SDD TP1: Polynomes

Auteurs: DUREL Enzo, VILLEPREUX Thibault
Référent: BURON Maxime

March 7, 2023

Sommaire

List of Figures

List of Algorithms

Présentation

Notre objectif principal est de pouvoir construire des fonctions de base pour manipuler des polynômes. Nous essayons donc de nous rapprocher le plus possible de la programmation modulaire avec les contraintes du langage C, pour pouvoir importer par la suite notre objet polynôme dans d'autres projets.

Nous pouvons retrouver comme fonctions de base :

- la dérivation d'un polynôme,
- l'addition de deux polynômes
- et la multiplication de deux polynômes.

Un polynôme est stocké sous forme d'une liste simplement chaînée (aussi appelé linkedlist) de monôme. Il est donc nécessaire, de créer aussi des fichiers contenant des fonctions pour manipuler des listes simplement chaînées, et des fichiers pour manipuler des monômes, afin de respecter l'approche de la programmation modulaire.

De plus, les maillons de la liste simplement chaînée sont rangés par ordre de degré croissant et il ne peut pas y avoir deux monômes différents avec un même degré, ou un monôme avec un coefficient nul.

Il nous a été imposé de représenter des polynômes avec une linkedlist plutôt qu'avec un tableau pour les avantages suivants :

- gérer la flexibilité de la taille plus facilement, exemple : avec la multiplication, le degré du polynôme peut être très vite élevé
- une meilleure complexité dans beaucoup de cas, exemple : pour insérer lors de l'addition pas d'appel de decaleGauche ou de decaleDroite sur un tableau, insertion en O(1) pour une linkedlist car on connaît le précédent
- gain en complexité spaciale : on ne représente pas les monômes de coefficient nul
- facilité de manipulation : insertion et suppression plus facile par exemple

Dans la suite du rapport, nous expliquerons les structures de données, et détaillerons les principes des fonctions.

Structures de Données

2.1 Description

2.1.1 Monôme

Un monôme est stocké sous forme d'une structure en C, son type est $\mathbf{monom_t}$.

Cette structure possède deux champs :

- un champ coef, de type double, qui représente comme son nom l'indique le coefficient du monôme.
- un champ degree, de type unsigned int, qui représente comme son nom l'indique le degré du monôme. Nous avons fait le choix de choisir une variable non signée afin de provoquer une erreur dans le programme si le degré devient négatif, car le polynôme ne doit pas avoir de degré négatif. Ainsi, nous pourrons en théorie stocker des degrés de taille un bit plus grand.

2.1.2 Liste simplement chaînée

Une liste simplement chaînée ou linkedlist est stockée sous forme de structure, son type est **cell$_t$**.

Cette structure possède deux champs :

- un champ val, de typage monom$_t$, qui représente les informations stockées sous la forme d'un monôme.
- un champ next, de typage cell$_t$ *, qui représente un pointeur vers le maillon suivant du maillon actuel.

Pour résumer, chaque maillon de la liste possède trois informations :

- le degré du monôme du maillon
- le coefficient du monôme du maillon
- et un pointeur vers le prochain maillon, qui peut être NULL si le maillon courant est le dernier de la liste.

2.1.3 Polynôme

Un polynôme est une liste simplement chaînée. Il n'y a pas de structure polynôme spécialement défini.

Le fichier source **polynome.c** possède seulement des fonctions pour manipuler cet objet. Et c'est grâce à ces dernières, que l'on peut garantir de l'unicité d'un maillon pour le degré monôme, ou la bonne insertion d'un monôme dans le polynôme, en respectant l'ordre de degré croissant par exemple.

2.2 Schéma

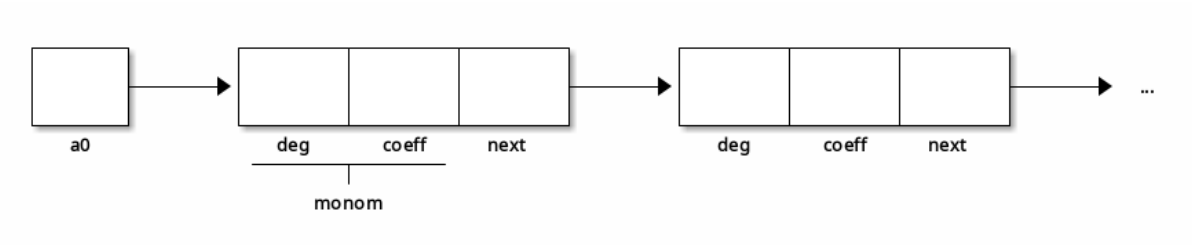

Fig. 1: Schéma d'un polynome non vide:

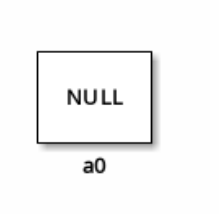

Fig. 2: Schéma d'un polynôme vide:

On retrouve les cases mémoires expliquées précédemment comme le degré d'un monôme, le coefficient d'un monôme, et le pointeur next de la liste chainée. La variable a0 quand à elle est un pointeur vers

le premier maillon du polynome/linkedlist. De plus, le contenu de la case next du dernier maillon vaut NULL.

2.3 Fichiers de données

La mise en place de tests unitaires a permis de vérifier la validité des fonctions, en détectant les erreurs dans le comportement des fonctions isolées, ce qui a contribué à améliorer la qualité et la fiabilité du code.

Afin de garantir une couverture la plus complète possible de toutes les situations et des différents cas d'utilisation, nous avons utilisé divers fichiers de test. Ces fichiers sont présents dans /src/data.

Les fichiers que l'on a créés sont :

- poly1.txt à poly9.txt : pour les tests dans $\text{polynomial}_{\text{main.c}}$.
- listeChainneeTest.txt : pour les tests dans $\text{linkedlist}_{\text{main.c}}$, il y a un seul fichier test car on fait beaucoup d'insertion à la main, pour tester la fonction $\text{LL}_{\text{addcell}}$.

Architecture

Dans le dossier **tp1** se trouve plusieurs fichiers et répertoires. Il y a :

- le répertoire **rapport** qui contient le rapport en pdf.
- le répertoire **src** qui contient l'ensemble des fichiers de code et qui contient à son tour le répertoire **data** avec tous les fichiers de test
- le fichier README qui contient des informations de pré-requis, installations, et sur les auteurs

Comme expliqué précédemment, afin de manipuler les polynômes, nous avons créé un objet monôme et un objet linkedlist. Les fonctions pour manipuler des monômes se trouve dans **valCell.c** et leurs prototypes dans **valCell.h**. Les fonctions pour manipuler des linkedlist se trouve dans **linkedList.c** et leurs prototypes dans **linkedList.h**. Il y a aussi le fichier $\textbf{linkedlist}_{\textbf{main.c}}$ qui contient les tests unitaires pour certifiée la validité des méthodes.

