PROJET
CALCULE POLINOMIAL

MAJIDI Mohammed
OULAHR AOUI Ayoub

16/01/ 2014

—

Structures Des Données Avancées

—

Prof M.Dargham

Project Description: Polynomial Calculator

This C program implements a polynomial calculator that allows users to perform various operations on polynomials. The program utilizes a hash table to store and manage polynomials associated with unique keys. Users can interact with the calculator through a command-line interface, executing commands to manipulate and analyze polynomials.

Supported Operations:

Assigning Polynomials:

LET <key> = <polynomial>: Assign a polynomial to a key.
SET <key> = <polynomial>: Same as LET.
Displaying Polynomials:

DISPLAY <key>: Display the polynomial assigned to a key.
Arithmetic Operations:

ADD <key1>, <key2>: Add two polynomials and store the result in key1.
SUB <key1>, <key2>: Subtract key2 from key1 and store the result in key1.
MUL <key1>, <key2>: Multiply two polynomials and store the result in key1.
...

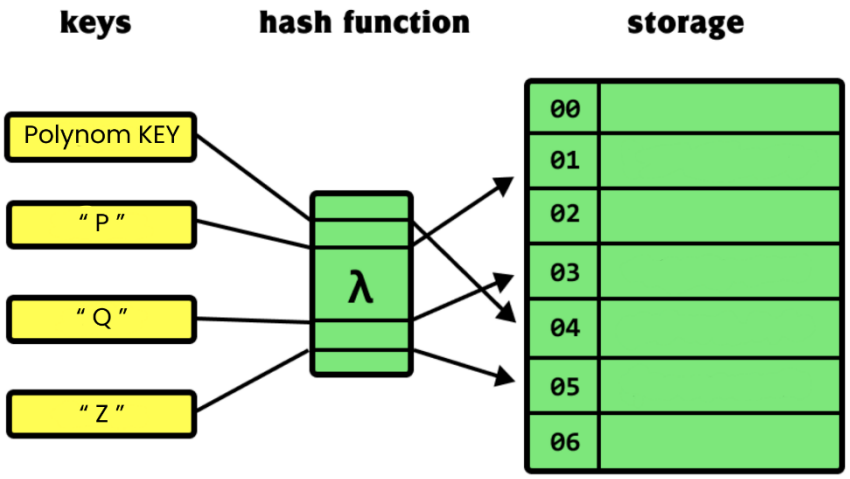
Help Command:

HELP: Display a list of available commands.
Exiting the Program:

EXIT: Exit the program.
Feel free to use the provided commands to perform operations on polynomials. If you need assistance, type HELP for a list of available commands. Happy polynomial calculations!

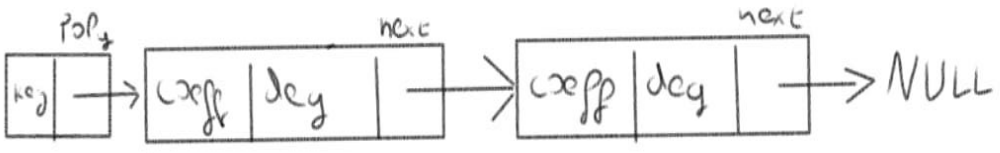
HASH TABLE

- KEY : Convertir en index dans le tableau de la HASH TABLE.
- ARRAY : composée de nœud qui contient le polynôme.



Chaque Nœud dans la HASH TABLE contient les éléments suivants :

- KEY : le nom du polynôme
- POLY : pointer tête vers 1er élément du polynôme.



La complexité temporelle de l'insertion et de l'accès (recherche) dans une table de hachage dépend en grande partie de la manière dont la fonction de hachage est implémentée et de la gestion des collisions. Voici une description générale de la complexité de ces opérations :

1. Insertion dans une Table de Hachage :

L'insertion dans une table de hachage consiste généralement à calculer la fonction de hachage de la clé, puis à insérer la valeur associée à cette clé à l'emplacement calculé. La complexité temporelle moyenne de l'insertion dans une table de hachage bien conçue est généralement en **O(1)**, ce qui signifie que le temps nécessaire pour insérer un élément est constant en moyenne.

