MIT World Peace University
Advanced Data Structures

Assignment 1

Naman Soni Roll No. 10

Contents

1 Problem Statement

Implement polynomial operations using Circular Linked List: Create, Display, Addition and Evaluation.

2 Objective

1. To study data structure: Circular Linked List.
2. To study differnet operations that can be performed on Circular Linked List.
3. To study applications of Circular Linked List.

3 Theory

3.1 *Circular Linked List*

A circular linked list is a variation of a linked list where the last element in the list points back to the first element, creating a loop. This means that there is no null element at the end of the list, and traversing the list will continue indefinitely. In a circular linked list, there is no concept of "head" and "tail" as the first and last element are connected. This data structure can be useful in certain algorithms or applications where traversing the list in a circular fashion is useful.

3.2 *Diffrence between SLL, CLL and DLL*

SLL (Single Linked List) is a linked list where each element, or node, in the list contains a reference to the next element in the list, but not the previous one. This means that you can traverse the list in one direction, typically from the head (first element) to the tail (last element).

CLL (Circular Linked List) is a variation of a linked list where the last element in the list points back to the first element, creating a loop. This means that there is no null element at the end of the list, and traversing the list will continue indefinitely. In a circular linked list, there is no concept of "head" and "tail" as the first and last element are connected.

DLL (Double Linked List) is a linked list where each element, or node, in the list contains a reference to both the next and previous element in the list. This means that you can traverse the list in both directions, from the head to the tail or from the tail to the head.

In short, SLL is a one-way linked list, CLL is a circular linked list and DLL is a two-way linked list.

3.3 *Various Operations on CLL*

There are several operations that can be performed on a circular linked list:

- Insertion: This operation allows you to add new elements to the list. In a circular linked list, elements can be inserted at the beginning, at the end, or at a specific position in the list.
- Deletion: This operation allows you to remove elements from the list. In a circular linked list, elements can be deleted from the beginning, from the end, or from a specific position in the list.
- Traversal: This operation allows you to iterate through the elements of the list. In a circular linked list, traversal can be done in a circular fashion, starting from the head and going to the tail, and then back to the head again.
- Search: This operation allows you to search for a specific element in the list. In a circular linked list, the search can be done starting from the head and going to the tail, and then back to the head again.
- Reversal: This operation allows you to reverse the order of elements in the list.
- Sorting: This operation allows you to sort the elements of the list in a specific order.

- Length: This operation allows you to find the number of elements in the list.
- Display: This operation allows you to display the elements of the list.
- Concatenation: This operation allows you to join two different circular linked list.
- Splitting: This operation allows you to divide the circular linked list into two different list.

4 Implementation

4.1 *Platform*

Mac OS 64x
Visual Studio Code

4.2 *Input and Output*

TESTCASE NO	**INPUT**	**OUTPUT**
01	Row 1, Cell 2	Row 1, Cell 3
Row 2, Cell 1	Row 2, Cell 2	Row 2, Cell 3

4.3 *Test Conditions*

- Input atleast 5 nodes.
- Addition of two polynomials with atleast 5 terms.
- Evalute polynomial with floating values.