Fonctions

4.1 linkedList

4.1.1 $\text{LL}_{\text{initlist}}$

1. Description

   `La fonction initialise la liste à NULL.`

2. Algorithme

   Algorithm 1: Procédure d'initialisation d'une liste

   Data: ES: adrHeadPt

   1 $adrHeadPt \leftarrow NIL$;

3. Signature

```c
void LL_init_list(cell_t **adrHeadPt);
```

4. Code source

```c
/**
 * @fn void LL_init_list(cell_t **adrHeadPt)
 * @brief Initialize a void list
 * @param [in, out] adrHeadPt address of head pointer of the list
 */
void LL_init_list(cell_t **adrHeadPt)
{
    // initialise la liste a NULL
    *adrHeadPt = NULL;
}
```

5. Lexique
   (a) Paramètres
      - `cell_t ** adrHeadPt`: adresse du pointeur de tête de liste
   (b) Variables
      - Aucune variable dans cette fonction.

4.1.2 $\mathbf{LL_{createcell}}$

1. Description

   `La fonction créée et initialise une nouvelle cellule. Elle renvoie NULL si l'allocation a échoué, sinon l'adresse de la cellule allouée.`

2. Algorithme

Algorithm 2: Fonction de création de cellule

Data: E: $pmonome \neq NIL$
$new \leftarrow NIL$;
$new \leftarrow alloc(taille(cellule))$;
if $new \neq NIL$ **then**
    $m(coef(new)) \leftarrow coef(pmonome)$;
    $m(degree(new)) \leftarrow degree(pmonome)$;
    $m(next(new)) \leftarrow NIL$;
return *new;*

3. Signature

```c
cell_t * LL_create_cell(monom_t * pmonome);
```

4. Code source

```c
/**
 * @brief create a new cell for linked list from its data
 * @param [in] pdata address of the data
 * @return address of the new cell
 */
```

```
6  cell_t * LL_create_cell(monom_t * pmonome)
7  {
8      cell_t * new = NULL;
9      // alloue dynamiquement le monome
10     new = (cell_t *) malloc(sizeof(cell_t));
11     // si l'alloc a fonctionnee
12     if (NULL != new)
13     {
14     new->val.coef = pmonome->coef; //init le coef
15     new->val.degree = pmonome->degree; //init le degre
16     new->next = NULL; //init le suivant
17     }
18
19     return new; // retrun NULLsi erreur malloc
20 }
```

5. Lexique
   (a) Paramètres
      - `monom_t * pmonome`: adresse du monome à ajouter à la liste.
   (b) Variables
      - `cell_t * new`: pointeur vers la nouvelle cellule créée.
6. Hypothèse Le monôme n'est pas NULL et contient des valeurs cohérentes. Par exemple, si on l'utilise pour représenter des polynômes, le coefficient n'est pas nul et le degré est positif.

4.1.3 $LL_{addcell}$

1. Description

   `La fonction ajoute une cellule à une liste à partir d'un précédent.`

2. Algorithme

Algorithm 3: Procédure d'ajout de cellule

Data: ES: $a \neq NIL$, $prec \neq NIL$

1 $next(a) \leftarrow cm(prec)$;
2 $m(cm(prec)) \leftarrow a$;

3. Signature

```
void LL_add_cell(cell_t ** prec, cell_t * a);
```

4. Code source

```
1  /**
2   * @brief Insert a cell into a linked list at the given position
3   * @param [in, out] prec address of previous pointer of the cell
4   * @param [in] a address of the cell to be added to the linked list
5   */
6  void LL_add_cell(cell_t ** prec, cell_t * a)
7  {
```

```
8       a->next = *prec; // modifie le suivant du maillon qu'on insert
9       *prec = a; // realise le chainge avec prec
10  }
```

5. Lexique
   (a) Paramètres
      - `cell_t * a`: adresse de la cellule à ajouter.
      - `cell_t ** prec`: adresse de la cellule qui précède à après ajout.
   (b) Variables
      - Aucune variable dans cette fonction.
6. Hypothèse La cellule que l'on ajoute a été allouée précédemment, elle ne peut pas être NULL.
7. Liste des cas $\text{LL}_{\text{addcell}}$ quand :
   - la liste est vide
   - il faut insérer au milieu de liste
   - il faut insérer en tête de liste
   - il faut insérer en fin de liste

4.1.4 $\text{LL}_{\text{createlistfromFileName}}$

1. Description

   ```
   La fonction créée une liste à partir d'un nom d'un fichier. Si le fichier
   n'a pas réussi à être ouvert, la liste sera vide. Si une erreur d'allocation a
   lieu lors de la création des maillons pendant la lecture du fichier, la
   liste sera libérée proprement et la fonction renverra la liste vide.
   ```

2. Algorithme

Algorithm 4: Fonction de création de liste à partir d'un nom de fichier

Data: E: $filename$, ES: $a0$
$ouvrir(filename)$;
while *non fin(filename)* **do**
  $m(mread) \leftarrow lireMonome(filename)$; ▷ `Lit le monome dans le fichier`
  $m(curr) \leftarrow creerCellule(mread)$;
  $m(prec) \leftarrow chercherPrecedent(curr, pfcmp)$;
  $ajouterCellule(prec, curr)$;
return $a0$;

3. Signature

```
cell_t ** LL_create_list_fromFileName(cell_t ** a0,
                    char * fname,
                    int (*pfcmp) (monom_t *, monom_t *));
```

4. Code source

```c
/**
 * @brief Create a linked list from a file
 * @param [in, out] a0 address of head pointer of a linked list
 * @param [in] fname name of a file containing the data for a linked list
 * @param pfcmp fonction pointer for comparison of two cell's value
 * @return head pointer of the linked list
 */
cell_t ** LL_create_list_fromFileName(cell_t ** a0,
                    char * fname,
                    int (*pfcmp) (monom_t *, monom_t *))
{
    FILE * fp = fopen(fname, "r"); // si fname = NULL alors fp := NULL
    cell_t ** prec = NULL;
    cell_t * curr = NULL;
    int error = 0 ; // = 1 si erreur malloc
    monom_t mread;

    LL_init_list(a0);

    if (NULL != fp) { // ouverture fichier OK
    //lire une ligne
    while ( 2 == fscanf(fp, "%lg %d\n", &mread.coef, &mread.degree )
        && !error)
    {
        //crer cellule
        curr = LL_create_cell(&mread);

        if (NULL != curr) // si malloc reussi
        {
        //insrer cellule
        prec = LL_search_prev(a0, &mread, pfcmp);
        LL_add_cell(prec, curr);
        }
        else //message erreur malloc + free liste
        {
        error = 1;
        fprintf(stderr,
            "%s : Erreur problme lors de l'allocation : %s\n",
            __FUNCTION__,
            fname);
        // libere toute la liste en cas d'erreur
        LL_free_list(a0);
        }
    }
    fclose(fp);
    }
    else { // message erreur ouverture
    fprintf(stderr,
        "%s : 'Erreur lors de l'ouverture du fichier: %s'\n",
        __FUNCTION__,
        fname);
    }
    return a0; // retourne la liste
}
```

