University of Setif 1
Faculty of Science
Department of Computer Science
Course: 2nd A. Academic Bachelor's degree
Academic year 2020-21-22-23-24

Module: Algorithmics and Data Structure (ASD3)

CH2 – Sequential Structures
2- Simply chained list

Summary:

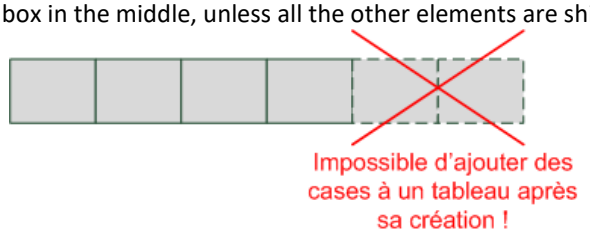
Prerequisite:

Data types
Structures
Using typedef
PointersUser functions

Introduction:

To store data in memory, we used ***simple variables*** (type int, double, etc.), ***arrays,*** and ***custom structures***. If you want to store a series of data, the easiest way is usually to use arrays.
However, the tables are sometimes quite limited. For example, if you create a table of 4 boxes and later realize in your program that you need more space, it will be impossible to enlarge this table. Similarly, it is not possible to insert a box in the middle of the table.

The problem with paintings is that they are frozen. It is not possible to enlarge them, unless new, larger ones are created. Similarly, it is not possible to insert a box in the middle, unless all the other elements are shifted.



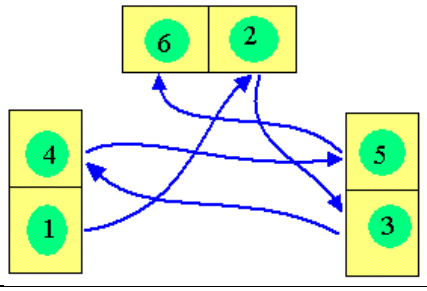
Linked lists are a way to organize data in memory in a much more flexible way.

A linked list is a way to organize a series of data in memory. This consists of assembling structures by linking them together using pointers. They could be represented as follows:

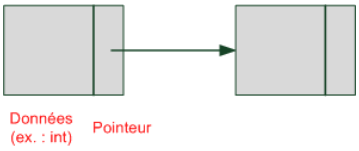
A linked list is an assembly of structures linked by pointers

Each element can contain whatever you want: one or more int, double... In addition to this, each item has a pointer to the next item.

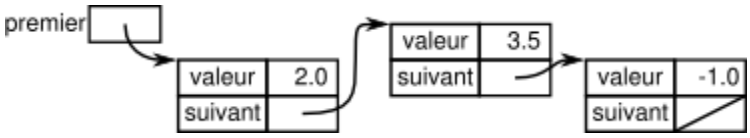
The elements form a string of pointers, hence the name "linked list".

NB: Unlike arrays, the elements of a linked list are not placed side by side "contiguous" in the memory. Each tile points to another tile in memory that is not necessarily stored right next to it.	Présentation dans la mémoire centrale 

Each element contains a piece of data (e.g., an int) and a pointer to the next element



Example



Building a Linked List

An item from the list

For our examples, we'll create a linked list of integers. Each item in the list will have the following structure:

```
typedef struct Element Element;
 struct Element
 {          Int number;
            Next element *;
 };
```

It can also be written:

```
typedef struct Element;
 struct Element
 {          Int number;
            Next element *;
 } Element;
```

- A piece of data, in this case a number of type int : we could replace this with any other data (a double, an array, etc.). This corresponds to what you want to store, it is up to you to adapt it according to the needs of your program.
- A pointer to an element of the same type called next. This is what makes it possible to link the elements to each other: each element "knows" where the next element is in memory.

The control structure

In addition to the structure we just created (which we will duplicate as many times as there are elements), we will need another structure to control the whole linked list. It will take the following form:

```
typedef struct List List ;
struct List
{          Element * prime;
};
```

This List structure contains a pointer to the first item in the list. Indeed, you have to keep the address of the first element to know where the list begins. If you know the first element, you can find all the others by "jumping" from element to element using the following pointers.

