* to avoid sending *front* back and forth?

Object-oriented programming

- Object-oriented programming (OOP) is based on the concept of objects.
- Each object contains data (in the form of instance variables, known as *attributes or properties*), and operations (in the form of methods).
- A feature of objects is that an object's methods can access and modify the data fields of the object with which they are associated.
- Encapsulate a linked list object with *front* as its data and insertion/deletion operations as its methods.

  → unnecessary to pass *front* as a parameter any more



```java
public class Point {
    public int x;
    public int y;

    public Point(int p, int q) {
        this.x = p;
        this.y = q;
    }

    public double distanceNonStatic(Point p) {
        return Math.sqrt(this.x * p.x + this.y * p.y);
    }

    public static double distanceStatic(Point p1, Point p2) {
        return Math.sqrt(p1.x * p2.x + p1.y * p2.y);
    }

    public static void main(String args) {
        Point p1 = new Point(0, 0);
        Point p2 = new Point(1, 1);
        double dis1 = p1.distanceNonStatic(p2);
        double dis2 = Point.distanceStatic(p1, p2);
        System.out.println(dis1 + " " + dis2);
    }
}
```

need object to call a non-static method

Use class name to call a static method

```
public Class Node{
    public String data;
    public Node link;
    public Node(int data,
                Node next){
        this.data = data;
        this.link = link;
    }
}

public static void main(String[]
                        args){
   LinkedList list = new
                LinkedList();
   list.addFront(“Apple”);
   list.addFront(“Banana”);
   list.addFront(“Orange”);
   list.print();
}
```

```
public Class LinkedList{
    Node front;
    public LinkedList(){
        front = null;
    }
    public void print(){
        Node ptr = front;
        while (ptr != null) {
    System.out.println(ptr.data);
           current = ptr.link;
    }
    public void addFront(int data){
        Node node = new Node(data,
                                 null);
        node.link = front;
        head = front;
    }
  }
```

Encapsulate data

non-static methods

```
public static void main(String[] args){
   Node list = new Node(); //list points to the first node
   list = addFront(list, “Apple”);
   list = addFront(list, “Banana”);
}
```

static methods: belong to Class, all objects have the same view, so just use class name to invoke static methods, no need to use objects: className.methodName(), className can be omitted if the method is called within the class.

```
public static void main(String[] args){
   LinkedList list = new LinkedList(); //list is an
object with the instance variable head which points to
the first node (encapsulate the data “head”)
   list.addFront(“Apple”);
   list.insertFront(“Banana”);
}
```

non-static methods: different objects have different views (different values for different instance variables), need to invoke a method on an object: obj.methodName()

head is an instance variable

1. *Non-static methods have access to head, so no need to pass head as a parameter*
2. *The modifications in the non-static methods are made to head and each object has access to its instance variable (head) , so no need to return head*

Generics

- Define IntNode for integers, define StringNode for strings, is there a general way applicable for all data types?
- Define a class to accept objects of some generic type
- By convention, we use ***T*** and it stands for “template”
- A generic class is said to be a template that can be concretized to contain objects of any particular type when the generic class itself is instantiated

Generics

```
//T: generic type parameter
public Class Node<T>{
   public T info;
   public Node link;
   …
}
```

```
public class LinkedList<T>(){
   Node<T> head;
   …
   public void addFront(T o) {…}
   public T get(int index) {…}
}
```

```
public class BookApp(){
   …
   LinkedList<Book> list = new LinkedList<Book>();
                        //instantiate the generic class
   list.addFront(New Book(“Name”, “author”))
   Book book = list.get(6);
                     //the retrieved item is a Book object
   System.out.println(book.author);
   …
}
```