5. Lexique
   (a) Paramètres
      - `char * filename`: nom du fichier.
      - `cell_t ** a0`: pointeur vers le début de la liste.
      - `int * pfcmp (monom_t* , monom_t* )`: pointeur de fonction qui compare les monomes en paramètres.
   (b) Variables
      - `cell_t * curr`: adresse de la cellule lue.
      - `cell_t ** prec`: adresse de la cellule qui précède lue d'après la fonction de comparaison.
      - `FILE * fp`: fichier de lecture.
      - `int * error`: erreur d'allocation de la mémoire pour curr.
      - `monom_t mread`: monome qui est lu dans le fichier et qui sert à créer la cellule.
6. Hypothèse La fonction pfcmp existe et elle est utilisable/fonctionelle.
7. Liste des cas $\text{LL}_{\text{createlistefromFileName}}$ foncitonne quand :
   - le fichier n'existe pas
   - le fichier est vide
   - le fichier contient des lignes sous la forme suivante : "double entier "

4.1.5 $\text{LL}_{\text{savelisttoFile}}$

1. Description

   `La fonction écrit la liste dans un fichier pour la sauvegarder. Le fichier doit avoir été ouvert avant l'appel à la fonction.`

2. Algorithme

Algorithm 5: Procédure de sauvegarde d'une liste dans un fichier

Data: E: $pfprint$, ES: $file \neq NIL$, $head$

$m(curr) \leftarrow cm(head)$;
while $curr \neq NIL$ **do**
    $pfprint(file, val(curr))$; ▷ `Ecrit le monome dans le fichier`
    $m(curr) \leftarrow next(curr)$;

3. Signature

```
void LL_save_list_toFile(FILE * file, cell_t * head,
           void (*pfprint) (FILE *, monom_t *))
```

4. Code source

```
1  /**
2   * @brief Write the linked list to an output stream
3   * @param [in] file file pointer of an output stream
4   * @param [in] head pointer of a linked list
5   * @param pfprint fonction pointer for printing the data of a cell on an output stream
6   */
```

```c
void LL_save_list_toFile(FILE * file, cell_t * head,
            void (*pfprint) (FILE *, monom_t *))
{
    cell_t * curr = head;
    if (NULL != file) // verifie que FILEa ete correctement ouvert
    {
    while (NULL != curr) // parcours la liste : tq on est pas a la fin
    {
        (*pfprint)(file, &curr->val); // ecrit dans le fichier
        curr = curr->next; // passage au maillon suivant
    }
    }
}
```

5. Lexique
   (a) Paramètres
   - `FILE * file`: fichier ouvert.
   - `cell_t * head`: pointeur vers le début de la liste.
   - `void * pfprint (FILE* , monom_t* )`: pointeur de fonction qui affiche le monome dans le fichier.

   (b) Variables
   - `cell_t * curr`: adresse de la cellule courante. (parcours)
6. Hypothèse La fonction pfprint existe et elle est utilisable/fonctionelle.
7. Liste des cas $\text{LL}_{\text{savelisttoFile}}$ fonctionne quand :
   - la liste est vide
   - la liste est non vide
   - le fichier a précédement été ouvert

4.1.6 $\text{LL}_{\text{savelisttoFileName}}$

1. Description

   `La fonction écrit la liste dans un fichier pour la sauvegarder. Si le fichier n'a pas réussi à être ouvert (problème de nom, de droit), la fonction ne fait rien à part afficher un message d'erreur.`

2. Algorithme

Algorithm 6: Procédure de sauvegarde d'une liste dans un fichier à partir de son nom

Data: E: $pfwrite$, ES: $fname$, $head$

$m(curr) \leftarrow cm(head)$;
$m(file) \leftarrow ouvrirFichier(fname)$;
if $file \neq NIL$ **then**
  while $curr \neq NIL$ **do**
    $pfwrite(file, val(curr))$; ▷ Ecrit le monome dans le fichier
    $m(curr) \leftarrow next(curr)$;
else
  $afficherErreur()$;
$fermerFichier(file)$

3. Signature

```
void LL_save_list_toFileName(cell_t * head, char * fname, void (*pfwrite) (FILE *, monom_t *));
```

4. Code source

```
/**
 * @brief Save a linked list into a file
 * @param [in, out] head : head pointer of a linked list
 * @param [in] fname name of the backup file
 * @param pfwrite fonction pointer for writing the data of a cell to a output stream
 */
id LL_save_list_toFileName(cell_t * head, char * fname, void (*pfwrite) (FILE *, monom_t *))

  cell_t * curr = head;
  FILE * fp;

  fp = fopen(fname, "w+"); // ecrase le fichier s'il existe
  if (NULL != fp) // si on a reussi a ouvrir le fichier
  {

  while (NULL != curr) // parcours la liste : tq on est pas a la fin
  {
      (*pfwrite)(fp, &(curr->val)); // ecrit dans le fichier
      curr = curr->next; // passage au maillon suivant
  }
  fclose(fp); // ferme le fichier
  }
  else // message d'erreur si l'ouverture ne fonctionne pas
  {
  fprintf(stderr, "%s : 'Erreur lors de l'ouverture du fichier %s'",
      __FUNCTION__,
      fname);
  }
```

5. Lexique

(a) Paramètres

- `FILE * fname`: nom du fichier.

- `cell_t * head`: pointeur vers le début de la liste.
- `void * pfwrite (FILE* , monom_t* )`: pointeur de fonction qui affiche le monome dans le fichier.

(b) Variables

- `cell_t * curr`: adresse de la cellule courante. (parcours)
- `FILE * fp`: fichier de nom fname.

6. Hypothèse La fonction pfwrite existe et elle est utilisable/fonctionelle.
7. Liste des cas $LL_{savelisttoFileName}$ fonctionne quand :
   - la liste est vide
   - la liste est non vide
   - le fichier existe et à les droits d'écriture

4.1.7 $LL_{searchprev}$

1. Description

```
La fonction cherche l'adresse du maillon précédent une valeur dans
une liste triéée. La comparaison se fait par le pointeur d'une fonction
de comparaison qui renvoie:
- '=0' si les éléments sont égaux.
- '<0' si l'élément a est plus petit que b.
- '>0' si l'élément a est plus grand que b.
```

2. Algorithme

Algorithm 7: Fonction de rechercher d'un précédent d'une valeur

Data: E: $pfcmp$, ES: $a0$, $value$

$m(prec) \leftarrow a0$;

while $cm(prec) \neq NIL\ ET\ ALORS\ pfcmp(value, val(cm(prec))) > 0$ **do**

  $m(prec) \leftarrow adresse(next(cm(prec)))$;

return $prec$

3. Signature

```
cell_t ** LL_search_prev(cell_t ** a0, monom_t * value, int (*pfcmp) (monom_t *, monom_t *));
```

4. Code source

```
1  /**
2   * @brief Search a value in a linked list, and return the address of the previous pointer
3   * @param [in] a0 address of the head pointer
4   * @param [in] value address of the value to search
5   * @param pfcmp fonction pointer for comparison of two values
6   * @return the address of the previous pointer
7   */
8  ll_t ** LL_search_prev(cell_t ** a0, monom_t * value, int (*pfcmp) (monom_t *, monom_t *))
9
10   cell_t ** prec = a0; // maillon precedent
```

```
11    // tq prec!= NULL et que le maillon courant est pluspetit quecelui quel'on cherche
12    while (*prec != NULL && (*pfcmp)(value, &(*prec)->val) > 0)
13    {
14    prec = &((*prec)->next); // on avance d'un maillon
15    }
16    return prec; // retourne le precedent
```

5. Lexique
   (a) Paramètres
      - `cell_t ** a0`: pointeur vers le début de la liste.
      - `int * pfcmp (monom_t* , monom_t* )`: pointeur de fonction qui affiche le monome dans le fichier.
      - `monom_t* value`: monome qui représente la valeur a comparer.