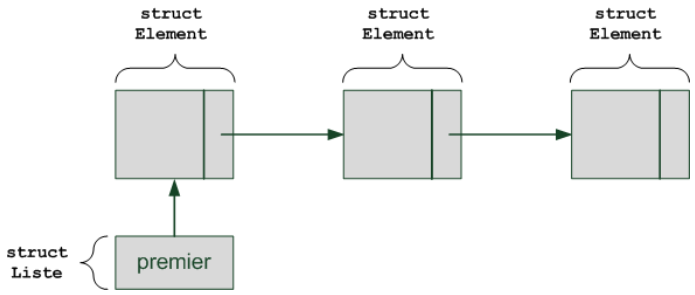
NB: A structure composed of a single subvariable is generally not very useful. Nevertheless, I think that we will need to add subvariables later, we could for example store in addition to the **size** of the list, a pointer to the **last** element.

We will only need to create one copy of the List structure. It allows you to control the entire list

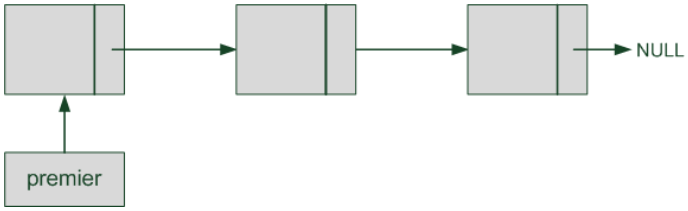
The last item on the list

Indeed, you will have to stop going through the list at some point. With what could we signify the last element in our program? It would be possible to add a pointer to the last element in the List structure. However, there is something even simpler: simply point the last item in the list to NULL, i.e. set its next pointer to NULL.

NB: we don't know what the previous element is, so it is impossible to go back from an element with this type of list. This is called a "simply chained" list, while "double-chained" lists have pointers in both directions and do not have this defect. However, they are more complex.



The List structure gives us information about the entire linked list



The last item in the list points to NULL to indicate the end of the list

Example in C

```
#include <stdlib.h>          /* to use the NULL macro */

typedef struct element element;
struct element
{ int val;
   struct element *nxt;
};

typedef element* llist;

int main(int argc, char **argv)
{ /* Let's declare 3 lists linked in different but equivalent ways */
   llist ma_liste1 = NULL;
   element *ma_liste2 = NULL;
   struct element *ma_liste3 = NULL;

   return 0;
}
```

List management functions

We have created two structures that allow you to manage a linked list:

- Element, which corresponds to an item in the list and can be duplicated as many times as necessary;
- List, which controls the entire list. We will only need one copy.

But it is still missing: the functions that will manipulate the linked list (add/modify/...):

- Initialize the list.
- Add an item.
- Delete an item.
- View the contents of the list.
- delete the entire list.

Other functions could be created (e.g. to calculate the size of the list, sort, concatenate,...)

Initialize the list

The initialization function is the very first one that needs to be called. It creates the control structure and the first item in the list.

List * initialization()

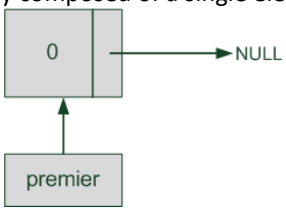
```
{       List * list = malloc ( sizeof (* list);
        Element * element = malloc ( sizeof (* element ));

        if ( list == NULL || element == NULL )
        {       exit ( EXIT_FAILURE );
        }
        element -> number = 0;
        next -> element = NULL;
        list -> first = element;

        return list;
}
```

Explanation:
We start by creating the list control structure. It is dynamically allocated with a malloc. The size to be allocated is automatically calculated with sizeof(*list).