4.4 *Psuedo Code*

```c
/*Problem Statement: Implement polynomial operations using Circular Linked List: Create,
    Display, Addition and
Evaluation
Name: Naman Soni
Roll No. 10
Batch A1
*/

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

struct node
{
    int coeff;
    int exp;
    struct node *next;
};

void add_data(struct node *head)
{
    int choice = 1;
    struct node *temp = head;

    do
    {

        struct node *curr = (struct node *)malloc(sizeof(struct node));

        printf("\nEnter coefficient:\n");
        scanf("%d", &curr->coeff);
        printf("\nEnter exponent:\n");
        scanf("%d", &curr->exp);
        curr->next = head;
        temp->next = curr;
        temp = temp->next;
        printf("\nDo you want to enter more terms?\nEnter 1 for yes and 0 for no\n");
        scanf("%d", &choice);
    } while (choice != 0);
}

void display(struct node *head)
{
    if (head->next == head)
    {
        printf("\nNo data available");
    }
    struct node *curr = (struct node *)malloc(sizeof(struct node));
    curr = head->next;
    while (curr != head)
    {
        printf("%dx^%d", curr->coeff, curr->exp);
        curr = curr->next;
        if (curr != head)
        {
            printf("+");
        }
    }
    printf("\n");
}

struct node *add_polynomials(struct node *head1, struct node *head2)
{
    // Pointers for the result polynomial.
    struct node *result_head = (struct node *)malloc(sizeof(struct node));
    result_head->next = result_head;
```

```
    struct node *result_temp = result_head;
    struct node *result_current;

    // p1 and p2 are the pointers to the first node of the two polynomials.
    struct node *p1 = head1->next;
    struct node *p2 = head2->next;

    // In case one of the polynomial exhausts before the other one.
    while (p1 != head1 && p2 != head2)
    {
        // if the exponents are equal, add the coefficients and add the node to the result polynomial.
        if (p1->exp == p2->exp)
        {
            // Copy the data of thesum of the nodes to the result polynomial.
            result_current = (struct node *)malloc(sizeof(struct node));
            result_current->coeff = p1->coeff + p2->coeff;
            result_current->exp = p1->exp;
            result_current->next = result_head;
            result_temp->next = result_current;

            // Increment the result polynomial pointer, and other polynomial pointers.
            result_temp = result_temp->next;
            p1 = p1->next;
            p2 = p2->next;
        }

        // If the exponent of the first polynomial is greater than the second one, add the node to the result polynomial.
        else if (p1->exp > p2->exp)
        {
            result_current = (struct node *)malloc(sizeof(struct node));
            result_current->coeff = p1->coeff;
            result_current->exp = p1->exp;
            result_current->next = result_head;
            result_temp->next = result_current;

            // increment the result polynomial pointer, and p1
            result_temp = result_temp->next;
            p1 = p1->next;
        }

        // If the exponent of the second polynomial is greater than the first one, add the node to the result polynomial.
        else if (p2->exp > p1->exp)
        {
            result_current = (struct node *)malloc(sizeof(struct node));
            result_current->coeff = p2->coeff;
            result_current->exp = p2->exp;
            result_current->next = result_head;
            result_temp->next = result_current;

            // increment the result polynomial pointer, and p2
            result_temp = result_temp->next;
            p2 = p2->next;
        }
    }

    // Case when p2 exhausts before p1.
    if (p1 == head1 && p2 != head2)
    {
        result_temp->next = p2;

        // This loop is to make the last node of the result polynomial point to the head of the result polynomial.
        while (result_temp->next != head2)
        {
            result_temp = result_temp->next;
```

```c
        }
        result_temp->next = result_head;
    }

    // Case when p1 exhausts before p2.
    else if (p1 != head1 && p2 == head2)
    {
        result_temp->next = p1;
        while (result_temp->next != head1)
        {
            result_temp = result_temp->next;
        }
        result_temp->next = result_head;
    }

    // Case when both p1 and p2 exhaust.
    else if (p1 != head1 && p2 != head2)
    {
        result_temp->next = p1;
        while (result_temp != head1)
        {
            result_temp = result_temp->next;
        }
        result_temp->next = result_head;

        result_temp->next = p2;
        while (result_temp != head2)
        {
            result_temp = result_temp->next;
        }
        result_temp->next = result_head;
    }

    return result_head;
}

int main()
{
    int choice = 0;
    struct node *head = (struct node *)malloc(sizeof(struct node));
    struct node *head1 = (struct node *)malloc(sizeof(struct node));
    struct node *head2 = (struct node *)malloc(sizeof(struct node));
    struct node *added;

    printf("What you want to do:\n1.Insert Polynomial\n2.Addition of Two polynomials:\n");
    scanf("%d", &choice);

    switch (choice)
    {
    case 1:
        printf("Insert polynomial:\n");
        add_data(head);
        display(head);
        break;

    case 2:
        printf("Please enter the first polynomial:\n");
        add_data(head1);
        display(head1);
        printf("\nEnter second polynomial:\n");
        add_data(head2);
        display(head2);
        printf("Addition of Polynomials:");
        added = add_polynomials(head1, head2);
        display(added);
        break;
    default:
        printf("!Invalid!");
```

```
        break;
    }

    return 0;
}
```