   (b) Variables
      - `cell_t ** prec`: adresse du maillon précédent. (parcours)
6. Hypothèse La fonction pfcmp existe et elle est utilisable/fonctionelle. C'est la fonction pfcmp qui gère si le maillon est NULL.
7. Liste des cas $\text{LL}_{\text{searchprev}}$ fonctionne quand :
   - la liste est vide (renvoie NULL)
   - le maillon que l'on cherche est la tête de la liste
   - le maillon que l'on cherche est en fin de la liste
   - le maillon que l'on cherche est au milieu de la liste
   - le maillon que l'on cherche n'est pas dans la liste

4.1.8 $\text{LL}_{\text{delcell}}$

1. Description

   `Procédure de suppression d'une cellule.`

2. Algorithme

Algorithm 8: Procédure de suppression d'une cellule

Data: ES: *prec*

$m(tmp) \leftarrow cm(prec)$;
$m(prec) \leftarrow next(cm(prec))$;
$m(next(tmp)) \leftarrow NIL$;
$libérer(tmp)$;

3. Signature

```
   void LL_del_cell(cell_t ** prec);
```

4. Code source

```
1  /**
2   * @brief Delete a cell from a linked list
```

```
 * @param [in, out] prec address of the previous pointer of the cell to delete
 */
id LL_del_cell(cell_t ** prec)

  cell_t * tmp = *prec;
  (*prec) = (*prec)->next;
  (*tmp).val.coef = 0; // mets a 0 dans la RAM (nettoyage de memoire)
  (*tmp).val.degree = 0; // permet de rendre confidentiel les donnees
  (*tmp).next = NULL; // manipulees aprs fin du programme
              // limite le reverse engineering sur les poly :)

  free(tmp); // liberation du maillon
```

5. Lexique
   (a) Paramètres
   - `cell_t ** prec`: pointeur vers l'adresse du maillon à supprimer.

   (b) Variables
   - `cell_t * tmp`: sauvegarde du maillon à supprimer pour le libérer. (parcours)

6. Hypothèses Il faut avoir donné le précédent de la cellule à supprimer et que la cellule précédent et courant ne soit pas NULL.

4.1.9 $\mathbf{LL_{freelist}}$

1. Description

   `Procédure de libération d'une liste.`

2. Algorithme

Algorithm 9: Procédure de libération d'une liste

Data: ES: $a0$
$m(curr) \leftarrow cm(a0)$;
while $cm(curr) \neq NIL$ **do**
  $suppressionCellule(curr)$;
$cm(a0) \leftarrow NIL$

3. Signature

```
void LL_del_cell(cell_t ** prec);
```

4. Code source

```
/**
 * @brief Free the memory location occupied by a linked list
 * @param [in, out] a0 address of head pointer of a linked list
 */
id LL_free_list(cell_t ** a0)

  cell_t * curr = *a0; // maillon courant

```

```
9   while (curr != NULL) // parcours de la liste
10  {
11  LL_del_cell(&curr); // supprime le maillon en cours
12  }
13  (*a0) = NULL; // mets a0 a NULL car la liste est vide
```

5. Lexique
   (a) Paramètres
      - `cell_t ** a0`: pointeur vers la tête de la liste.
   (b) Variables
      - `cell_t * curr`: pointeur vers le maillon courant.
6. Liste des cas $\text{LL}_{\text{freelist}}$ fonctionne quand :
   - la liste est vide
   - la liste est non vide

4.1.10 $\text{LL}_{\text{multicationmonometolist}}$

1. Description

   `Fonction qui calcule la multiplication d'un polynôme avec un monôme et qui retourne le polynome résultant.`

2. Algorithme

Algorithm 10: Procédure de libération d'une liste

Data: E: $monome \neq NIL$, ES: $a0$
$m(polyRes) \leftarrow NIL$;
$m(precA0) \leftarrow a0$;
$m(precPolyRes) \leftarrow polyRes$;
while $cm(precA0) \neq NIL$ **do**
  ▷ multiplication du monomes et d'un maillon de a0
  $m(monomTmp) \leftarrow multiplicationMonome(val(cm(precA0)), monome)$;
  ▷ création du maillon résultant issu de la multiplication
  $m(cellTmp) \leftarrow creerCellule(monomTmp)$;
  ▷ ajout du maillon au polynome résultant
  $ajouterCellule(precPolyRes, cellTmp)$;
  ▷ Avancement dans le polynome a0 et le polynome résultant
  $m(precA0) \leftarrow next(cm(precA0))$;
  $m(precPolyRes) \leftarrow next(cm(precPolyRes))$;
return $polyRes$

3. Signature

```
cell_t * LL_multication_monome_to_list(monom_t * monome, cell_t ** a0)
```

4. Code source

```c
/**
 * @brief Multiply a list by a monome
 * @param [in, out] a0 address of head pointer of a linked list
 * @param [in, out] monome address of the monome used for multiplication
 * @return result of the multiplication between the list and the monome in a new list allocated in
     the function
 */
ll_t * LL_multication_monome_to_list(monom_t * monome, cell_t ** a0)

  cell_t * list_res = NULL; // liste contenant le resultat
  cell_t ** prec_a0 = a0; // precedent du maillon de la liste en parametre
  cell_t ** prec_list_res = &list_res; // precedent maillon en cours de la
                 // liste resultat
  monom_t monom_tmp; // variable temporaire pour plus de lisibilitee
  cell_t * cell_tmp; // variable temporaire pour plus de lisibilitee

  if (NULL != monome) // si le monome n'est pas NULL
        // car sinon a0 * NULL= NULL
  {
  while((*prec_a0) != NULL) // parcours a0
  {
      // multiplication entre le monome un maillon de a0
      monom_tmp = monom_multiplication(&(*prec_a0)->val, monome);

      // alloue la memoire pour creer le poly resultat
      cell_tmp = LL_create_cell(&monom_tmp);

      // ajoute le produit dans le poly resultat, respecte l'ordre
      // croissant, pas besoin de s'en occupe
      LL_add_cell(prec_list_res, cell_tmp);

      // avance le prec de a0 et prec de poly res
      prec_a0 = &(*prec_a0)->next;
      prec_list_res = &(*prec_list_res)->next;
  }
  }
  return list_res; // retourne la liste resultat, nouvelle liste
```

5. Lexique

   (a) Paramètres

   - `cell_t ** a0`: pointeur vers la tête de la liste.
   - `monom_t* monome`: pointeur sur le monome à multiplier à la liste.