The memory needed to store the first element is then allocated in the same way.
We check if the dynamic allocations worked. In the event of an error, the program is immediately terminated by calling exit().
If everything went well, we define the values of our first element:

- the number data is set to 0 by default;
- the next pointer points to NULL because the first item in our list is also the last one at the moment.

So we have now managed to create a list in memory composed of a single element and having the following form:

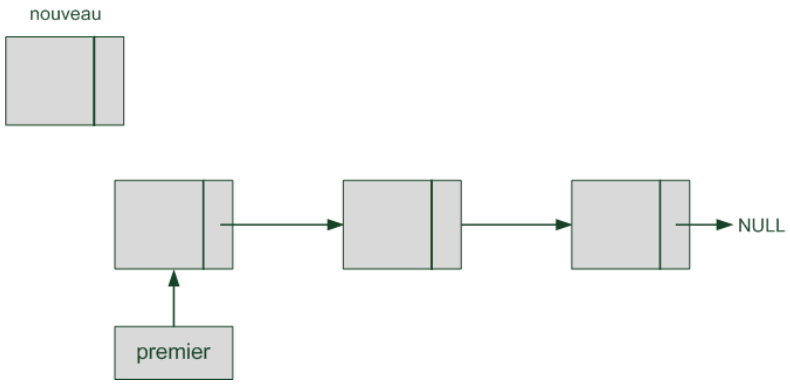


NB: we don't have to create an element, you just have to initialize only the head of the list to NULL:
list -> first = NULL;

Add an item

Where will a new element be added? At the beginning of the list, at the end, in the middle ?

NB: that the data type is List and that the variable is called List. The capital letters make it possible to differentiate them. We could also have written sizeof(List), but if later we decide to change the type of the list pointer, we will also have to adapt the sizeof.



Insert new item **at the beginning of the list** even if the list is empty

We will have to adapt the first pointer of the list as well as the next pointer of our new element to correctly "insert" it into the list.

Steps:

1. Memory allocation for the new item
2. Fill in the data field of the new item
3. The next pointer of the new item points to the 1st item

4. The first pointer points to the new item

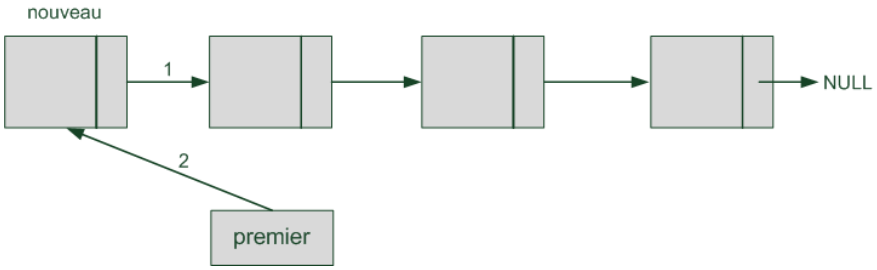
```
void insertion ( List *list , int nvNumber )
{
/* Creating the new item */
        Element * new = malloc ( sizeof (* new));   /* 1 memory allowance*/
        yew (new == NULL )
        {       exit ( EXIT \ _FAILURE );   /* in case of allocation return -1*/
        }
        new -> number = nvNumber;       /* 2 Fill in the data field for the new item */

/* Inserting the item at the beginning of the list */
        new -> next = list -> first; /*3- the next pointer of the new element points to the 1st element*/
        list -> first = new;                    /*4- the first pointer points to the new */ element
}
```

Explanation:
The insertion() function takes as a parameter the control element list (which contains the address of the first element) and the number to be stored in the new element that we are going to create.
First, we allocate the space needed to store the new element and place the new number nvNumber in it. There is then a tricky step: inserting the new item into the linked list.

For simplicity, we have chosen to insert the element at the beginning of the list. To update the pointers correctly, we need to do it in this order:
3. point our new item to its future successor, which is the current first item on the list;
4. Point the first pointer to our new item.

NB: You can't follow these steps in reverse order! This is because if you point first to our new item, you lose the address of the first item in the list! Take the test, you will immediately understand why the opposite is impossible.