2. Accès (Recherche) dans une Table de Hachage :

L'accès (recherche) dans une table de hachage consiste à calculer la fonction de hachage de la clé et à rechercher la valeur associée à cette clé à l'emplacement calculé. La complexité temporelle moyenne de l'accès est également en **O(1)** dans le cas idéal.

COMPLEXITÉ DES OPÉRATIONS SUR LES POLYNÔMES :

Opération LET :

Description :

La command LET exécute une fonction **void poly_table_set(hash_table_t *pt, char *id, char *name);** prend en entrée une chaîne de caractères input représentant un polynôme et le clé du polynôme et stock une structure de données poly_node_t qui représente le polynôme dans le HASH TABLE. Le polynôme est analysé terme par terme à partir de la chaîne d'entrée, et chaque terme est ajouté à la structure de données du polynôme en respectant l'ordre décroissant du degré de chaque monôme. La fonction prend en charge les coefficients, les exposants, et les signes positifs/négatifs pour chaque terme du polynôme.

Complexité Temporelle :

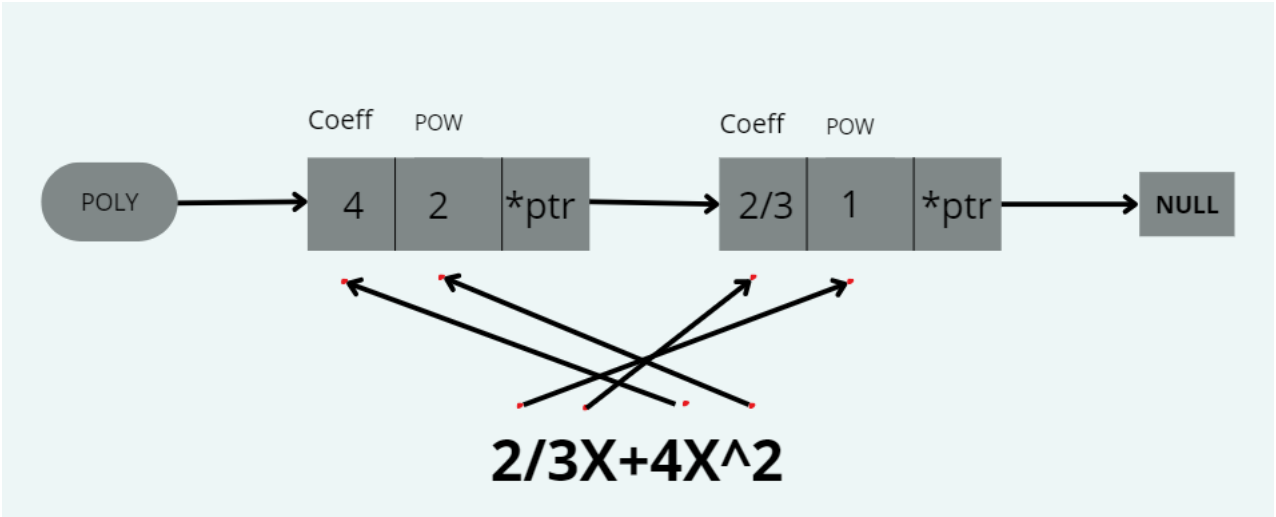
La complexité temporelle de cette fonction dépend principalement de la longueur de la chaîne d'entrée input. Analysons les principaux aspects :

Boucles While : Les deux boucles while qui parcourent la chaîne d'entrée et extraient les termes contribuent à une complexité linéaire par rapport à la longueur de la chaîne, soit **O(n)**, où n est la longueur de la chaîne.

Opérations à l'intérieur de la boucle principale : Les opérations effectuées pour chaque terme (allocation de mémoire, traitement des coefficients et des exposants) sont des opérations constantes par terme.

Appel de la fonction addTerm : Elle insère le terme dans le polynôme de manière ordonnée en fonction de l'exposant **(pow)** du terme. Si un terme avec le même exposant existe déjà dans le polynôme, les coefficients sont combinés. La complexité temporelle de la fonction addTerm est en général linéaire par rapport au nombre de termes déjà présents dans le polynôme, soit **O(n)**, où n est le nombre de termes dans le polynôme.

