INTRODUCTION TO DATA STRUCTURES

Use linked lists to construct polynomials and perform addition and subtraction in C

HW3_D84099084_賴澍雨

Shih Yu Lai

1.Evaluating postfix expressions

The C program is designed to perform operations on polynomials using a linked list data structure.

```
PS D:\Data Structure\HW3_D840990B4> cd "d:\Data Structure\HW3_D840990B4\" ; if ($?) { gcc main.c -o main } ; if ($?) { .\main }
File: test1.txt
A:
(3)X^90 + (2)X^5 + (10)X^0
B:
(5)X^40 + (3)X^20 + (6)X^3 + (10)X^0
A+B:
(3)X^90 + (5)X^40 + (3)X^20 + (2)X^5 + (6)X^3 + (20)X^0
A-B:
(3)X^90 + (-5)X^40 + (-3)X^20 + (2)X^5 + (-6)X^3
File: test2.txt
A:
(10)X^100 + (5)X^50 + (3)X^1 + (99)X^0
B:
(10)X^100 + (5)X^50 + (3)X^1 + (99)X^0
A+B:
(20)X^100 + (10)X^50 + (6)X^1 + (198)X^0
A-B:
0
File: test3.txt
A:
(9)X^10 + (89)X^4 + (85)X^2 + (10)X^0
B:
(10)X^11 + (10)X^6 + (89)X^4 + (-5)X^2
A+B:
(10)X^11 + (9)X^10 + (10)X^6 + (178)X^4 + (80)X^2 + (10)X^0
A-B:
(-10)X^11 + (9)X^10 + (-10)X^6 + (90)X^2 + (10)X^0
File: test4.txt
A:
(98)X^10 + (98)X^5 + (97)X^1 + (96)X^0
B:
(98)X^10 + (97)X^5 + (96)X^1 + (95)X^0
A+B:
(197)X^10 + (195)X^5 + (193)X^1 + (191)X^0
A-B:
(1)X^10 + (1)X^5 + (1)X^1 + (1)X^0
File: test5.txt
A:
(-1)X^50 + (-3)X^25 + (-6)X^0
B:
(-10)X^51 + (-3)X^25 + (-14)X^0
A+B:
(-10)X^51 + (-1)X^50 + (-6)X^25 + (-20)X^0
A-B:
(10)X^51 + (-1)X^50 + (8)X^0
```

Output

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

// the data structure of a polynomial node
typedef struct Node{
    int coef;
    int expo;
    struct Node *next;
}Node;

// helper function to print out the polynomial
// input polynomial linked list head
void print_poly(Node *head){
    printf("\n");
    if(head){
        while(head->next){
            printf("(%d)x^%d + ",head->coef,head->expo);
            head = head->next;
        }
        printf("(%d)x^%d",head->coef,head->expo);
    }
    else
        printf("0");
    printf("\n");
}

// create a new polynomial node and return the pointer point to the node
Node* new_node(int coef, int expo){
    Node* n = (Node*)malloc(sizeof(Node));
    n->coef = coef;
    n->expo = expo;
    n->next = NULL;
    return n;
}

// construct polynomial and return the pointer point to the head of polynomial
// you can customize the input of this function
Node* sorted_insert(Node *head, Node *node){
    if (!head || node->expo > head->expo){
        node->next = head;
        return node;
    }
    Node* current = head;
    while (current->next && current->next->expo > node->expo){
        current = current->next;
    }
    node->next = current->next;
    current->next = node;
    return head;
}

void construct_poly(const char *filepath, Node **A_head, Node **B_head) {
    FILE *file = fopen(filepath, "r");
    if (file == NULL) {
        perror("Error opening file");
        return;
    }

    int coef, expo;
    int is_first_poly = 1; // Flag to indicate whether it's the first polynomial
    while (fscanf(file, "%d %d", &coef, &expo) == 2) {
        if (is_first_poly) {
            *A_head = sorted_insert(*A_head, new_node(coef, expo));
        } else {
            *B_head = sorted_insert(*B_head, new_node(coef, expo));
        }
        // Check if we have read the first polynomial completely
        if (is_first_poly && fgetc(file) == '\n') {
            is_first_poly = 0; // Move to the second polynomial
        }
    }

    fclose(file);
}
```

Node Creation

Sorted Insert

Polynomial Construction

```c
// A + B and return the pointer point to the head of result
// input are two polynomial linked list heads
Node* plus(Node *A_head, Node* B_head){
    Node *result_head = NULL, *current, *temp;
    while (A_head && B_head){
        if (A_head->expo > B_head->expo){
            result_head = sorted_insert(result_head, new_node(A_head->coef, A_head->expo));
            A_head = A_head->next;
        } else if (A_head->expo < B_head->expo){
            result_head = sorted_insert(result_head, new_node(B_head->coef, B_head->expo));
            B_head = B_head->next;
        } else {
            int sum_coef = A_head->coef + B_head->coef;
            if (sum_coef != 0){
                result_head = sorted_insert(result_head, new_node(sum_coef, A_head->expo));
            }
            A_head = A_head->next;
            B_head = B_head->next;
        }
    }

