Handout 14

Dynamic Data Structures.
Linked Lists (Part 2).

Last time – looked at representation of each list node. Recall, for pizza shop example we have defined a class PizzaOrderNode:

```java
public class PizzaOrderNode{
    ...
    // instance variables
    private String name;    // name of person placing the order
    private String address; // address of person placing the order
    private String pizzaType; //
    private int quantity;    //

    private PizzaOrderNode next; // a link to the next node in the list
    ...
}
```

To represent a linked list of nodes – define a class that contains the reference to the head (first) node in the list. Then, the rest of the nodes can be accessed following the links from the head node:

```java
public class OrdersList
{
    ...
    // instance variable
    PizzaOrderNode head; // the head of linked list.

    // Constructor: initializes the head to null
    public OrdersList ()
    {
        head = null;
    }
    ...
}
```

To insert, delete, print, count, etc. the nodes of the linked lists the OrdersList class has to define appropriate methods.

Insert and delete methods have to consider a special case when the list is empty, i.e. the value of the instance variable `head` is `null`. 
1. Adding a node to the list:
   - nodes can be added to the front of the list, to the end or anywhere else depending on problem requirements.

Adding to front:  `private void AddNodeToFront (PizzaOrderNode newNode)`

   a) adding to the `empty list`, i.e. when `head` equals `null`:

   ![Diagram of adding to empty list](image)

   b) adding to the front of a nonempty list:

   ![Diagram of adding to nonempty list](image)
Here’s how it’s written in Java code:

```java
/* from OrderList class:
 * private void AddNodeToFront (PizzaOrderNode newNode)
 * inserts the newNode as the first node into the linked list */

private void AddNodeToFront (PizzaOrderNode newNode) {
    // the special case : adding to an empty list
    if (head == null)
        head = newNode;
    else // the general case
    {
        newNode.setNext (head);
        head = newNode;
    }
}
```

Adding between two nodes: need references to both predecessor and successor nodes:

```
head
```

```
newNode
```

transform into:

```
head
```

```
newNode
```
2. Traversing the list:

```java
/* public void print ()
 * Traverses the list printing contents of each node
 * /
public void print ()
{
    PizzaOrderNode orderNode; // pointer to an object in the list
    System.out.println("Printing the list of Pizza Orders");
    orderNode = head; // start at the first node
    while (orderNode != null) {
        orderNode.print();
        orderNode = orderNode.getNext(); // set orderNode to point at
        // the next node of the list
    }
}
```

3. Deleting a node:

a) deleting the front node

```
head

```

transform into:

```
head

```

```
b) deleting an **internal** node: need **references to both predecessor and the deleted node**

![Diagram of deleting an internal node in a list]

Transform into

![Diagram of transformed list]

- 5 -