Schéma :



Opération SET :

Description :

La fonction **void poly_set(hash_table_t *pt, char *key, char *new_name);** est utilisée pour mettre à jour un polynôme associé à une clé spécifique dans une table de hachage **(hash_table_t).** Elle supprime l'ancien polynôme associé à la clé donnée et le remplace par un nouveau polynôme créé à partir de la chaîne de caractères **new_name**.

Complexité Temporelle :

La complexité temporelle de cette fonction peut être analysée comme suit :

Calcul de l'Index :

La fonction commence par calculer l'index associé à la clé fournie. Ce calcul a généralement une complexité constante, **O(1).**
Vérification de l'Existence du Polynôme :

La fonction vérifie si un polynôme existe déjà à cet index. Cette opération a une complexité constante, **O(1)**.
Suppression de l'Ancien Polynôme :

Si un polynôme existe à l'index spécifié, la fonction parcourt tous les termes de ce polynôme et les libère un par un. La complexité de cette opération dépend du nombre de termes dans l'ancien polynôme, soit O(m), où m est le nombre de termes dans l'ancien polynôme.
Création et Attribution du Nouveau Polynôme :

Un nouveau polynôme est créé à partir de la chaîne **new_name** à l'aide de la fonction **create_polynome**. La complexité de cette création dépend de la longueur de la chaîne **new_name**, soit **O(n)**, où n est la longueur de la chaîne.
Ensuite, le nouveau polynôme est attribué à la clé spécifiée. L'attribution a une complexité constante, **O(1)**.
Complexité Totale :
La complexité totale de la fonction **poly_set** est donc dominée par la suppression de l'ancien polynôme, soit **O(m)**, où m est le nombre de termes dans l'ancien polynôme.

```
Algorithme poly_set(pt: hash_table_t, key: char, new_name: char) :
    i ← poly_key_index(key, pt.size)    // Calcul de l'index associé à la clé

    Si pt.array[i] est NULL alors
        Afficher "No polynomial with the given name exists."
        Retourner
    Fin Si

    tmp ← pt.array[i].poly    // Récupération de l'ancien polynôme associé à la clé
    nouveau_poly ← create_polynome(new_name)    // Création du nouveau polynôme

    Tant que tmp n'est pas NULL faire
        prochain ← tmp.next
        Libérer tmp    // Libération du terme actuel de l'ancien polynôme
        tmp ← prochain
    Fin Tant que

    pt.array[i].poly ← nouveau_poly    // Attribution du nouveau polynôme à la clé
Fin Algorithme
```

Opération DISPLAY :

Description :

La fonction **poly_table_print** prend en entrée une table de hachage (pt) et une clé (key). Elle utilise la fonction **poly_key_index** pour calculer l'index associé à la clé dans la table de hachage, puis récupère le polynôme associé à cette clé. Enfin, elle imprime la clé suivie du polynôme associé en appelant la fonction **displayPolynomial**.

Complexité Temporelle :

La complexité temporelle de cette fonction peut être analysée comme suit :

Calcul de l'Index :

La fonction commence par calculer l'index associé à la clé fournie à l'aide de **poly_key_index**. Ce calcul a généralement une complexité constante, **O(1)**.

Récupération du Polynôme :

L'index est utilisé pour accéder au polynôme associé à la clé dans la table de hachage. Cela a une complexité constante, **O(1)**, car l'accès à un emplacement dans un tableau a une complexité constante.

Appel à displayPolynomial :

La complexité de displayPolynomial dépend du nombre de termes dans le polynôme. Si le nombre de termes est m, la complexité de l'affichage sera **O(m)**.