    // if there are remaining terms in A or B, add them to the result
    while (A_head) {
        result_head = sorted_insert(result_head, new_node(A_head->coef, A_head->expo));
        A_head = A_head->next;
    }
    while (B_head) {
        result_head = sorted_insert(result_head, new_node(B_head->coef, B_head->expo));
        B_head = B_head->next;
    }
    return result_head;
}
```

Addition

```c
// A - B and return the pointer point to the head of result
// input are two polynomial linked list heads
Node* minus(Node *A_head, Node* B_head){
    Node *result_head = NULL;
    while (A_head && B_head){
        if (A_head->expo > B_head->expo){
            result_head = sorted_insert(result_head, new_node(A_head->coef, A_head->expo));
            A_head = A_head->next;
        } else if (A_head->expo < B_head->expo){
            result_head = sorted_insert(result_head, new_node(-B_head->coef, B_head->expo));
            B_head = B_head->next;
        } else {
            int diff_coef = A_head->coef - B_head->coef;
            if (diff_coef != 0){
                result_head = sorted_insert(result_head, new_node(diff_coef, A_head->expo));
            }
            A_head = A_head->next;
            B_head = B_head->next;
        }
    }

    // if there are remaining terms in A or B, add them to the result
    while (A_head) {
        result_head = sorted_insert(result_head, new_node(A_head->coef, A_head->expo));
        A_head = A_head->next;
    }
    while (B_head) {
        result_head = sorted_insert(result_head, new_node(-B_head->coef, B_head->expo));
        B_head = B_head->next;
    }
    return result_head;
}
```

Subtraction

```c
// Function to deallocate memory of the polynomial linked list
void free_poly(Node *head) {
    Node *temp;
    while (head) {
        temp = head;
        head = head->next;
        free(temp);
    }
}
```

Memory Management

```c
int main(){
    char filepath[100];
    Node *A_head = NULL, *B_head = NULL;
    Node *sum = NULL, *diff = NULL;

    // Loop to process each of the five text files
    for (int i = 1; i <= 5; i++) {
        // Construct the file path for the current file
        sprintf(filepath, "D:\\Data Structure\\HW3_D840990B4\\Test\\test%d.txt", i);

        // Read and construct the two given polynomials from the current file
        construct_poly(filepath, &A_head, &B_head);

        // Perform A+B and A-B
        sum = plus(A_head, B_head);
        diff = minus(A_head, B_head);

        // Print results for the current file
        printf("\nFile: test%d.txt\n", i);
        printf("A:");
        print_poly(A_head);
        printf("B:");
        print_poly(B_head);
        printf("A+B:");
        print_poly(sum);
        printf("A-B:");
        print_poly(diff);

        // Free memory for the current file's polynomials
        free_poly(A_head);
        free_poly(B_head);
        free_poly(sum);
        free_poly(diff);

        // Reset head pointers for the next iteration
        A_head = NULL;
        B_head = NULL;
        sum = NULL;
        diff = NULL;
    }

    return 0;
}
```

Implementation

Data Structures

- Node: This structure represents a single term of a polynomial, containing coefficients (`coef`), exponents (`expo`), and a pointer (`next`) to the next term in the list. This allows polynomials of arbitrary length and non-zero terms to be dynamically constructed and manipulated.
- Linked List: The linked list is used to store the polynomial in a sorted order based on exponents in descending order. This organization facilitates polynomial operations.

Testing and Test Data Design

The correct operation of this program depends on carefully designed test cases that reflect various scenarios:

1. **Simple Cases:**
   - Polynomials like 1x^2 + 2x^1 and 2x^2 - 3x^1 to check basic arithmetic handling.
2. **Zero Coefficients:**
   - Including terms like 0x^n to ensure they are handled and ignored appropriately.
3. **Boundary Conditions:**
   - Polynomials where one polynomial is significantly longer than the other, or where exponents have large gaps.
4. **Complex Cases:**
   - Multiple terms with the same exponent, requiring correct summing or subtracting of coefficients.
5. **File Handling:**
   - The program expects a specific file format. Misformatted files or incorrect paths should be gracefully handled.

Program Logic and Algorithms

1. **Node Creation (`new_node`):**
   - A utility function that allocates memory for a new `Node`, initializes it with given coefficients and exponents, and sets the `next` pointer to `NULL`.
2. **Sorted Insert (`sorted_insert`):**
   - Inserts a new node into the linked list while maintaining the order of exponents in descending order. It checks if the new node should be inserted at the beginning or finds the correct position between existing nodes.
3. **Polynomial Construction (`construct_poly`):**
   - Reads coefficients and exponents from a file to construct two separate polynomials, distinguishing between them by a newline character in the file. This function utilizes sorted_insert for inserting each read term.
4. **Addition and Subtraction (`plus`, `minus`):**
   - Both functions iterate through two polynomial linked lists, comparing exponents to decide whether to add/subtract coefficients or simply transfer a node to the result list. For addition, coefficients are added, and for subtraction, they are subtracted. Nodes with zero coefficients are omitted.
5. **Memory Management (`free_poly`):**
   - After operations are complete, this function iterates through each polynomial's linked list to free the allocated memory, ensuring no memory leaks.