Exceptions

```
public class LinkedList<T>(){
    Node<T> front;
    …
    public T getFront() throws NoSuchElementException{
        if (front == null) {
            // throw new NoSuchElementException();
            throw new NoSuchElementException(“empty …");
        }
        return front.data;
    }
}
```

throws: declare an exception in method signature, meaning likely to throw

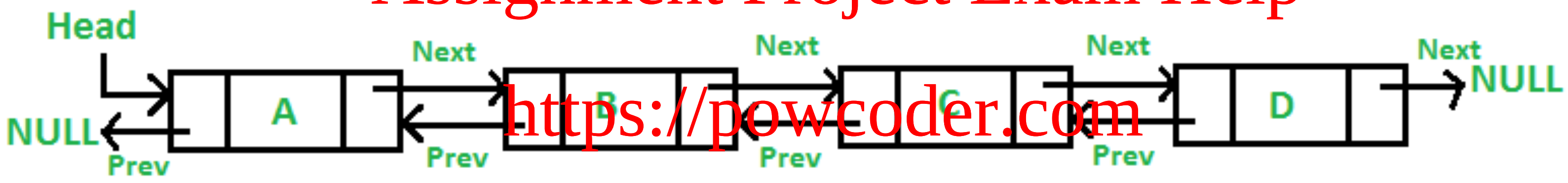
throw: throw an exception in method body explicitly. After executing throw statement, the control is given back to its caller (just like return), and the following statements will not be executed.

- Separate normal flow from error situations with exceptions
- When an error occurs, create an exception and throw it to launch an exceptional control flow

- Refer to Sakai code for Linked List with generics and exceptions

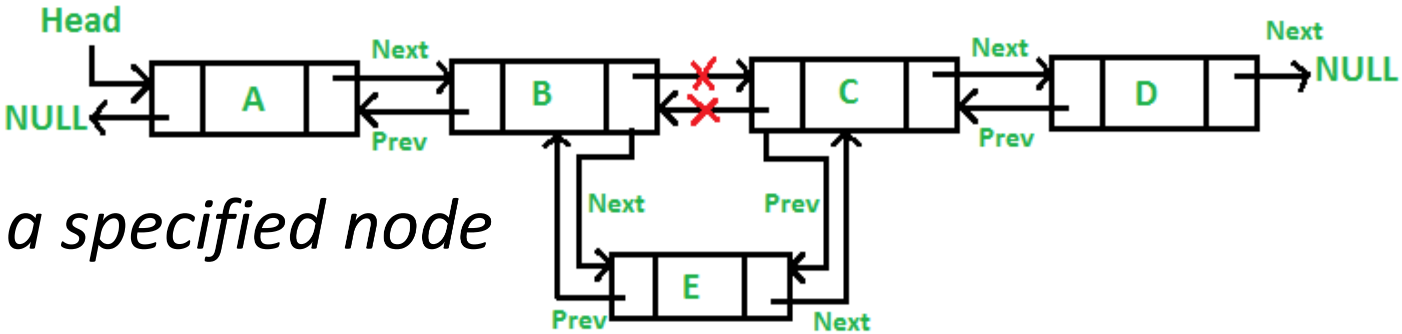
Doubly linked list

- A doubly linked list is a linked list in which each node has previous link that points to the previous node in the linked list



The previous link of the first node is null

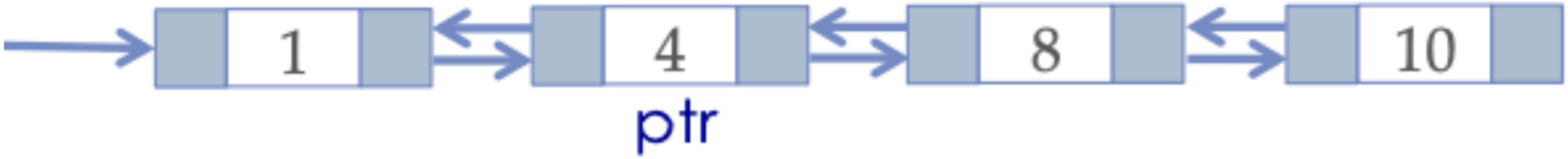
```
public Class Node{
    String data;
    Node next;
    Node prev;
}
```