Complexité Totale :

La complexité totale de la fonction poly_table_print est dominée par la complexité de displayPolynomial, soit **O(m)**, où m est le nombre de termes dans le polynôme associé à la clé spécifiée.

```
Algorithme poly_table_print(pt: hash_table_t, key: char) :
    i ← poly_key_index(key, pt.size)    // Calcul de l'index associé à la clé

    tmp ← pt.array[i].poly    // Récupération du polynôme associé à la clé

    Afficher key, " : "
    displayPolynomial(tmp)    // Appel à la fonction d'affichage du polynôme
Fin Algorithme
```

Opération ADD :

Description :

La fonction **add_poly** prend en entrée une table de hachage (pt) ainsi que deux clés (key1 et key2). Elle récupère les polynômes associés à ces clés dans la table de hachage, puis effectue l'addition des polynômes. Le résultat est un nouveau polynôme qui est créé dynamiquement et renvoyé.

Le processus d'addition consiste à parcourir les deux polynômes termes par termes en fusionnant les termes de même degré. La somme des coefficients est calculée à l'aide de la fonction **fract_sum** pour garantir la représentation sous forme de fraction.

Complexité Temporelle :

Calcul des Indices (**poly_key_index**) :

Chaque appel à **poly_key_index** a une complexité constante, **O(1)**.
Vérification de l'Existence des Polynômes :

La vérification de l'existence des polynômes est une opération constante, **O(1).**
Allocation de Mémoire pour le Résultat (**sum**) :

L'allocation mémoire pour le résultat est constante, **O(1).**
Boucle While (Parcours des Polynômes) :

La boucle parcourt les deux polynômes, terme par terme, avec un nombre d'itérations dépendant du nombre total de termes dans les polynômes.
Calcul des Fractions (**fract_sum**) :

Le calcul des fractions est une opération constante par itération.
Allocation de Mémoire pour les Termes du Résultat (**tmp->next**) :

L'allocation mémoire pour chaque terme du résultat dépend du nombre total de termes dans le résultat. Dans le pire cas, la complexité serait linéaire en fonction du nombre total de termes.
La complexité temporelle totale de la fonction **add_poly** est donc en **O(m + n),** où m et n sont le nombre de termes dans les polynômes associés aux clés key1 et key2.

Opération SUB :

Description - Soustraction de Polynômes (**sub_poly**) :

La fonction **sub_poly** prend en entrée une table de hachage (pt) ainsi que deux clés (key1 et key2). Elle récupère les polynômes associés à ces clés dans la table de hachage, puis effectue la soustraction des polynômes (**pt->array[i]->poly - pt->array[j]->poly**). Le résultat est un nouveau polynôme qui est créé dynamiquement et renvoyé.

Le processus de soustraction consiste à parcourir les deux polynômes termes par termes. Si les termes de même degré sont présents dans les deux polynômes, leurs coefficients sont soustraits. Si un terme est présent dans un polynôme mais pas dans l'autre, il est simplement ajouté au résultat avec un changement de signe si nécessaire.

Complexité Temporelle - Soustraction de Polynômes (**sub_poly**) :

La complexité temporelle de cette fonction dépend principalement du nombre total de termes dans les deux polynômes en entrée. Soit m et n les nombres de termes dans les polynômes associés aux clés key1 et key2.

La création du polynôme résultant (**sub**) implique une allocation de mémoire pour chaque terme, et cela peut être linéaire par rapport au nombre total de termes.

La boucle While parcourt les deux polynômes, terme par terme, avec des opérations constantes à chaque itération. La complexité totale de cette boucle dépend du nombre total de termes dans les polynômes.

L'utilisation des fonctions **fract_sub** et **malloc** peut également contribuer à la complexité, mais dans la plupart des cas, elles sont considérées comme constantes.

En conclusion, la complexité temporelle totale de la fonction **sub_poly** est en **O(m + n)**, où m et n sont le nombre de termes dans les polynômes associés aux clés key1 et key2.

Opérations MUL & POW :

Description - Multiplication de Polynômes (multiplyPolynomials) :

La fonction multiplyPolynomials prend en entrée deux polynômes (poly1 et poly2) représentés par des listes chaînées de termes. Elle retourne un nouveau polynôme résultant de la multiplication des deux polynômes.