   (b) Variables

   - `cell_t * poly_res`: adresse du polynome résultat.
   - `cell_t ** prec_a0`: adresse de maillon précédent de la liste a0 pour opérer la multiplication. (parcours)
   - `cell_t ** prec_poly_res`: adresse de maillon précédent de la liste poly_{res} pour opérer la multiplication. (parcours)
   - `monom_t monom_tmp`: monom résultant de la multiplication entre les monomes de a0 et monome en paramètre.
   - `cell_t * cell_tmp`: pointeur vers la cellule contenant le monome résultant.

6. Liste des case $\text{LL}_{\text{multicationmonometolist}}$ fonctionne quand :
   - quand a0 est NULL: renvoie une nouvelle liste NULL
   - quand le monôme est NULL : renvoie une nouvelle liste NULL
   - quand ni le monôme, ni la liste est NULL : renvoie la multiplication classique dans une nouvelle liste, allouée dans la fonction

4.2 valCell

4.2.1 $\text{monom}_{\text{degreecmp}}$

1. Description

```
La fonction compare le degré de 2 monomes a et b. Elle renvoie :
- >0 si a est plus grand que b
- <0 si a est plus petit que b
- =0 si a est égal à b ou au minimum un monôme est NULL
```

2. Algorithme

Algorithm 11: Fonction de création de cellule

Data: E: $m1, m2$

$res \leftarrow 0$;

if $m1 \neq NIL\ ET\ m2 \neq NIL$ **then**

  $m(res) \leftarrow (degree(m1) - degree(m2))$;

return *res;*

3. Signature

```
int monom_degree_cmp(monom_t * m1, monom_t * m2);
```

4. Code source

```
/**
 * @brief Compare the degree of two monomials
 * @param [in] m1 address of the first monomial
 * @param [in] m2 address of the second monomial
 * @return <0 if m1.degree<m2.degree; =0 if m1.degree=m2.degree; >0 if m1.degree>m2.degree
 * 0 if pointeur null
 */
t monom_degree_cmp(monom_t * m1, monom_t * m2)

  int res = 0; // 0 si au minimum un des monomes est NULL
  if(m1 != NULL && m2 != NULL) // si les deuxmonomes ne sont pas NULL
  {
  res = (m1->degree - m2->degree); // difference entre les deux degres
  }
  return res;
```

$\text{monom}_{\text{degreecmp}}$ fonctionne quand :

- les deux pointeurs de monômes sont différents de NULL

- si un pointeur de monôme vaut NULL : renvoie 0
- si les deux pointeurs de monôme sont NULL : renvoie 0

5. Lexique
   (a) Paramètres
   - `monom_t * m1`: adresse du premier monome à comparer.
   - `monom_t * m2`: adresse du deuxième monome à comparer.

   (b) Variables
   - `int res`: contient le résultat de la différence des degrées des monomes.
6. Liste des cas

4.2.2 monom$_{\text{save2file}}$

1. Description

   `La procédure écrit dans un fichier donné en paramètre le monome donné en paramètre avec la sérialisation suivante: "coef degree", avec une précision de 3 chiffres après la virgule pour le coeff.`

2. Algorithme

Algorithm 12: Procédure de sauvegarde d'un monome dans un fichier

Data: E: $monome$, ES: $file$

if $file \neq NIL$ ET $monome \neq NIL$ **then**
  $afficher(file, "\%.3f\%d", monome)$;

3. Signature

```
void monom_save2file(FILE * file, monom_t * monome);
```

4. Code source

```
/**
 * @brief write the information of a monomial to the given output stream
 * @param [in] file file pointer of an output stream
 * @param [in] m address of a monomial
 */
id monom_save2file(FILE * file, monom_t * monome)

  if (file != NULL && monome != NULL) // si le fichier est ouvert et le
                // pointeur n'est pas a NULL
  {
  fprintf(file, "%.3f %d\n", monome->coef, monome->degree); // ecrit dans file
  }
```

5. Lexique
   (a) Paramètres
   - `monom_t * monome`: adresse du monome à sauvegarder.

- `FILE * file`: fichier où écrire.

(b) Variables

- Aucune variable dans cette fonction.

6. Liste des cas $\text{monom}_{\text{save2file}}$ fonctionne quand :
   - le monome est NULL : ne fait rien
   - le pointeur de fichier pointe sur NULL: ne fait rien
   - le monome exite et le fichier est ouvert: ecrit le monome correctement

4.2.3 $\text{monom}_{\text{save2fileForPoly}}$

1. Description

   `La procédure écrit dans un fichier donné en paramètre le monome donné en paramètre avec la sérialisation suivante: "(coef, degree)", avec une précision de 2 chiffres après la virgule pour le coeff. Cette fonction sert pour les tests de polynomes.`

2. Algorithme

Algorithm 13: Procédure de sauvegarde d'un monome dans un fichier (polynome)

Data: E: $monome$, ES: $file$

if $file \neq NIL \ ET \ monome \neq NIL$ **then**
  $affiche r(file, "(\%.2f, \%d)", monome)$;

3. Signature

```
void monom_save2fileForPoly(FILE * file, monom_t * monome);
```

4. Code source

```
/**
 * @brief write the information of a monomial to the given output stream for
 * polynome test
 * @param [in] file file pointer of an output stream
 * @param [in] m address of a monomial
 */
id monom_save2fileForPoly(FILE * file, monom_t * monome)

 if (file != NULL && monome != NULL) // si le fichier est ouvert et le
                // pointeur n'est pas a NULL
 {
 fprintf(file, "(%.2f, %d) ", monome->coef, monome->degree); // ecrit dans file
 }
```

5. Lexique

(a) Paramètres

- `monom_t * monome`: adresse du monome à sauvegarder.
- `FILE * file`: fichier où écrire.

(b) Variables

- Aucune variable dans cette fonction.

6. Liste de cas
   $\text{monom}_{\text{save2fileForPoly}}$ fonctionne quand :
   - le monome est NULL : ne fait rien
   - le pointeur de fichier pointe sur NULL: ne fait rien
   - le monome exite et le fichier est ouvert: ecrit le monome correctement

4.2.4 $\text{monom}_{\text{multiplication}}$

1. Description

   La fonction retourne un nouveau monome résultant de la multiplication entre deux monomes donnés en paramètres. Pour cela, on additionne (resp. multiplie) leurs degrés (resp. coefficients).

2. Algorithme

Algorithm 14: Fonction de multiplication de deux polynomes

Data: $monome1, monome2$
$m(res) \leftarrow creerMonome()$;
$m(coef(res)) \leftarrow coef(monome1) * coef(monome2)$;
$m(degree(res)) \leftarrow degree(monome1) + degree(monome2)$;
return *res;*

3. Signature

```
monom_t monom_multiplication(monom_t * monome1, monom_t * monome2);
```

4. Code source

```
/**
 * @brief Multiply two monomes together. The first receives the result
 * @param [in, out] monome1 address of the monome which receives the
 * multiplication between itself and the monome2
 * @param [in, out] monome2 address of the second monome participating in the
 * multiplication
 * @return result of the multiplication between the 2 monomes passed in parameter
 */
nom_t monom_multiplication(monom_t * monome1, monom_t * monome2)

  monom_t res; // monome resultat
  res.coef = monome1->coef * monome2->coef; // multiplication des coef
  res.degree = monome1->degree + monome2->degree; // somme des degres
  return res;
```

5. Lexique

(a) Paramètres

- `monom_t * monome1`: adresse du premier monome à multiplier.