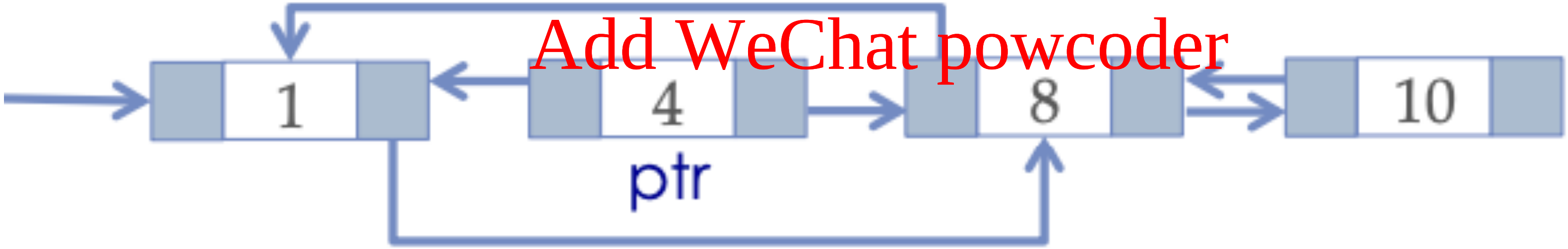
Insert after a specified node

```
public static void insertAfter(Node prevNode, String data) {
   //1. check whether the given previous node is null
     if(prevNode == null)
           throw new NullPointerException(“Previous node is NULL.”);
   //2. create a new node with the given data
     Node newNode = new Node(data);
    //3. make the new node’s next point to the next node of prevNod
     newNode.next = prevNode.next;
   //4. make the previous node’s next point to the new node
     prevNode.next = newNode;
    //5. make the new node’s prev link point to the previous node
     newNode.prev = prevNode;
     //6. make the new node’s next node’s prev point to the new node
     if(newNode.next != null)
          newNode.next.prev = newNode;
}
```

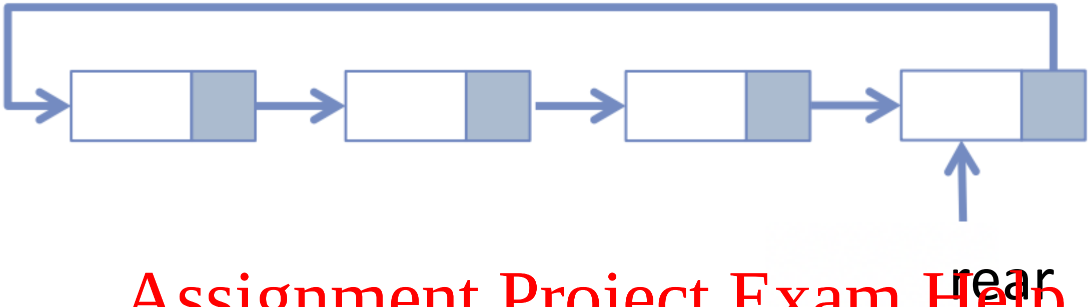
Delete a specified node in the middle



```
ptr.next.prev = ptr.prev;
ptr.prev.next = ptr.next;
```