This will correctly insert our new item into the linked list:



Delete an item (At the beginning of the list, at the end, in the middle)

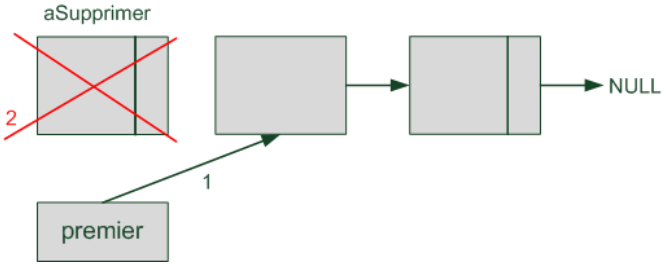
Delete the first item
The abolition does not pose any additional difficulty. However, it is necessary to adapt the pointers of the list in the right order so as not to "lose" any information.

```
void deletion ( List * list )
{       yew ( list == NULL )
        {       exit ( EXIT_FAILURE );
        }
        if (list -> prime != NULL )
        {
                Element * aDelete = list -> first;
                list -> first = list -> first -> next;
                free ( aDelete );
        }
}
```

Explanation:

- We start by checking that the pointer we are sent is not NULL, otherwise we can't work.
- We then check that there is at least one item in the list, otherwise there is nothing to do.
- Save the address of the item you want to delete in a pointer to Delete.
- We then adapt the first pointer to the new first element, which is currently in the second position of the linked list.

- All that's left to do is delete the element corresponding to our aDelete pointer with a free



NB: Things must be done in a specific order:
1. Point first to the second element;
2. Delete the first item with a free.

If we did the opposite, we would lose the skill of the second element!

View linked list

To see what is in our linked list, a display function would be ideal! Just start from the first element and display each element one by one by "jumping" from block to block.

```
void displayList ( List * list )
{       yew ( list == NULL )
        {       exit ( EXIT_FAILURE );
        }
        Current element * = list -> prime;
        while ( current != NULL )
        {       printf ("%d -> ", current -> number );
                current = current -> next;
        }
        printf(" NULL \n");
}
```

Explanation:
We start with the first element and display the content of each element in the list (a number). The next pointer is used to move to the next item each time.
We can have fun testing the creation of our linked list and its display with a hand :

```
int main ()
{       List * myList = initialization();
        insertion ( myList , 4);
        insertion ( myList , 8);
        insertion (maListe, 15);
        delete (myList);
        displayList (myList);
        return 0;
}
```

In addition to the first element (which we left here at 0), we add three new ones to this list. Then one is deleted.

Execution:
In the end, the content of the linked list will be:
8 -> 4 -> 0 -> NULL

Tip:
I advise you to group all the functions of managing the linked list in liste_chainee.c and liste_chainee.h files for example. This will be your first library! You can reuse it in any program where you need linked lists.

In short

- Linked lists are a new way to store data in memory. They are more flexible than tables because you can add and remove "boxes" at any time.
- Sequential treatment-oriented structure"
- Use whenever updates are more important than consultations
- Can be manipulated iteratively or recursively"
- Some languages offer them as basic, others don't, like C:
  There is no system for managing linked lists in the C language, we have to write it ourselves! It's a great way to progress in algorithms and programming in general.

- In a linked list, each item is a structure that contains the address of the next item.
- It is advisable to create a control structure (such as a List in our case) that retains the address of the first element.
- The cost of an algorithm is evaluated in occupied space and in the number of pointers traversed or assigned"
- We can introduce a double chaining, in which each element also has the address of the one before it (double-linked lists or bidirectional lists) to allow a journey in both directions"
- In some cases, a circular or ring list is recommended

Important note

In order to execute these functions on a C++ compiler (e.g., C++ dev) the following statements must be added:

`#include <stdio.h>`	for input/output operations (e.g., printf)
`#include<stdlib.h>`	for memory management (e.g., malloc, free, NULL)

Training

Here are a few other functions that are missing and that we invite you to write, they will be very good exercises !

