```c
#include<stdlib.h>
#include<stdio.h>

#include"commit.h"

static int nextId = 0;

/**
  * new_commit - alloue et initialise une structure commit correspondant au parametre
  * @major:      numero de version majeure
  * @minor:      numero de version mineure
  * @comment:    pointeur vers une chaine de caracteres contenant un commentaire
  */
struct commit *new_commit(unsigned short major, unsigned long minor, char *comment)
{
       /* TODO : Exercice 3 - Question 2 */
       struct commit *my_commit;
       my_commit = (struct commit*)malloc(sizeof(struct commit));
       my_commit->id = nextId++;
       (my_commit->version).major = major;
       (my_commit->version).minor = minor;
       (my_commit->comment) = comment;
       //new_c->prev = NULL;
       //new_c->next = NULL;
       /* Exercice 4 - De l'intérêt des list_head */
       INIT_LIST_HEAD(&(my_commit->list)); // Question 4.3
       /* ------------------------------------------ */
       /* Exercice 5 - Double liste et raccourcis */
       INIT_LIST_HEAD(&(my_commit->major_list));
       my_commit->major_parent = &(my_commit->major_list);
       /* ------------------------------------------ */
  return my_commit;
}

/**
  * insert_commit - insert sans le modifier un commit dans la liste doublement chainee
  * @from:       commit qui deviendra le predecesseur du commit insere
  * @new:        commit a inserer - seul ses champs next et prev seront modifie
  */
static struct commit *insert_commit(struct commit *from, struct commit *new)
{
       /* Exercice 3 - Implémentation artisanale d'une liste doublement chaînée */
       /* TODO : Exercice 3 - Question 3 */
       /*struct commit *temp;
       temp = last->next;
       last->next = new;
       new->prev = last;
       new->next = temp; */
       /* ------------------------------------------ */
       /* Exercice 4 - De l'intérêt des list_head */
       list_add(&(new->list), &(from->list)); // Question 4.3
       /* ------------------------------------------ */
  return new;
}

/**
  * add_minor_commit - genere et insert un commit correspondant a une version mineure
  * @from:           commit qui deviendra le predecesseur du commit insere
  * @comment:        commentaire du commit
  */
struct commit *add_minor_commit(struct commit *from, char *comment)
{
       /* Exercice 3 - Implémentation artisanale d'une liste doublement chaînée */
       /* TODO : Exercice 3 - Question 3 */
       struct commit *my_new_commit = new_commit((from->version).major, (from->version).minor+1, comment);

       /* Exercice 5 - Double liste et raccourcis */
```

```c
	my_new_commit->major_parent = (from->major_parent); // Nous avons le même parent

	struct commit *inserted = insert_commit(from, my_new_commit); /* Question 3.3 */
  return inserted;
}

/**
  * add_major_commit - genere et insert un commit correspondant a une version majeure
  * @from:           commit qui deviendra le predecesseur du commit insere
  * @comment:        commentaire du commit
  */
struct commit *add_major_commit(struct commit *from, char *comment)
{
	/* TODO : Exercice 3 - Question 3 */
	struct commit *my_new_commit = new_commit((from->version).major+1, 0, comment);

	/* Exercice 5 - Double liste et raccourcis */
	my_new_commit->major_parent = &(my_new_commit->major_list);
	list_add(&(my_new_commit->major_list), (from->major_parent));

	struct commit *inserted = insert_commit(from, my_new_commit);
  return inserted;
}

/**
  * del_commit - extrait le commit de l'historique
  * @victim:         commit qui sera sorti de la liste doublement chainee
  */
struct commit *del_commit(struct commit *victim)
{
	/* TODO : Exercice 3 - Question 5 */
	//victim->prev->next = victim->next;
	//victim->next->prev = victim->prev;
	list_del(&(victim->list)); /* Exercice 4 - De l'intérêt des list_head */
  return NULL;
}

/**
* Exercice 6 : Audit mémoire et destruction d’une liste
* Problématique: Il y a des block de mémoire inoutil après la terminaison du programme
*      valgrind –leak-check=full ./testCommit
* freeHistory - Libérer la mémoire occupée par l'ensemble des éléments d'une liste de commit
* @from: premier élément de la liste
* @Notes: Faire attention pour chaque fois qu'on supprime un élément => list_for_each_safe
*/
void freeHistory(struct commit *from){
	struct commit *ptr = from;
	struct list_head *pos, *tmp;
	/* #define list_for_each_safe(pos, n, head)
	for (pos = (head)->next, n = pos->next; pos != (head); pos = n, n = pos->next) */
	list_for_each_safe(pos, tmp, &(from->list))
	{
		free(list_entry(pos, struct commit, list));
	}
	free(from);
}

/**
  * display_commit - affiche un commit : "2:  0-2 (stable) 'Work 2'"
  * @c:             commit qui sera affiche
  */
void display_commit(struct commit *c)
{
	/* TODO : Exercice 3 - Question 4 */
	printf("%d: ", c->id);
	display_version(&(c->version), is_unstable_bis);
	printf("\t\t'%s'\n", c->comment);
```

```c
}

/**
  * display_history - affiche tout l'historique, i.e. l'ensemble des commits de la liste
  * @from:           premier commit de l'affichage
  */
void display_history(struct commit *from)
{
       /* TODO : Exercice 3 - Question 4 */
       /*struct commit next;
       next = from;
       while(next != NULL){
              display_commit(next);
              next=next->next;
       }*/

       /* Exercice 4 - De l'intérêt des list_head */
       struct list_head *pos;
       pos = from;
       display_commit(pos);
       list_for_each(pos, &(from->list)){
              display_commit(list_entry(pos, struct commit, list));
       }
       printf("\n");
}

/**
  * infos - affiche le commit qui a pour numero de version <major>-<minor>
  * @major: major du commit affiche
  * @minor: minor du commit affiche
  */
void infos(struct commit *from, int major, unsigned long minor){
       /* TODO : Exercice 3 - Question 5 */
       /*struct commit *ptr;
       ptr = from;
       while(ptr != NULL){
              if((ptr->version).major == major && (ptr->version).minor == minor){
                     display_commit(ptr);
                     return;
              }
       }*/

       /* Exercice 4 - De l'intérêt des list_head */
       struct commit *ptr = from;
       struct commit *ptr2 = from;
       struct list_head *pos;
       struct list_head *pos2;

       list_for_each(pos2, &(from->major_list)){
              ptr2 = list_entry(pos2, struct commit, major_list); // chaque major commit
              if (((ptr2->version).major == major)){
                     list_for_each(pos, &(ptr2->list)){ // parcourir chaque minor commit de major commit
trouvé
                            ptr = list_entry(pos, struct commit, list); // un minor commit
                            if (((ptr->version).major == major) && ((ptr->version).minor == minor)){
                                   display_commit(ptr);
                                   return;
                            }
                     }
              }
       }
       /* ------------------------------------------- */

       printf("%2u-%lu Not Here !!!\n", major, minor);
}

/**
```

```c
  * commitOf - retourne le commit qui contient la version passe en parametre
  * @version:  pointeur vers la structure version dont cherche le commit
  * Note:      cette fonction continue de fonctionner meme si l'on modifie
  *            l'ordre et le nombre des champs de la structure commit.
  */
struct commit *commitOf(struct version *version){
       /* TODO : Exercice 2 - Question 2 */
  return (struct commit *)((void *)version - (void *)(&(((struct commit *)0)->version)));
}
```