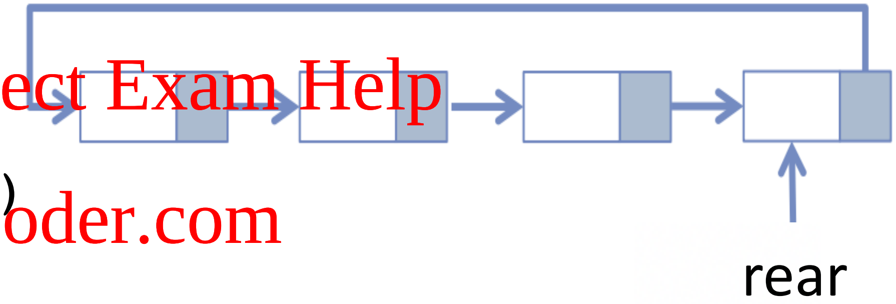
Circular Linked List



- A circular linked list is a linked list in which the last node refers back to the first node
- All the nodes are connected to form a circle
- There is no NULL at the end
- By keeping a pointer to the last entry, we have access to the first and last entry both in constant time.
  - Last: rear
  - First: rear.link

Search target

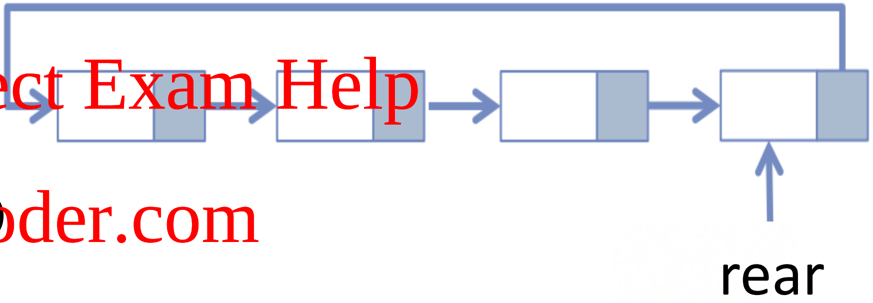
```
public boolean search(String target) {
   if(rear == null)
      return false;
   Node ptr = rear.next;
   while(ptr != rear){
      if(ptr.data == target)
         return true;
       ptr = ptr.next;
   }
   return rear.data == target;
}
```



- Search from the first node.
- If there is only one node in the list, then rear.next is rear. No iterations are performed. Comparison is done at last line.

Search target

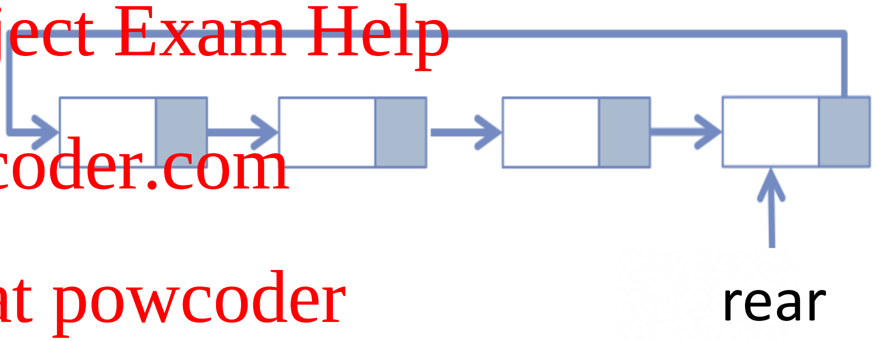
```
public boolean search(String target) {
    if(rear == null)
        return false;
    Node ptr = rear;
    do {
        if(ptr.data == target)
            return true;
         ptr = ptr.next;
    } while(ptr != rear)
    return false;
}
```



- Compare the last node and then search from the first node.
- If there is only one node in the list, then rear.next is rear. One iteration is done to perform the comparison.

Add Front

```
public void AddFront(String data) {
   Node node = new Node(data, null);
   if(rear == null){     // no node, an empty list
      rear = node;
      rear.next = rear;
   }
   else{
      node.next = rear.next
      rear.next = node;
   }
}
```



Delete Front

```
public int deleteFront() throws NoSuchElementException{
   if(rear == null)      // no node, an empty list
      throw new NoSuchElementException();
   int tmp = rear.next.data;
   if(rear == rear.next)    //only one node
      rear = null;
   else
      rear.next = rear.next.next; //at least two nodes
   return tmp;
}
```