Exercise 1: Add to End of List
Exercise 2: Test if the list is empty
Exercise 3: Delete an item at the end of the list
Exercise 4: Searching for an "Item" of "Some Value" in a List
Exercise n°5: Count the number of occurrences of a "value"
Exercise n°6: Search for the i-th element
Exercise 7: Retrieving the value of an element
Exercise n°9: Erase all elements with a certain "value"
Exercise 10: Completely erase a linked list from iterative and recursive memory

Algorithms

Exercise n°11: Element search (by position or value)
Exercise 12: Inserting an element at a given position
Exercise 13: Deleting an element at a given position

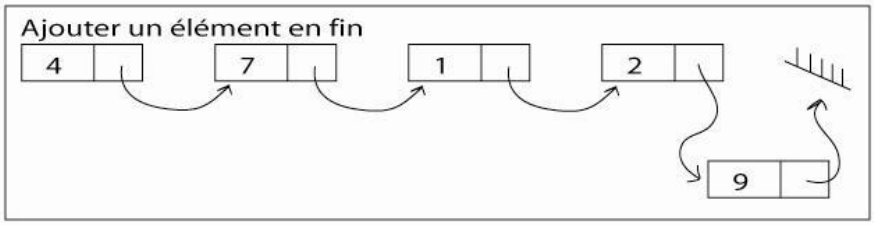
Corrected

Exercise 1: Add to End of List

We need to:

- first create a new item,
- assign it its value,
- and set the address of the next element to NULL.

Indeed, as this element will end the list, we must point out that there is no more next element. Next, we need to point the last original list item to the new item we just created.
To do this, you need to create a temporary pointer to an element that will move from element to element, and see if this element is the last one on the list. An element will necessarily be the last one in the list if NULL is assigned to its nxt field.



Code: C

```
llist addEnd(llist list, int value)
{    /* We create a new element */
     element* newElement = malloc(sizeof(element));

     /* The value is assigned to the new element */
     newElement->val = value;

     /* We add at the end, so no element will follow */
     newElement->nxt = NULL;
```

```
    if(list == NULL)
    {   /* If the list is empty, just return the created item */
        return nouvelElement;
    }
    else
    {   /* Otherwise, we browse the list using a temporary pointer and we
        indicates that the last item in the list is linked to the new item */
        element* temp=list;
        while(temp->nxt != NULL)
        {   temp = temp->nxt;
        }

        temp->nxt = newElement;
        return list;
    }
}
```

As you can see, we move along the linked list using the temp pointer. If the element pointed to by temp is not the last one (temp->nxt != NULL), we move forward one notch (temp = temp->nxt) by assigning temp the address of the next element. Once you are at the last element, all that remains is to connect it to the new element.

Exercise 2: Test if the list is empty

Write a function that returns 1 if the list is empty, and 0 if it contains at least one element. Its prototype is as follows:

Code: C

```
1 int isEmpty(llist list);
```

Code: C

```
if(isEmpty(ma_liste))
    {   printf("The list is empty");  }
    else
    {   showList(ma_liste);   }
```

Correction

Code: C

```
int isEmpty(list list)
{ if(list == NULL)
    {   return 1;    }
    else
    {   return 0;    }
}
```

Or, in condensed:

Code: C

```
int isEmpty(llist list)
{ return (list == NULL)? 1 : 0;  }
```

If the list is NULL, it does not contain any items, so it is empty. Otherwise, it contains at least one element.

Exercise 3: Delete an item at the end of the list

This time, you will have to go through the list to its last item, indicate that the second-to-last item will become the last item in the list and release the last item to finally return the pointer to the first item in the original list.