- `monom_t * monome1`: adresse du deuxième monome à multiplier.

(b) Variables

- `monom_t res` monome correspondant au résultat de la multiplication des deux monomes donnés en paramètres.

6. Hypothèse Les deux pointeurs de monôme ne pointent pas sur NULL.
7. Liste des cas $\text{monom}_{\text{multiplication}}$ fonctionne quand :
   - les pointeurs des deux monômes pointent sur des monômes existant sinon erreur segmentation (cf hypothèse)

4.3 polynomial

4.3.1 $\text{poly}_{\text{derive}}$

1. Description

```
La fonction dérive un polynôme représenté par une liste chaînée donnée en paramètre.
Le polynôme est supposé trié par ordre croissant de ses degrés.
Le polynôme est aussi supposé n'avoir aucun monome de même degrés.
Pour cela, on vérifie si la première cellule est de degré 0:
Si oui, on le supprime.
On parcourt les cellules du polynomes en effectuant les opérations suivantes:
- le coefficient prend le résultat de la multiplication entre le degré de la cellule
  et de son coefficient.
- le degree prend le degree du monome moins 1.
```

2. Algorithme

Algorithm 15: Procédure de dérivation d'un polynome

Data: E: *polynome*

$prec \leftarrow polynome$;
if $m1 \neq NIL\ ET\ m2 \neq NIL$ **then**
    $m(res) \leftarrow (degree(m1) - degree(m2))$;
while $cm(prec) \neq NIL$ **do**
    $m(coef(val(cm(prec)))) \leftarrow cm(coef(val(cm(prec)))) * cm(degree(val(cm(prec))))$;
    $m(degree(val(cm(prec)))) \leftarrow cm(degree(val(cm(prec)))) - 1$;
    $m(prec) \leftarrow adresse(next(cm(prec)))$;

3. Signature

```
void poly_derive(cell_t ** polynome);
```

4. Code source

```
/**
 * @brief compute 'in place' the derive of a polynomial
 * @param [in, out] xxx address of a polynomial's head pointer
 */
```

```
id poly_derive(cell_t ** polynome)
 hypothese : fonctione pour tous les polynomes tri par degr croissant

  cell_t ** prec = polynome; // parcours avec prec pour supprimer si deg == 0

  if ((*prec) != NULL && (*prec)->val.degree == 0) // deg maillonTete == 0
  // meilleure complexit que de testere dans le TQ a chaque fois
  // hyp: triee oredre croissant donc supprime tete si deg == 0
  {
  LL_del_cell(prec); // supprime tete
  }
  while ((*prec) != NULL) // parcours polynome avec prec
  {
  (*prec)->val.coef *= (*prec)->val.degree; // change coef (coef* deg)
  (*prec)->val.degree = (*prec)->val.degree - 1; // decremente deg
  prec = &((*prec)->next); // passage maillon suivant
  }
```

5. Lexique
   (a) Paramètres
      - `cell_t * polynome`: adresse de la première cellule du polynome.
   (b) Variables
      - `cell_t ** prec`: pointeur de pointeur sur l'adresse de la cellule précédente.
6. Hypothese Le polynôme est trié par ordre croissant.
7. Liste des cas $\text{poly}_{\text{derive}}$ foncitonne quand :
   - le polynome est vide
   - le polynome est quelconque et respecte l'hypothèse
   - le polynome contient un maillon de degré nul et qui respecte l'hypothèse : la derivé d'une constante provoque la suppression du maillon du polynôme

4.3.2 poly_{add}

1. Description

```
La fonction ajoute deux polynômes représentés par une liste chaînés donnés en paramètre.
Les polynôme sont supposés triés par ordre croissant de ses degrés.
Pour cela, on parcourt le polynôme 1 et le polynôme 2
avec des pointeurs de pointeurs précédents.
On compare le degre des cellules du polynôme 1 et du polynôme 2:
- Si le degré du monome du polynôme 1 est égale à celui du polynôme 2:
  - Si l'ajout des coefficients vaut zéro (monôme nul):
    - On supprime la cellule de polynôme 1 (aucun ajout)
  - Sinon:
    - On ajoute les coefficients des deux cellules dans celle du polynôme 1.
    - On avance à la cellule suivante du polynôme 1.
- Sinon:
  - Si le degré du monôme du polynôme 1 est supérieur à celui du polynôme 2:
    - On sauvegarde la cellule suivante du polynôme 2.
```

- On ajoute la cellule du polynôme 2 au polynôme 1.
- On reprend le parcours de P2 à partir de la cellule sauvegardé auparavant.

- Sinon:
- On avance dans le polynome 1.

A la fin du parcours, on regarde s'il reste des cellules dans le polynôme 2. Si oui, on redirige la tête du polynôme 2 sur la fin du polynôme 1 et on rend nul la tête du polynôme 2.

2. Algorithme

Algorithm 16: Procédure d'addition de deux polynomes

Data: E: $poly1$, $poly2$

```
1  m(precP1) ← poly1;
2  m(precP2) ← poly2;
3  while cm(precP1) ≠ NIL ET cm(precP2) ≠ NIL do
4      if cm(degree(val(cm(precP1)))) = (degree(val(cm(precP2)))) then
5          if abs(cm(coef(val(cm(precP1)))) + cm(coef(val(cm(precP2))))) then
6              supprimerCellule(precP2);
7          else
8              cm(coef(val(cm(precP1)))) + cm(coef(val(cm(precP2))));
9      else
10         if cm(degree(val(cm(precP1)))) > (degree(val(cm(precP2)))) then
11             m(tmp) ← adresse(next(cm(precP2)));
12             ajouterCellule(precP1, cm(precP2));
13             m(precP2) ← tmp;
14         else
15             m(precP1) ← adresse(next(cm(precP1)));
16 if cm(precP2) ≠ NIL then
17     m(cm(precP1)) ← cm(precP2);
18     m(cm(precP2)) ← NIL;
```

