Linked-list: Succession of links from first link to the last link that points to reference “null”

The class Link contains some data and a reference to the next link

The class Linked List contains the reference to the first link and all major methods that operate on the list (ex: add, remove, display, search, etc.)

Advantages:
- Number of links can be expanded dynamically (memory efficient)
- Insertion/removal does not require moving items

Disadvantages:
- Conceptually less intuitive than arrays
- The objects/items of the link can be located anywhere in memory (loss of data locality)
class Link {
    public String name; // data
    public int age; // data
    public Link next; // reference to next link

    // Constructor
    public Link(String name, int age) {
        this.name = name;
        this.age = age;
        next = null; // optional
    }
}

Link 1stlink = new Link(name1, age1);

Link 2ndlink = new Link(name2, age2);

1stlink.next = 2ndlink;
class LinkList {

private Link first; // Reference to the first link

public LinkList() { first = null; } // constructor

// methods
public boolean isEmpty() { return (first == null); }

public void insertFirst(String name, int age) {
    Link newLink = new Link(name, age); // create link
    newLink.next = first; // newlink → old first (step 1)
    first = newLink; // first → newLink (step 2)
}

public void displayList() {
    Link current = first; // start probe at the beginning
    while (current != null) { // until the end of the list
        System.out.println(current.name + current.age);
        current = current.next; // move to next Link
    }
}

public Link deleteFirst() { // to complete
}
Linked-List: List Traversal

- When do we need to traverse the list?
  - Print all the items ...displayList
  - Find a specific item/link
  - Delete a specific item/link
  - Insert item at specific location (sorted list)
  - Count the total number of links in the list:

```java
public int size(){
    if (isEmpty()){
        return 0;
    }
    Link current = first; //start probe at the beginning
    int count=0;
    while (current!=null) {
        count++;
        current = current.next; // move to next Link
    }
    return count;
}
```
Linked-List: Find a specific link

- Work similarly to the displayList method

```java
public Link findName(String keyName){
    if (isEmpty()){
        return null;
    }
    Link current = first; //start probe at the beginning
    while (current!=null && !keyName.equals(current.name)){
        current = current.next; // move to next Link
    }
    return current; // return Link if found or null if not found
}
```

Link probe = mylist.findName("Luke");
if (probe!=null) System.out.println(probe.name+"'s age is "+probe.age);
Linked-List: Delete a specific link

- Work similarly to the find method but maintain a reference to the previous link.
  - **Delete 'Han'** (current=first)
  - **Delete 'Luke'**

```
Delete 'Han' (current=first)

first = current.next

Delete 'Luke'

previous.next = current.next
```
- Work similarly to the find method but maintain a reference to the previous link.

```java
public Link delete(String keyName) {
    if (isEmpty()) { return null; }
    Link current = first;  // start probe at the beginning
    Link previous = first;

    while (current != null && !keyName.equals(current.name)) {
        previous = current;  // save previous Link
        current = current.next;  // move to next Link
    }
    if (current == first) {
        first = current.next;
    } else if (current != null) {
        previous.next = current.next;
    }
    return current;
}
```
A double-ended list contains an additional reference to the last link.

The class `Linked List` contains the reference to the first and last link.

It is then possible to insert a new link directly at the end of the list without the need to iterate along the entire list:

- `InsertFirst` and `insertLast` have the same complexity $O(1)$
- This is suitable for some situations like for implementing a `queue`

Unfortunately, it still does not help with deleting the last link (you need the “previous reference”)

class LinkList2Ends {

private Link first; // Reference to the first link
private Link last; // Reference to the last link
public LinkList() { first = null; last = null; }
// methods

public void insertFirst(String name, int age) {
    Link newLink = new Link(name, age); // create link
    if (isEmpty()) last = newLink; // special case
    newLink.next = first; // newlink → old first (step 1)
    first = newLink; // first → newLink (step 2)
}

public void insertLast(String name, int age) {
    Link newLink = new Link(name, age); // create link
    if (isEmpty()) first = newLink; // special case
    else last.next = newlink; // (step 1)
    last = newLink; // (step 2)
}
// Other methods here
}
class FirstLastpp
{
    public static void main(String[] args)
    {
        LinkList2Ends mylist = new LinkList2Ends();
        mylist.insertFirst("Obiwan", 55);
        mylist.insertFirst("Luke", 20);
        mylist.insertFirst("Han", 40);

        mylist.insertLast("Anakin", 40);
        mylist.insertLast("Leia", 20);
        mylist.insertLast("Yoda", 400);

        mylist.displayList();
    }
}

Remark: repeated insertions at the front reverse the list of items, while repeated insertions at the end/rear preserve the order
Linked-List: Summary

- Insertion/Deletion at the beginning of the list is $O(1)$
- Insertion at the end of a double-ended linked list is $O(1)$
- Finding, deleting or inserting next to a specific link is $O(N)$
  - The number of comparisons is $O(N)$ likes for arrays
  - However, items do not need to be shifted/moved
  - Those operations are then expected to be faster than using arrays
- A linked-list uses exactly as much memory as it needs and it can expand dynamically