Code: C

```
llist deleteElementEnFin(llist list)
{   /* If the list is empty, we return NULL */
    if(list == NULL)
        return NULL;

    /* If the list contains only one item */
    if(list->nxt == NULL)
    {   /* We release it and return NULL (the list is now empty) */
        free(list);
        return NULL;
```

```
    }

    /* If the list contains two or more items */
    element* tmp = list;
    element* ptmp = list;

    /* As long as we are not at the last element */
    while(tmp->nxt != NULL)
    {   /* ptmp stock the address of tmp */
        ptmp = tmp;
        /* We move tmp (but ptmp keeps the old value of tmp */
        tmp = tmp->nxt;
    }
    /* At the end of the loop, tmp points to the last element, and ptmp to
    the penultimate one. It is indicated that the penultimate becomes the end of the
list
    and we delete the last element */

 ptmp->nxt = NULL;
    free(tmp);
    return list;
}
```

Exercise 4: Searching for an "Item" of "Some Value" in a List

The goal of the game this time is to return the address of the first item found with a certain value. If no element is found, we will return NULL. The interest is to be able, once the first element has been found, to search for the next occurrence by searching from elementFind->nxt. So you go through the list to the end, and as soon as you find an item that matches what you're looking for, you return its address.

Code: C

```
llist searchElement(llist list, int value)
{   element *tmp=list;
    /* As long as we are not at the end of the list */
    while(tmp != NULL)
    {   if(tmp->val == value)
        {   /* If the element has the desired value, its address is returned */
            return tmp;
        }
        tmp = tmp->nxt;
    }
    return NULL;
}
```

Exercise n°5: Count the number of occurrences of a "value"

To do this, we'll use the previous function to search for an item. We look for a first occurrence: if we find it, then we continue the search from the next element, as long as there are occurrences of the value we are looking for.
It is also possible to write this function without using the previous one of course, by going through the entire list with a counter that you increment each time you pass over an element with the desired value.

Code: C

```
int numberOccurences(llist list, int value)
{   int i = 0;

    /* If the list is empty, we return 0 */
    if(list == NULL)
        return 0;

    /* Otherwise, as long as there is still an element with the value = value */
    while((list = searchElement(list, value)) != NULL)
    {   /* We increment */
        list = list->nxt;
        i++;
    }
    /* And we return the number of occurrences */
```

```
    return i;
}
```

Exercise n°6: Search for the i-th element

Simply move i times with the tmp pointer along the linked list and return the element to the subscript i. If the list contains less than i element(s), then we will return NULL.

Code: C

```
llist element_i(llist list, int index)
{   int i;
    /* We move from i boxes, as long as it's possible */
    for(i=0; i<index & list != NULL; i++)
    {  list = list->nxt;
    }

    /* If the element is NULL, the list contains fewer than i elements */
    if(list == NULL)
    { return NULL;
    }
    else
    {   /* Otherwise we return the address of the element i */
        return list;
    }
}
```

Exercise 7: Retrieving the value of an element

This is a function similar in style to the function isEmpty. All you have to do is return the value of an element to the user. You will have to return an entire error code if the item does not exist (the list is empty),
In this code, I consider that we are only working with positive integers, so we will return -1 for an error.
Another solution is to return a pointer to int instead of an int, leaving you with the option to return NULL.

Code: C

```
#define ERROR -1
int value(llist list)
{
    return((list == NULL)?ERROR:(list->val));
}
```

Exercise 8: Count the number of items in a linked list

We go through the list from end to end and increment by one for each new item you find.
Until now, we have only used **iterative** algorithms that consist of closing until we are at the end. We will see that there are algorithms that are called **recursive** and which in fact consist of asking a function to call itself.

NB: to create a recursive algorithm, you need to know the **stop condition** (or exit condition) and the recurrence condition. In fact, you have to imagine that your function has done its job for the following n-1 elements, and that all that remains is to deal with the last element.