Listing 1: Input Code

```
    cd "/Users/cyrus/Desktop/Sem-4/ADS/" && g++ Assignment1.cpp -o Assignment1 && "/Users/cyrus/Desktop/Sem-4/ADS/"Assignment1
    cyrus@Namans-MacBook-Air Sem-4 % cd "/Users/cyrus/Desktop/Sem-4/ADS/" && g++ Assignment1.cpp -o Assignment1 && "/Users/cyrus/Desktop/Sem-4/ADS/"Assignment1
    What you want to do:
    1.Insert Polynomial
    2.Addition of Two polynomials:
    1
    Insert polynomial:

    Enter coefficient:
    3

    Enter exponent:
    2

    Do you want to enter more terms?
    Enter 1 for yes and 0 for no
    1

    Enter coefficient:
    cyrus@Namans-MacBook-Air ADS % cd "/Users/cyrus/Desktop/Sem-4/ADS/" && g++ Assignment1.cpp -o Assignment1 && "/Users/cyrus/Desktop/Sem-4/ADS/"Assignment1
    What you want to do:
    1.Insert Polynomial
    2.Addition of Two polynomials:
    2
    Please enter the first polynomial:

    Enter coefficient:
    3

    Enter exponent:
    2

    Do you want to enter more terms?
    Enter 1 for yes and 0 for no
    1

    Enter coefficient:
    5

    Enter exponent:
    1

    Do you want to enter more terms?
    Enter 1 for yes and 0 for no
    1

    Enter coefficient:

    9

    Enter exponent:
    0

    Do you want to enter more terms?
    Enter 1 for yes and 0 for no
    0
    3x^2+5x^1+9x^0
```

```

Enter second polynomial:

Enter coefficient:
4

Enter exponent:
6

Do you want to enter more terms?
Enter 1 for yes and 0 for no
1

Enter coefficient:
8

Enter exponent:
0

Do you want to enter more terms?
Enter 1 for yes and 0 for no
0
4x^6+8x^0
Addition of Polynomials:4x^6+3x^2+5x^1+17x^0
cyrus@Namans-MacBook-Air ADS %
```

Listing 2: Output

5 Conclusion

Thus, implemented different operations on CLL.

6 FAQ

1. Write an ADT for CLL.
An Abstract Data Type (ADT) for a Circular Linked List (CLL) could include the following operations:

1. Initialize: This operation creates an empty CLL.
2. Insert: This operation adds a new node to the CLL. The new node can be inserted at the beginning, end, or at a specific position in the CLL.
3. Delete: This operation removes a node from the CLL. The node to be deleted can be specified by its position in the CLL.
4. Search: This operation searches for a specific node in the CLL. The search can be based on the data stored in the node or the position of the node in the CLL.
5. Traverse: This operation visits each node in the CLL in a specific order, such as in a clockwise or counter-clockwise direction.
6. Length: This operation returns the number of nodes in the CLL.
7. isEmpty: This operation checks whether the CLL is empty or not.

Here is an example of an ADT for a CLL in C language:

```c
// ADT for a Circular Linked List

struct Node {
  int data;
  struct Node* next;
}
```

```

struct CLL {
  int length;
  struct Node* head;
}

// Initialize an empty CLL
void init(struct CLL* cll) {
  cll->length = 0;
  cll->head = NULL;
}

// Insert a new node at the beginning of the CLL
void insert_at_beginning(struct CLL* cll, int data) {
  struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
  new_node->data = data;

  if (cll->head == NULL) {
    new_node->next = new_node;
    cll->head = new_node;
  }
  else {
    struct Node* temp = cll->head;
    while (temp->next != cll->head) {
      temp = temp->next;
    }
    temp->next = new_node;
    new_node->next = cll->head;
    cll->head = new_node;
  }
  cll->length++;
}

// Delete a node from the CLL
void delete_node(struct CLL* cll, int position) {
  if (cll->head == NULL) {
    printf
```

Listing 3: EXAMPLE

2. How to perform multiplication of two polynomials?
Multiplying two polynomials is a process of combining like terms. The process of multiplying two polynomials is similar to the process of multiplying two numbers, with the difference that each term in a polynomial has an exponent.
Here is the general process for multiplying two polynomials:

1. Write down both polynomials to be multiplied.
2. Distribute the first term of the first polynomial with every term of the second polynomial.
3. Distribute the second term of the first polynomial with every term of the second polynomial.
4. Repeat step 2 and 3 for each term of the first polynomial.
5. Add up all the results obtained in step 2, 3 and 4.

3. Write polynomial addition algorithm if terms are not sorted.
The algorithm for adding two polynomials if the terms are not sorted is as follows:

1. Create an empty polynomial to store the result.
2. Create two pointers, one for each polynomial to be added.
3. Initialize both pointers to the beginning of their respective polynomials.
4. While both pointers are not at the end of their respective polynomials: a. Compare the exponents of the two terms pointed to by the pointers. b. If the exponents are the same, add the coefficients and store the result in the new polynomial with the same exponent. c. If the exponent of one term is greater than the other, add that term to the new polynomial and move the pointer to the next term of that polynomial. d. If the exponent of one term is less than the other, add that term to the new polynomial and move the pointer to the next term of that polynomial.
5. While one of the pointers is not at the end of its respective polynomial, add the remaining terms to the new polynomial.
6. Return the new polynomial which contains the sum of the two polynomials.

Here is an example implementation of the algorithm in C-like language:

```
    struct Term {
        int coefficient;
        int exponent;
        struct Term* next;
      };

      struct Polynomial {
        struct Term* head;
      };

      struct Polynomial* add(struct Polynomial* poly1, struct Polynomial* poly2)
        struct Poly
```

Listing 4: example