DoubleLinkedLists

COLLABORATORS

	TITLE : DoubleLinkedLists		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		December 6, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

DoubleLinkedLists

1.1 Implementation notes

```
               The A++ double linked list classes
            ----------------------------------------
               ($Date: 1994/07/17 11:39:14 $)

The A++ Library provides several container list classes that are used
throughout the whole library. All these classes are based on the Exec®
double linked lists, the MinList and MinNode structures, which are supported
by the Amiga® system software, thus making the A++ list classes compatible
to the Exec® lists.
Of course, A++ list classes not only encapsulate access to Exec® lists but
enhance them to make their usage more comfortable and safe.

Much work has been spent to make the pointer usage within the list safer
and to prevent the user from trashing the integrity of the links.

Now, let us track down the neuralgic points of the Exec® lists that may
lead to errors but that are safe for A++ lists:

 removing an already removed node
 deleting nodes which are still within a list
 deleting a list that still possesses nodes

More features of A++ lists are..

-  virtual destructors (implementation note)
-  virtual apply method
-  getting the list to a node
-  iterating through A++ lists

----------------------------------------------------------
 -> Back to the root menu..
```

1.2 problem_1

```
Removing an already removed node:
```

```
Problem:
The Remove() implementation simply redirects the successor/predecessor
pointers of the predecesor/successor of the node to point to the successor/
predecessor of the node thus writing to memory refered to by the node.
The system probably crashes when the node is not within a list.

Solution:
The A++ lists simply ensure for all nodes that are not linked into a list
to have itself as successor/predecessor thus allowing nodes to be removed
whenever you want!
(Removing always reinitialises this circular link thus neutralizing the
pointers.)
```

1.3 problem_2

```
Deleting nodes which are still within a list:

Problem:
Deleting a node that is linked into a list, for instance, when freeing its
memory, severes the list. Subsequently running the list may lead to unpre-
dictable results.

Solution:
The deletion of an A++ list node causes the node automatically to be
removed from any list it was linked to and neutralizes its pointers.
```

1.4 problem_3

```
Deleting a list that still posesses nodes:

Problem:
Deleting a list (better: list head) would leave its nodes 'headless' thus
severing the list, too.

Solution:
Before being deleted A++ lists unlink each of their nodes from the list by
neutralizing their pointers. Again, each node is in a safe state.
```

1.5 features_1

```
A++ lists serve as base classes for many other A++ Library classes
therefore they must have virtual destructors.

That comes with an implementatory disadvantage:

When using virtual methods in a class the compiler adds a pointer to the
virtual method table to each object of that class. This pointer is placed
before or behind the class data. Derived classes data is therefore
intertwined with the virtual method table pointers.
```

```
Therefore, if we wanted to derive "Node" and "List" classes from "MinNode"
and "MinList" the representation in memory would no longer be compatible to
those Exec® functions that use the additional Node/List fields (ln_Type,
ln_Pri,ln_Name and lh_Type).

That is the reason why "NodeC" and "ListC" are not derived from "MinNodeC"
and "MinListC" though they have all "MinNodeC" and "MinListC" methods
available with the same semantics.
```

1.6 features_2

1.7 features_3

```
You can get the A++ list object to a given A++ node object by using

   list = node->findList();

   MinListC *list; MinNodeC *node;
.
If the node was not within a list NULL is being returned. This method runs
the list and therefore its speed is proportional to the length of the list.
```

1.8 features_4

```
            How to iterate through a A++ list
         ---------------------------------------

The so well known template for a 'for' loop that takes each node of an Exec®
list is slightly modified when using A++ lists:

   MinListC *list;
   for(MinNodeC *node=list->head();node;node=node->succ())
       ;

The 'head()' and 'succ()' methods only return (Min)Node objects which have
to be casted to the derived classes' objects you have linked in the list.
This formal inconvenience is due to the lack of templates being implemented
in all supported compilers.
There are a few little defines that give you typed lists. But you must make
sure that the list really consists of that type of object you are casting
it to:

Give your node type and a pointer to your list and you will get a pointer
"node" to your node type in the subsequent loop body. Of course your node
type must be derived from MinNode or Node class as must your list be derived
from MinList or List class.
This loop does NOT allow to delete the node object within the loop, since the
successor could not be read from an invalid node! Also FOREACHOF() within
another FOREACHOF loop shadowes the first node. Instead use the macro with
explicit variable name.  This hides the type conflict with the return type of
the list classes in the absence of parameterized classes (templates). As
```

```
 soon as templates are available all list classes will be doubled with template
 classes.

#define FOREACHOF(type,list) for(type *node=(type*)((list)->head());node;node=( ↩
   type*)node->succ())
#define FOREACHOFNAME(type,list,node) for(type *node=(type*)((list)->head()); ↩
   node;node=(type*)node->succ())


This loop allows deleting the object addressed in 'node'.
Add NEXTSAFE behind your loop body (behind the closing brace if there is one).

#define FOREACHSAFE(type,list) for(type *node=(type*)((list)->head()),*next=NULL ↩
   ;node;node=next) \
 { next=(type*)node->succ();

#define NEXTSAFE }
```