Le processus de multiplication consiste à parcourir tous les termes de poly1 et poly2 et à créer un nouveau terme pour chaque paire de termes correspondants. Les nouveaux termes sont ensuite ajoutés au résultat en utilisant la fonction addTerm pour maintenir l'ordre décroissant des termes en fonction de leurs puissances.

Complexité Temporelle - Multiplication de Polynômes (multiplyPolynomials) :

La complexité temporelle de cette fonction dépend du nombre total de termes dans les deux polynômes. Soit m et n les nombres de termes dans poly1 et poly2.

La fonction utilise deux boucles imbriquées pour parcourir tous les termes de poly1 et poly2. La complexité de ces boucles est O(m * n), où m et n sont les nombres de termes dans les polynômes.

Pour chaque paire de termes, un nouveau terme est créé et ajouté au résultat en utilisant la fonction addTerm. La complexité de addTerm dépend de la position du terme dans la liste résultante, mais elle peut être considérée comme linéaire en fonction du nombre total de termes dans le résultat, dans le pire des cas.

L'allocation mémoire pour chaque nouveau terme a une complexité constante par itération, car elle ne dépend pas du nombre total de termes dans les polynômes.

En conclusion, la complexité temporelle totale de la fonction multiplyPolynomials est en O(m * n), où m et n sont les nombres de termes dans poly1 et poly2.

Description - Puissance d'un Polynôme (pow_poly) :

La fonction pow_poly utilise la fonction multiplyPolynomials pour élever un polynôme à une puissance spécifiée. Elle parcourt la boucle for pour effectuer la multiplication du polynôme initial (tmp) avec lui-même (pt->array[i]->poly) pw - 1 fois.

Complexité Temporelle - Puissance d'un Polynôme (pow_poly) :

La complexité temporelle de la fonction pow_poly dépend principalement de la valeur de l'exposant pw. Si pw est petit, la complexité totale peut être considérée comme linéaire par rapport à pw, car la multiplication de polynômes est effectuée pw - 1 fois.

La multiplication de polynômes est effectuée par la fonction multiplyPolynomials, dont la complexité est en O(m * n), où m est le nombre de termes dans le polynôme initial (tmp) et n est le nombre de termes dans le polynôme (pt->array[i]->poly) à multiplier.
En résumé, la complexité temporelle totale de la fonction pow_poly est principalement influencée par la valeur de l'exposant pw et peut être approximée à O(pw * m * n), où m est le nombre de termes dans le polynôme initial (tmp) et n est le nombre de termes dans le polynôme à multiplier (pt->array[i]->poly).

Opération AFFECT :

Description - Affectation de Polynômes (aff_poly) :

La fonction aff_poly prend en entrée une table de hachage (pt) ainsi que deux clés (key1 et key2). Elle récupère le polynôme associé à la clé key1 dans la table de hachage. Ensuite, elle crée une nouvelle entrée dans la table de hachage avec la clé key2 et le polynôme associé à key1.

Le processus d'affectation consiste à copier chaque terme du polynôme associé à key1 dans un nouveau polynôme, puis à créer une nouvelle entrée dans la table de hachage avec la clé key2 et le polynôme copié.

Complexité Temporelle - Affectation de Polynômes (aff_poly) :

La complexité temporelle de cette fonction dépend principalement du nombre de termes dans le polynôme associé à key1. Soit m le nombre de termes dans ce polynôme.

La copie des termes du polynôme associé à key1 dans un nouveau polynôme (cpy) implique une boucle while qui parcourt tous les termes du polynôme. La complexité de cette boucle est en O(m), où m est le nombre de termes.

Pour chaque terme copié, un nouveau terme est alloué dynamiquement. L'allocation mémoire pour chaque terme a une complexité constante par itération de la boucle.

L'allocation mémoire pour la nouvelle entrée dans la table de hachage (new_node) a une complexité constante.

En conclusion, la complexité temporelle totale de la fonction aff_poly est en O(m), où m est le nombre de termes dans le polynôme associé à key1.