3. Signature

```
void poly_add(cell_t ** poly1, cell_t ** poly2);
```

4. Code source

```
/**
 * @brief compute P1 = P1 + P2, P2 become empty
 * @param poly1 [in, out] address of the 1st polynomial's head pointer
 * @param poly2 [in, out] address of the 2nd polynomial's head pointer
 */

id poly_add(cell_t ** poly1, cell_t ** poly2)
 hypothese : P1 et P2 tris par ordre croissant

  cell_t ** precP1 = poly1; // precedent poly1, avance avec prec pour
             // insertion et suppression
  cell_t ** precP2 = poly2; // precedent poly2, avance avec prec pour
```

```
13              // insertion et suppression
14              // currP1 = (*precP1)->val
15  cell_t * tmp = NULL; // var temporaire
16
17  while (NULL != (*precP1) && NULL != (*precP2))
18  // on parcours jusqu'a ce qu'on arrive au dernier maillon de P1 ou P2
19  {
20  if ((*precP1)->val.degree == (*precP2)->val.degree) //mme degre
21  {
22      if (fabs ((*precP1)->val.coef + (*precP2)->val.coef) // fabs => math.h
23                  < DBL_EPSILON) // <==> == 0 pour double
24      //si ajout provoque coef null, on supprime le currP1
25      {
26      LL_del_cell(precP1);
27      }
28      else // sinon on modifie la valeur du coef de p1
29      {
30      (*precP1)->val.coef += (*precP2)->val.coef; // somme coef
31      precP1 = &(*precP1)->next; // avance P1 d'un maillon
32      }
33      LL_del_cell(precP2); // libere la memoire le prec de P2 (+ avance P2)
34  }
35  else if (((*precP1)->val.degree > (*precP2)->val.degree))
36      // si deg currP1 > deg currP2,
37      // alors on insre le maillon currP2 entre precP1 et currP1
38  {
39      tmp = (*precP2)->next; // sauvegarde suivant P2
40      LL_add_cell(precP1, *precP2); // ajout du maillon
41      precP2 = &tmp;
42
43  }
44  else // sinon deg currP1 < deg currP2
45  {
46      precP1 = &(*precP1)->next; // alors on avance precP1  son suivant
47  }
48  }
49  if (NULL != (*precP2))
50   // cas o la dimension de P1 est infrieure  celle de P2
51   // il faut ajouter la fin du polynome de P2  P1
52  {
53  *precP1 = *precP2; // chainage fin P2 a fin P1
54  *precP2 = NULL; // P2 pointe sur NULL, plusde reference au maillon de P1
55  }
```

5. Lexique
   (a) Paramètres
      - `cell_t ** poly1`: adresse du premier polynome à additionner.
      - `cell_t ** poly2`: adresse du deuxième polynome à additionner.
   (b) Variables
      - `cell_t ** precP1`: pointeur de pointeur sur l'adresse de la cellule précédente.
      - `cell_t ** precP2`: pointeur de pointeur sur l'adresse de la cellule précédente.
      - `cell_t * tmp`: sauvegarde du suivant du polynome 2 lorsque que l'on ajoute la cellule du polynome 2 au polynome 1

6. Hypothèse Il faut que les deux polynômes soient triés par ordre de degré croissant.
7. Liste des cas Lorsque l'hypothèse de départ est respectée, poly_{add} fonctionne quand :
   - P1 est NULL, P1 devient P2, et P2 devient NULL
   - P2 NULL, ne fait rien
   - P1 + P2 de la même taille
   - P1 + P2 avec des tailles différentes
   - P1 + P2 provoquant des suppressions de monôme, exemple : m1 = $5x^2$ + m2 = $-5x^2$
   - s'il y a une insertion d'un monôme de P2, exemple : P1 = $1x^0$ + $2x^2$ et P2 =$1X^1$

4.3.3 $\text{poly}_{\text{prod}}$

1. Description

```
La fonction multiplie deux polynômes, représenté par une liste chaînée, donnés en paramètre.
Les polynômes sont supposés triés par ordre croissant de ses degrés.
Pour cela, on parcourt chaque cellule d'un polynôme, ici le polynôme 2.
Pour chaque cellule du polynome parcouru, on fait :
  - On crée un polynôme temporaire qui contient la multiplication entre le
monôme courant du polynôme 2 et le polynôme 1
  - On fait une addition entre le polynôme précédemment obtenue et le polynôme résultat
On renvoie le nouveau polynome résultat.
```

2. Algorithme

Algorithm 17: Procédure de multiplication de deux polynomes

Data: E: $poly1$, $poly2$

$m(currP2) \leftarrow cm(poly2)$;
$m(polyRes) \leftarrow NIL$;
while $cm(currP2) \neq NIL$ **do**
    $m(polyTmp) \leftarrow cm(multiplicationPolynomeParMonome(adresse(val(currP2)), poly1)$;
    $ajouterPolynomes(polyTmp, polyRes)$;
    $m(currP2) \leftarrow next(cm(currP2))$;
return *polyRes;*

3. Signature

```
cell_t * poly_prod(cell_t ** poly1, cell_t ** poly2)
```

4. Code source

```
1  /**
2   * @brief compute P1 * P2
3   * @param poly1 [in, out] head pointer of the 1st polynomial
4   * @param poly2 [in, out] head pointer of the 2nd polynomial
5   * @return P3 = P1*P2
6   */
7  cell_t * poly_prod(cell_t ** poly1, cell_t ** poly2)
8  {
9      cell_t * poly_tmp = NULL; // polynome temporaire
```

```
10      cell_t * poly_res = NULL; // polynome qui va contenir le resultat
11      cell_t * currP2 = (*poly2); // maillon courant de P2
12
13      while(currP2 != NULL) // pour chaque maillon de P2
14      {
15      // multication maillon de P2 avec P1, forme poly temporaire
16      poly_tmp = LL_multication_monome_to_list(&currP2->val, poly1);
17
18      // addition entre poly_tmp et pol_res, occurence 1 : poly_res <-poly_tmp + NULL
19      poly_add(&poly_res, &poly_tmp);
20
21      // avance currP2
22      currP2 = currP2->next;
23      }
24      // retourne P3, construit avec la somme des poly_tmp
25      return poly_res;
26  }
```

5. Lexique
   (a) Paramètres
      - `cell_t ** poly1`: adresse du premier polynome.
      - `cell_t ** poly2`: adresse du deuxième polynome.

   (b) Variables
      - `cell_t * poly_tmp`: pointeur vers le polynome temporaire.
      - `cell_t * poly_res`: pointeur vers le polynome temporaire.
      - `cell_t * currP2`: pointeur courante de poly2.
6. Hypothèses Le polynome 1 et le polynome 2 sont triés par ordre de degré croissant, sinon il y a un problème lors de l'addition.
7. Liste de cas Soit P3 = P1*P2.
   Lorsque l'hypothèse de départ est respectée, $\text{poly}_{\text{prod}}$ fonctionne quand :
   - si P1 est NULL, P3 vaut NULL
   - si P2 est NULL, P3 vaut NULL
   - si P1 et P2 sont NULL, P3 vaut NULL
   - si la taille de P1 est différente de P2, il n'y a pas de cas particulier contrairement à poly_{add}
   - P1 et P2 quelconque, mais qui respecte l'hypothèse

Présentation des tests

Dans cette partie, nous allons présenter brièvement l'ensemble des tests.

5.1 Tests dans $\text{linkedlist}_{\text{main}}$

Les tests présents dans $\textbf{linkedlist}_{\textbf{main.c}}$ permettent de tester les fonctions qui sont codées dans **linkedlist.c** et dans **valCell.c**.

