6.1 Singly Linked List

A linked list is a data structure consisting of a group of nodes which together represent a sequence. Each node is composed of a data and a reference (in other words, a link) to the next node in the sequence. A Node class usually look like this:

```java
class Node<E> {
    public E data;
    public Node<E> next;
    Node(E item) {
        data = item;
    }
}
```

Usually Node class is nested inside the LinkedList class, and members of Node are private.

### 6.1.1 Create a simple linked list

Now, let us create a simple linked list.

```java
Node<String> n1 = new Node("Alice");
Node<String> n2 = new Node("Bob");
Node<String> n3 = new Node("Cathy");
Node<String> n4 = new Node("David");
n1.next = n2;
n2.next = n3;
n3.next = n4;
```

This linked list represents this:

![Linked List Diagram]

### 6.1.2 Display the Linked List

We can display all the linked list:

```java
Node<String> current = first;
while(current != null) {
    System.out.println(current.data);
    current = current.next;
}
```
Here is the recursive version of the same code

```java
public void print(){
    print(first);
    System.out.println("\n");
}

private void print(Node r){
    if(r == null) return;
    System.out.print(r.data",");
    print(r.next);
}
```

### 6.1.3 Insert a node

Now, let us insert a node between “Bob” and “Cathy”.

```java
Node<String> n5 = new Node("Ethan");
n5.next = n2.next;
n2.next = n5;
//use "first" to reference the first node of the list.
Node<String> first = n1;
```

This linked list represents this:

![Linked List Diagram]

### 6.1.4 Delete a node

#### 6.1.4.1 Delete the first node

To delete the first node, we can simply move “first” to next node.

```java
first = first.next;
```

#### 6.1.4.2 Delete other nodes

In order to delete a Node, we have to know the parent of the node. Assume “parent” references the node “Ethan”, to delete the node “Cathy” reference by “current”, we can do this:

```java
parent.next = current.next;
```

No, we have:

![Linked List Diagram]

6.2 Linked List Class

```java
/**
 * The Bag class represents a collection of generic items.
 * It supports insertion and iterating over the items in arbitrary order.
 */
import java.util.ArrayList;
import java.util.Iterator;

public class LinkedBag<E extends Comparable<E>> implements Iterable<E> {
    protected int N;  // number of items in the bag
    private Node<E> first;  // beginning of bag

    // helper linked list class
    private class Node<E> {
        private E data;
        private Node<E> next;
        Node(E item) {
            data = item;
        }
    }

    /**
    * Initializes an empty bag.
    */
    public LinkedBag() {
        first = null;
        N = 0;
    }

    /**
    * Returns an iterator that iterates through the items in the bag
    * @return an iterator that iterates through the items in the bag
    */
    public Iterator<E> iterator() {
        return new BagIterator(first);
    }

    /**
    * The iterator implementation
    */
    private class BagIterator implements Iterator<E> {
        private Node<E> current = null;
        public BagIterator(Node<E> first) {
            current = first;
        }
        public boolean hasNext() {
            return current != null;
        }
        public void remove() {
            System.out.println("to be implemented.");
        }
        public E next() {
            if (!hasNext()) {
                return null;
            }
            E item = current.data;
            current = current.next;
            System.out.println("work");
            return item;
        }
    }

    /**
    * Adds the item to this bag.
    * @param item the item to add to this bag
    */
    public void insert(E item) {
        Node<E> oldfirst = first;
        first = new Node<E>(item);
        first.next = oldfirst;
        N++;
    }
}
```
public E get(int index)
{
    Node<E> current = first;
    int i = 0;
    while (current != null && i < index){
        current = current.next;
        i++;
    }
    if(current != null){
        return current.data;
    }else{
        return null;
    }
}

public boolean remove(E item){
if(item == null) return false;
if(isEmpty()) throw new NoSuchElementException();
Node<E> current = first;
Node<E> parent = first;
while(current != null){
    if(current.data.equals(item)){
        parent.next = current.next;
        return true;
    }
    parent = current;
    current = current.next;
}
return false;
}

/**
 * Deletes an item recursively
 * @param item is the item to be deleted
 * @return true if item is deleted. false otherwise
 */
public void remove_rec(E item){
    first = remove_rec(first, item);
}

/**
 * Deletes an item recursively
 * @param item is the item to be deleted
 * @param r is the starting node
 * @return true if item is deleted. false otherwise
 */
private Node<E> remove_rec(Node<E> r, E item){
    if(r == null) return null;
    if(r.data.equals(item)){
        return r.next;
    }
    r.next = remove_rec(r.next, item);
    return r;
}

/**
 * Is this bag empty?
 * @return true if this bag is empty; false otherwise
 */
public boolean isEmpty() {
    return first == null;
}
/**
 * Returns the number of items in this bag.
 * @return the number of items in this bag
 */
public int size() {
    return N;
}

/**
 * if the bag contains a given item?
 * @return true if bag contains the item. false otherwise
 */
public boolean contains(E item)
{
    Node<E> current = first;
    while (current != null)
    {
        if (current.data.equals(item)) return true;
        current = current.next;
    }
    return false;
}

6.2.1 Test the Linked List

/**
 * test Linked List Bag
 */
public class LinkedBagUnitTest {
    public static void main(String[] args)
    {
        LinkedBag<Integer> bag = new LinkedBag();
        for (int i=1; i <= 3; i++)
        {
            bag.insert(i);
        }
        System.out.println("Size="+bag.size());
        if(bag.contains(3))
        {
            System.out.println("Bag contains 3");
        }else
        {
            System.out.println("Not Found");
        }
        //print all items using iterator
        for (Integer i:bag)
        {
            System.out.print(i + ",");
        }
        //print all items using get method, which is not efficient.
        System.out.println("\nAll\nItems");
        for (int i = 0; i < bag.size(); i++)
        {
            System.out.print(bag.get(i)+"","");
        }
    }
}

6.3 Code example

6.3.1 Merge two sorted linked list into one

We have two sorted linked lists list1 and list2.
We want to generate the list:

Here is the code that takes two lists as input, and merges them into one list. This function takes $O(n_1 + n_2)$ time to merge two lists of size $n_1$ and $n_2$.

```java
public Node merge(Node l1, Node l2){
    //if one list is empty, return the other list
    if(l1 == null){
        return l2;
    }
    if(l2 == null){
        return l1;
    }
    //if both lists are not empty
    Node c1 = l1;
    Node c2 = l2;
    Node m = null;
    //pick the larger node from l1 and l2.
    if(c1.data > c2.data){
        m = c1;
        c1 = c1.next;
    }else{
        m = c2;
        c2 = c2.next;
    }
    /**
     * walk through l1 and l2, every time pick the larger node.
     * comparison only occurs at the head of two lists.
     */
    Node c3 = m;
    while(c1 != null && c2 != null){
        if(c1.data > c2.data){
            c3.next = c1;
            c1 = c1.next;
            c3 = c3.next;
        }else{
            c3.next = c2;
            c2 = c2.next;
            c3 = c3.next;
        }
    }
    if(c1 != null){
        }
The recursive version

```java
public Node merge_rec(Node l1, Node l2)
{
    if(l1 == null) return l2;
    if(l2 == null) return l1;
    if(l1.data > l2.data){
        l1.next = merge2(l1.next, l2);
        return l1;
    } else{
        l2.next = merge2(l1, l2.next);
        return l2;
    }
}
```