Opérations DER & INT :

Description - Dérivation de Polynôme (der_poly) :

La fonction der_poly prend en entrée une table de hachage (pt) ainsi qu'une clé (key) correspondant à un polynôme. Elle récupère le polynôme associé à la clé dans la table de hachage, puis calcule la dérivée de ce polynôme. La dérivée est stockée dans un nouveau polynôme, qui est ensuite retourné.

Le processus de dérivation consiste à parcourir tous les termes du polynôme initial, calculer la dérivée de chaque terme et les copier dans un nouveau polynôme.

Complexité Temporelle - Dérivation de Polynôme (der_poly) :

La complexité temporelle de cette fonction dépend principalement du nombre de termes dans le polynôme initial associé à la clé key. Soit m le nombre de termes dans ce polynôme.

La boucle while parcourt tous les termes du polynôme initial. La complexité de cette boucle est en O(m), où m est le nombre de termes.

Pour chaque terme, un nouveau terme est créé dans le polynôme dérivé. L'allocation mémoire pour chaque nouveau terme a une complexité constante par itération de la boucle.

L'appel à la fonction simplifyFraction a une complexité constante par itération de la boucle.

L'allocation mémoire pour le polynôme dérivé (der) a une complexité constante.

En conclusion, la complexité temporelle totale de la fonction der_poly est en O(m), où m est le nombre de termes dans le polynôme initial associé à la clé key.

Description - Intégration de Polynôme (inter_poly) :

La fonction inter_poly prend en entrée une table de hachage (pt) ainsi qu'une clé (key) correspondant à un polynôme. Elle récupère le polynôme associé à la clé dans la table de hachage, puis calcule l'intégrale de ce polynôme. L'intégrale est stockée dans un nouveau polynôme, qui est ensuite retourné.

Le processus d'intégration consiste à parcourir tous les termes du polynôme initial, calculer l'intégrale de chaque terme et les copier dans un nouveau polynôme.

Complexité Temporelle - Intégration de Polynôme (inter_poly) :

La complexité temporelle de cette fonction dépend principalement du nombre de termes dans le polynôme initial associé à la clé key. Soit m le nombre de termes dans ce polynôme.

La boucle while parcourt tous les termes du polynôme initial. La complexité de cette boucle est en O(m), où m est le nombre de termes.

Pour chaque terme, un nouveau terme est créé dans le polynôme intégré. L'allocation mémoire pour chaque nouveau terme a une complexité constante par itération de la boucle.

L'appel à la fonction simplifyFraction a une complexité constante par itération de la boucle.

L'allocation mémoire pour le polynôme intégré (inter) a une complexité constante.

En conclusion, la complexité temporelle totale de la fonction inter_poly est en O(m), où m est le nombre de termes dans le polynôme initial associé à la clé key.

Opération EVAL :

Description - Évaluation de Polynôme (eval_poly) :

La fonction eval_poly prend en entrée une table de hachage (pt), une clé (key) correspondant à un polynôme, et une valeur x. Elle récupère le polynôme associé à la clé dans la table de hachage, puis évalue ce polynôme pour la valeur donnée x. Le résultat de l'évaluation est retourné.

Le processus d'évaluation consiste à parcourir tous les termes du polynôme, calculer la valeur de chaque terme pour la valeur x, puis sommer ces valeurs.

Complexité Temporelle - Évaluation de Polynôme (eval_poly) :

La complexité temporelle de cette fonction dépend principalement du nombre de termes dans le polynôme associé à la clé key. Soit m le nombre de termes dans ce polynôme.

La boucle while parcourt tous les termes du polynôme. La complexité de cette boucle est en O(m), où m est le nombre de termes.

Pour chaque terme, la fonction power est appelée pour calculer la puissance de x. La complexité de cette fonction dépend de la mise en œuvre spécifique de la fonction power. Si elle utilise une méthode d'exponentiation rapide, la complexité est généralement en O(log n), où n est l'exposant.

Les opérations d'addition, multiplication et division dans le calcul de la somme ont une complexité constante par itération de la boucle.

En conclusion, la complexité temporelle totale de la fonction eval_poly est dominée par la boucle while et est en O(m), où m est le nombre de termes dans le polynôme initial associé à la clé key. La complexité dépend également de la complexité de la fonction power.