Lists

List

A List represents a data structure which allows to dynamically add, access and remove objects of the same type. Adding objects to the list is usually done via the *add()* method. The *get(int i)* method allows to non-destructively retrieve the element at position i.

The list is a data structure that combines the properties of both the stack and the properties of the queue.

- Implementing it using an array does not make sense, so we will only implement using references. Such a list is called "simple linked list":
- simple - each node contains only one reference, so the connection leads in one direction only,
- linked - connection is via links, references,
- list.

Add an element to the top of the list

You start with the head of the list *head* (containing *null*) and an additional variable *pom* (reference to instance of class Node, also *null*)

Using the operator *new* we create a new element, referenced by *pom*.

fill data to the new element:

```
pom.data1 = ...
pom.data2 = ...
```

The original content of the variable *head* is copied to the *dalsi* item on the new element (in our example, there was a *null* value, so into *pom.dalsi* is inserted *null*).

```
pom.dalsi = hlava;
```

Finally, the pointer into the *hlava* will be copied the contents of *pom*, i.e. the address of the new element.

This addition is completed.

Note: we could not straight assign *hlava = new*, because we would lose the original contents of the variable *hlava*.

We proceeded this way because it is universal algorithm that can be applied to already partially filled list as well.

The picture shows the situation after the addition of the next element into the list.

Browse the list

<CZ>Seznamem se dá procházet jen jedním směrem.

Potřebujeme pomocnou proměnnou *pom*.

Začíná se vždy v *hlava*

```
pom = hlava;
```

a prochází se postupným posouváním proměnné *pom*:

```
pom = pom.dalsi;
```
</CZ>One can browse through the list in one direction only.

We need additional variable *pom*.

It starts always in *head*:

```
   pom = head;
```

browsing:

```
   pom = pom.dalsi;
```

Note that when you go through the list from the *hlava* to the end, the order of elements is from newest to oldest, i.e. LIFO. Thus stack can be implemented as a special case of a list in which we add to the beginning and read from the *head*.

Append item to the end

Appending at the end of the list is difficult, because we first had to find the end (which is time consuming).

Therefore, in addition to the reference to *hlava* we introduce yet another variable *zaKonec* which will point after the end of the list to the auxiliary element - sentinel.

At the end of the list, create auxiliary element sentinel (bumper).

The initial state of the list will look like this. Even the newly created list is not empty, because it will contain sentinel.

Browsing the list and adding elements to the beginning is pretty much the same.

The only difference is that we must stop browsing the list **before** we reach the sentinel.

Appending element on the end of the list is performed in the following steps:

- Fill data into sentinel.

- Create a new sentinel and connect it after the original sentinel.
- Reference *zaKonec* moves to a new sentinel.
- <CZ>Doplní se *null* v novém sentinelu.</CZ>Fill *null* as a reference in new sentinel.

```
zaKonec = zaKonec.dalsi;
zaKonec.dalsi = null;
```

List x Queue

Insert element into the middle

For inserting "into the middle" and for other operations we need to somehow mark the spot on the list, with whom we work.

Therefore, we introduce another variable *znacka* to mark the item.

Variable *znacka* will always point to the element of the list we are currently using.

When inserting, we can insert new item either **after the mark** or **on the mark**.

Inserting after the mark

We will use an additional variable *pom* to create and populate a new element.

The address of the next element is copied into the newly created element

```
pom.dalsi = znacka.dalsi;
```

<CZ>Adresu nového prvku vložíme do pole „dalsi“ prvku, na který ukazuje značka

```
znacka.dalsi = pom; </CZ>
```

The address of new element put into the field "dalsi" of element pointed to

```
znacka.dalsi = pom;
```

Inserting before the mark

The reason is we do not know from where comes the blue pointer, so we cannot change it where necessary (to red pointer)

We have two options now.

- Find the unknown "previous" element and thereby convert this job to solve previous task "remove the element after the marked one." It's easy, but slow.
- Or know the "trick" (see below).

Note: the exam must either know the "trick" or be able to write a program which finds the "previous" element

An initial state for „trick“ is as follows.

First, we must copy the element pointed to by the *znacka*, using a temporary variable *pom*.

Into element pointed by mark copy the entire element that follows him.

This actually tranforms job to the previous job "to remove the element after mark“, which we know how to solve. Just beware that we can not simply write

```
pom = znacka;
```

because that would have assigned references, not values (both *pom* and *znacka* would point to the same element). We must create a duplicate element using the method *copy()*:

```
pom = znacka.copy();
```

As a final step, a new element will connect to the list and fill the data

```
znacka.dalsi = pom;
```

Remove an element in the middle

Like when inserting elements in the list, also when deleting an element we will point to it by a marker.

We have basically two options: either remove the element pointed to by the marker. Or to remove the element that follows it. Both options will now discuss.

Remove element after the mark

Like the insert, deleting the element behind the mark is easy.

To auxiliary variable *pom* store address of marked element

```
pom = znacka.dalsi
```

<CZ>Změníme referenci na následující prvek seznamu

znacka.dalsi = pom.dalsi</CZ>

Change the reference to the next element in the list

```
znacka.dalsi = pom.dalsi
```

Delete unwanted item

```
pom = null;
```

Remove element with the mark

The reason is we do not know from where comes the blue pointer, so we cannot change it where necessary (to red pointer)

We have two options now.

- Find the unknown "previous" element and thereby convert this job to solve previous task "remove the element after the marked one." It's easy, but slow.
- Or know the "trick" (see below).

Note: the exam must either know the "trick" or be able to write a program which finds the "previous" element

An initial state for „trick“ is as follows.

First, we set the auxiliary variable to the element following the element pointed to by a marker

```
pom = znacka.dalsi;
```

Into element pointed by mark copy the entire element that follows him.

This actually tranforms job to the previous job "to remove the element after mark“, which we know how to solve. Just beware that we can not simply write

```
mark = pom;
```

because that would have assigned references, not values (both *pom* and *znacka* would point to the same element). We must create a duplicate element using the method copy():

```
znacka = pom.copy();
```

Then just assign *null* to *pom*, the rest will take care *garbage collector*.

```
pom = null;
```

The element is deleted.

Special lists

List is a trio

Trinity pointers *hlava* + *zaKonec* + *znacka* is so frequent and characteristic that some authors by "list" just understand this trio.

Implementation of the list using arrays

There was demonstrated through the implementation of a list using references. But it does not mean that it can not be implemented eg. using dynamic arrays.

Sorted List

It is a typical feature of list that elements are identified by their position in the list.

To increase the speed of search, you can sort the list, ie. sort its elements according to the values of the keys, but it's not very convenient (linked lists cannot easily implement halving method).

Structures in which the elements are identified by the value of its keys are called table or map.

Double Linked List

The biggest drawback of a simple list is that we can not quickly find the previous element (browsing is unidirectional).

Where it matters, there we use the double-linked list.

Content