Code: C "**recursive**"

```
int numberElements(llist list)
{   /* If the list is empty, there are 0 items */
    if(list == NULL)
        return 0;

    /* Otherwise, there is an element (the one we are dealing with)
    plus the number of items in the rest of the list */
    return numberElements(list->nxt)+1;
}
```

Code: "**Iterative**" C

```
int numberElements(llist list)
{   int nb = 0;
    element* tmp = list;
```

```
    /* We browse the list */
    while(tmp != NULL)
    {   /* We increment */
        NB++;
        tmp = tmp->nxt;
    }
    /* Return the number of items browsed */
    return nb;
}
```

We simply browse the list until we have reached the end, and we increment the nb counter that we return to finish.

Exercise n°9: Erase all elements with a certain "value"

I give you the recursive code as an indication, you can recode this function yourself with an iterative algorithm.

Code: C

```
llist deleteElement(llist list, int  value)
{   /* if Empty List, there is nothing left to delete */
    if(list == NULL)
        return NULL;

    /* If the item being processed needs to be deleted */
    if(list->val == value)
    { /* Delete it, taking care to memorize the address of the next element */
      element* tmp = list->nxt;
      free(list);
      /* The item has been deleted, the list will start at the next item
      pointing to a list that no longer contains any items with the value
      Sought*/

      tmp = deleteElement(tmp, value);
      return tmp;
    }
    else
    {  /* If the item being processed should not be deleted, then the
       final list will start with this item and follow a list that no longer contains
       of an item with the desired value */

       >nxt-list = deleteElement(nxt-list>nxt value);
       return list;
    }
}
```

Exercise 10: Completely erase a linked list from iterative and recursive memory

We will now write a function to completely clear a linked list from memory. I suggest that you write with an iterative algorithm at first, then a second time thanks to a recursive algorithm. In the first case, you need to go through the list, store the next item, delete the current item, and move forward one box. At the end the list is empty, we will return NULL.

Code: C

```
llistclearlist(llist list)
{   element* tmp = list;
    element* tmpnxt;

    /* As long as we are not at the end of the list */
    while(tmp != NULL)
    {   /* We store the following element so that we can then move forward */
        tmpnxt = tmp->nxt;
        /* We delete the current element */
        free(tmp);
        /* We move forward one square */
        tmp = tmpnxt;
    }
```

```
    /* The list is empty: we return NULL */
    return NULL;
}
```

Code: C

```
llistclearlist(llist list)
{ if(list == NULL)
    If   the list is empty, there is nothing to delete, we return
        an empty list i.e. NULL */
        return NULL;
    }
    else
    {   /* Otherwise, we delete the first element and return the rest of the
        deleted list */
        element *tmp;
        tmp = list->nxt;
        free(list);
        return deleteList(tmp);
    }
}
```

Algorithms

Exercise n°11: Element search (by position or value)

Item Search

The following two functions are not intended to be used directly, but rather constitute a toolbox for the procedures and functions that will follow. They allow you to find the element that meets a certain criterion in a linked list, and return a pointer to the corresponding `element` structure . The two search criteria are the position of the item in the list or its value.

Search by Position

This function returns a pointer to the element at the `pos` position where the first element has position 1. If the list contains fewer `pos` elements, the null pointer is returned.

```
Function Telement* recherche_par_position (first, pos)
    first telement* entry
     integer POS input
    pre-condition p > 0
    Post-relationship see above

/* iterative version */
beginning
    Remotely* c ← first
    integer p ← 1
    as long as c ≠ null and p < pos do
        c ← (*c).next
        p ← p + 1
    end so much
    Go back c
end

/* recursive version */
beginning
    If prime = null or pos = 1 then
        Return First
    otherwise
        Returns recherche_par_position ((*first).next, pos - 1)
    end if
end
```

The recursive version deserves a comment.

The clause *then* deals with two simple cases at once: the empty list (`first = null`) and the non-empty list of which we are looking for the first element (`pos = 1`). In both cases, it is the first pointer that should be returned, either because it is null or because it points to the first element.

The otherwise clause deals with the complex case. If we are looking for the second element in the list, for example, we are looking for the first element in the list of the following, if we are looking for the third, it is the second in the list of the following, and so on. This justifies passing the `argument pos - 1` in the recursive call.

