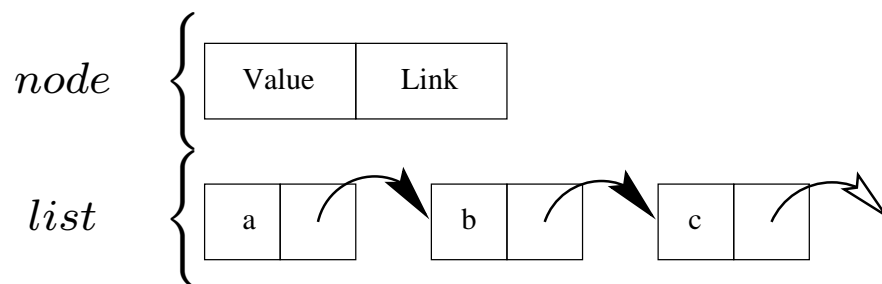


Lists

Fundamentals of Linked Lists:

A *linked list* is a structure of elements arranged one after another with each element connected to the next element by a “link”. *Lists* are probably one of the most important tools of programming, they easily allow you to keep track of an unknown amount of data, they serve as the basis for the implementation of other *Abstract Data Types* such as *stacks* or *trees*, they are the simplest example of dynamic memory allocation.

We refer to the structure that holds an *element* and a *link* to the next similar structure as a node. The reference to the first node, the beginning of the list, is called the head. The last node of a list is called the tail.



In java the end marker, or the reference in the tail that indicates the end of the list, is the null reference. If the head points to this *null* reference, we say that the list is empty. Note that there is an *exception* prepared for errors related to this object, for example you may see *NullPointerException* when trying to access the method of a *null* reference.

Typical operations on lists include :

- Inserting a new node, either at the beginning, the end, or at a specific position in an existing list.
- Removing a node, either generically such as always the first one, or a specific node.
- Calculating the number of nodes.
- Traversing the entire list (to print out the values for example).
- Concatenating two lists.

Lists can also be used with more specialized operations, for example :

- The insertion could automatically place the new item in the list in relation to the other already present elements. This would create a sorted list.
- The list could only allow new items to be added at one end of the list and removals to be done from that end. This data structure is called a stack.
- If the removals were to be done from the other end, the structure would represent a queue.

In java the class definition for a node could look like :

```
public class node
{
    private Object value;
    private node next;

    node(Object t_value, node t_next)
    {
        value = t_value;
        next = t_next;
    }

    void set_value(Object t_value)
    {
        value = t_value;
    }

    void set_next(node t_next)
    {
        next = t_next;
    }

    Object get_value()
    {
        return value;
    }

    node get_next()
    {
        return next;
    }
}
```

Adding a new node as the head of a list:

```
head = new node(value, head);
```

Removing a node from the head of a list:

```
if (head != null)
    head = head.get_next();
```

Traversing a list:

```
for (node temp = head; temp != null; temp = temp.get_next())
    ...
```

or

```
node temp = head;
while (temp != null)
{
    ...
    temp = temp.get_next();
}
```

Calculating the number of nodes:

```
int size = 0;
for (node temp = head; temp != null; temp = temp.get_next())
    size++
```

Inserting an Object after another object:

```
node temp = head;
while ((temp.get_value() != Objective) && (temp != null))
    temp = temp.get_next();

if (temp == null)
    System.out.println('Error');
else
    temp.set_next(new node(Addition, temp.get_next()));
```

Deleting a specific object :

```
if (head.get_value() == Subtraction)
    head = head.get_next();
else {
    node temp = head;

    while ((temp.get_next().get_value() != Subtraction) &&
        (temp.get_next() != null))
        temp = temp.get_next();

    if (temp.get_next() == null)
        System.out.println('Error');
    else
        temp.set_next(temp.get_next().get_next());
}
```

These are of course only one way of implementing these operations.

Although the representation of the lists that we have made is accurate, it does not offer a self-enclosed data type. The user must still make its own verification as to whether or not the list is empty as well as include all code fragments in its program. You may therefore encounter dummy classes. These enclose all the preceding code in one class. For example :

See `list.java`