Les premiers tests concernent les fonctions dans **valCell.c**, on y trouve :

- $\text{monom}_{\text{degreecmp}}$: vérifie que l'égalité et la notion d'ordre entre 2 maillons soit bien respectées
- $\text{monom}_{\text{save2file}}$: vérifie que le monôme soit correctement écrit dans le FILE passé en paramètre

Puis les tests suivant concernent les fonctions dans **linkedlist.c**, on y retrouve :

- $LL_{initlist}$: vérifie que lors de l'initialisation, la liste passée en paramètre pointe sur NULL
- $LL_{createcell}$: vérifie qu'il y a une bonne allocation lors de la création et que la cellule créée prend les valeurs du maillon passé en paramètre
- $LL_{addcell1}$: test l'insertion d'un monôme en tête de liste
- $LL_{addcell2}$ et $LL_{addcell3}$: test l'insertion de plusieurs monômes en tête de liste
- $LL_{createlistefromFileName0}$: vérifie la bonne récupération depuis un fichier
- $LL_{createlistefromFileName1}$: vérifie que la liste est vide quand le fichier n'existe pas
- $LL_{savelisttoFile0}$: vérifie que la sauvegarde se passe bien pour une liste quelconque
- $LL_{savelisttoFile1}$: vérifie qu'il n'y a pas de sauvegarde pour une liste NULL
- $LL_{searchprev1}$: vérifie que la fonction renvoie le bon précédent quand le maillon cherché se trouve au milieu de la liste
- $LL_{searchprev2}$: vérifie que la fonction renvoie le bon précédent quand le maillon cherché se trouve en tête de la liste
- $LL_{searchprev3}$: vérifie que la fonction renvoie le bon précédent quand le maillon cherché se trouve à la fin de la liste
- $LL_{delcell1}$: vérifie que la fonction supprime le bon maillon quand il se trouve au milieu de la liste
- $LL_{delcell2}$: vérifie que la fonction supprime le bon maillon quand il se trouve en tête de la liste
- $LL_{delcell3}$: vérifie que la fonction supprime le bon maillon quand il se trouve en fin de liste
- $LL_{freelist0}$: vérifie bonne libération sur une liste non vide
- $LL_{freelist1}$: vérifie que la libération sur une liste vide ne provoque pas d'erreur
- $LL_{savelisttoFileName0}$: vérifie que la sauvegarde d'une liste non vide fonctionne dans un fichier dont le nom est donné en paramètre
- $LL_{savelisttoFileName1}$: vérifie que la sauvegarde d'une liste vide ne fait rien sur un fichier dont le nom est donné en paramètre
- $LL_{multicationmonometolist0}$: vérifie que la multiplication entre un monôme NULL et une liste non NULL vaut une liste NULL
- $LL_{multicationmonometolist1}$: vérifie que la multiplication entre un monôme non NULL et une liste non NULL est correct
- $LL_{multicationmonometolist2}$: vérifie que la multiplication entre un monôme non NULL et une liste NULL vaut une liste NULL
- $LL_{multicationmonometolist3}$: vérifie que la multiplication entre un monôme NULL et une liste NULL vaut une liste NULL

Voici une photo prouvant qu'il n'y a pas de fuites de mémoires, ni d'erreur de mémoire dans le fichier **linkedlist$_{main.c}$** :

5.2 Tests dans polynome$_{main}$

Les tests présents dans **polynome$_{main.c}$** permettent de tester les fonctions qui sont codées dans **polynome.c**.

L'affichage des tests s'effectue grâce à la fonction $monom_{save2fileforpoly}$, qui respecte la sérialisation suivante : "(%2.f, %d)". Le fichier de test à changer lors de la dernière séance, nous n'avons pas effectué les changements car nous avions déjà fini l'ensemble des tests. De plus, l'affichage dépend de la fonction donnée en paramètre, donc non-pénalisant pour les tests de l'évaluation.

Dans le fichier tests, on retrouve :

```
--- teZZT REPORT ---
  0 test(s) failed
100 test(s) passed
[linkedList]==7429==
==7429== HEAP SUMMARY:
==7429==     in use at exit: 0 bytes in 0 blocks
==7429==   total heap usage: 177 allocs, 177 frees, 269,309 bytes allocated
==7429==
==7429== All heap blocks were freed -- no leaks are possible
==7429==
==7429== For lists of detected and suppressed errors, rerun with: -s
==7429== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Fig. 3: Test valgrind sur les listes chaînées

- $LL_{initlist}$: vérifie que lors de l'initialisation d'un polynôme, ce dernier pointe vers NULL
- $Poly_{derive1}$: vérifie la bonne dérivation avec un polynôme quelconque
- $Poly_{derive2}$: vérifie la bonne dérivation avec un polynôme possédant un monôme de degré null
- $Poly_{derive3}$: vérifie que la dérivation d'un polynôme NULL ne provoque pas d'erreur
- $Poly_{addition0}$: vérifie que l'addition est correcte entre deux polynômes avec le même nombre de maillons et de même degrés
- $Poly_{addition1}$: vérifie que l'addition est correcte entre deux polynômes quand l'addition provoque la suppression d'un monôme du milieu
- $Poly_{addition2}$: vérifie que l'addition est correcte entre deux polynômes quand l'addition provoque la suppression du monôme en tête
- $Poly_{addition3}$: vérifie que l'addition est correcte entre deux polynômes quand l'addition provoque la suppression du dernier monôme
- $Poly_{addition4}$: vérifie que l'addition est correcte entre deux polynômes de taille différente (dim P1 > dim P2)
- $Poly_{addition5}$: vérifie que l'addition est correcte entre deux polynômes de taille différente (dim P1 > dim P2)
- $Poly_{addition6}$: vérifie que l'addition est correcte lorsque des maillons de P2 doivent être insérés dans P1
- $Poly_{addition7}$: vérifie que l'addition est correcte lorsque la somme provoque le polynôme NULL [P1 + (-1*P1)]
- $Poly_{addition8}$: vérifie que l'addition vaut P2 lorsque P1 est NULL
- $Poly_{addition9}$: vérifie que l'addition vaut P1 lorsque P2 est NULL
- $Poly_{addition10}$: vérifie que l'addition vaut NULL lorsque P1 et P2 sont NULL
- $Poly_{produit0}$: vérifie que le produit entre deux polynômes quelconque est correcte
- $Poly_{produit1}$: vérifie que le produit entre P1 NULL et P2 quelconque donne un polynôme NULL
- $Poly_{produit2}$: vérifie que le produit entre P1 quelconque et P2 NULL donne un polynôme NULL
- $Poly_{produit3}$: vérifie que le produit entre P1 NULL et P2 NULL donne un polynôme NULL, ne provoque pas d'erreur

Voici une photo prouvant qu'il n'y a pas de fuites de mémoires, ni d'erreur de mémoire dans le fichier **polynome$_{main.c}$**

```
--- teZZT REPORT ---
  0 test(s) failed
119 test(s) passed
==9058==
==9058== HEAP SUMMARY:
==9058==     in use at exit: 0 bytes in 0 blocks
==9058==   total heap usage: 318 allocs, 318 frees, 431,084 bytes allocated
==9058==
==9058== All heap blocks were freed -- no leaks are possible
==9058==
==9058== For lists of detected and suppressed errors, rerun with: -s
==9058== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Fig. 4: Test valgrind sur les polynomes