Search by Value

This function returns a pointer to the *first* element found with a value of `value`, or the pointer null if no element has that value.

```
function Télément* recherche_par_valeur (first, val)
    first telement* entry
     Val Floating Entrance
    Post-relationship see above

/* iterative version */
beginning
    Remotely* c ← first
    as long as c ≠ null and (*c).value ≠ worth doing
        c ← (*c).next
    end so much
    Go back c
end

/* recursive version */
beginning
    If prime = null or (*first).value = val then
        Return First
    otherwise
        returns recherche_par_valeur ((*first).next, val)
    end if
end
```

Exercise 12: Inserting an element at a given position

Allocation and release of any element

You can now use the `recherche_par_position` and `recherche_par_valeur` procedures to insert or remove any item from a linked list.

Inserting an element at a given position

This procedure inserts a new element at the pos position and with the value `value`. The position can be between 1 (insert at the top) and `size (first) + 1` (insert after the last item). If the position is too high, the inserted parameter will be FALSE (in all other cases, it will be TRUE).

```
Proc insère_pos (First, POS, Val, Inserted)
    Entry/Exit Remote* first
     integer POS input
     Val Floating Entrance
    Logic output inserted
    pre-condition pos > 0
beginning
    inserted ← TRUE
    if pos = 1 then
        insère_en_tête (first, val)
    otherwise
        Previous Telt* ← recherche_par_position (first, pos - 1)
        if previous ≠ null then
            insère_en_tête ((*previous).next, val)
        otherwise
            inserted ← FALSE
        end if
    end if
end
```

Note that the first parameter of `insère_en_tête` is an input/output parameter. When it is passed (`*previous).next`, the `next attribute` of the element pointed to by `previous` will be *modified* (to point to the newly created element).

Exercise 13: Deleting an element at a given position

Delete an item at a given position

This procedure deletes the element at the `pos` position. The position can be between 1 and `size (first)`. If the position is too high, the deleted parameter will be FALSE (in all other cases, it will be TRUE).

```
Proc libère_pos (First, POS, VAL, Deleted)
    Entry/Exit Remote* first
     integer POS input
    Logic output removed
    pre-condition pos > 0
beginning
    deleted ← TRUE
    if pos = 1 then
        libère_premier (first)
    otherwise
        Previous Telt* ← recherche_par_position (first, pos - 1)
        if previous ≠ null then
            libère_premier ((*previous).next)
        otherwise
            deleted ← FALSE
        end if
    end if
end
```

Note that the first parameter of `libère_premier` is an input/output parameter. When it is passed `(*previous).next`, the `next attribute` of the element pointed to by `previous` will be *modified* (to point to the successor of the deleted element).

Other uncorrected exercises:

1. List copying
2. Inserting into an ordered list
3. Sorting, merging a chained list: see https://forums.commentcamarche.net/forum/affich-16096903-fusion-de-deux-liste-simplementchaine#answers
4. Search for minimum, maximum and many other things...
5. Create a list with the first n integers in descending order.
6. Calculate the average of a list of integers.

Some references:

WEBOGRAPHY

- https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c/19733-les-listes-chainees
- https://perso.liris.cnrs.fr/pierre-antoine.champin/enseignement/algo/listes_chainees/
- http://deptinfo.cnam.fr/Enseignement/CycleA/SD/cours/structuress%E9quentielleschain%E9es.pdf
- http://pauillac.inria.fr/~maranget/X/421/poly/listes.html#toc2
- https://www.mongosukulu.com/index.php/en/contenu/informatique-et-reseaux/algorithme/663-les-listes-chainees?showall=1
- http://sdz.tdct.org/sdz/les-listes-chainees-2.html
- cours.thirion.free.fr

PDF documents:

DVDMIAGE_Algo_Chapitre_10_Listes.pdf
Chained Lists dynamiques.pdf

NB: Proper operation of examples/exercises is not guaranteed with newer versions of development software.