MIT World Peace University

Advanced Data Structures
Second Year B. Tech, Semester 4

Addition of Polynomials using Circular Linked Lists

Assignment No. 1

Prepared By

Krishnaraj Thadesar
Cyber Security and Forensics
Batch A1, PA 20

January 23, 2023

Contents

1 Objectives

1. To study data structure: Circular Linked List
2. To Study different operations that could be performed on CLL
3. To Study Applications of Circular Linked list

2 Problem Statement

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

3 Theory

3.1 Brief about Data structure

Data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

Depending on our requirement and project, it is important to choose the right data structure for our project. For example, if we want to store data sequentially in the memory, then you can go for the Array data structure, while if we have something more complex, we may want to opt for graph, or trees.

They are broadly classified into two types:

1. Linear Data Structures: Like Arrays, Linked Lists, Stacks, Queues, etc.
2. Non-Linear Data Structures: Like Trees, Graphs, etc.

3.2 Circular Linked List

Circular Linked List is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.

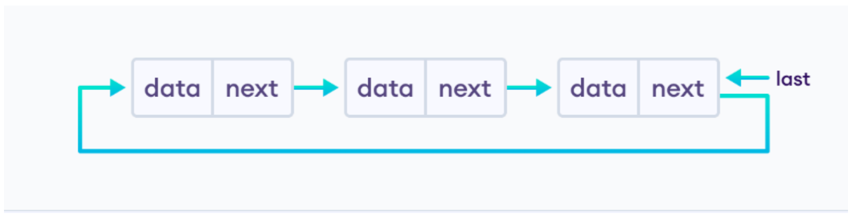

Figure 1: Circular Linked List

3.2.1 Representation using Structures

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

3.2.2 Advantages of Circular Linked List

1. Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

2. Useful for implementation of queue. Unlike linear data structures, we don't need to move all elements after a dequeue operation.

3. Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running in a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the rest of the applications execute. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

4. It is used in multiplayer games to give a chance to each player to play the game.

5. Multiple running applications can be placed in a circular linked list on an operating system. The os keeps on iterating over these applications.

3.3 Difference between Singly Linked List, Circular Linked List, and Double Linked List

3.3.1 Singly Linked List

It is the simplest type of linked list. Each node contains two parts: data and pointer to the next node. The last node points to NULL.

1. Each node has two parts: data and pointer to the next node.

2. Each node has only one pointer to the next node.

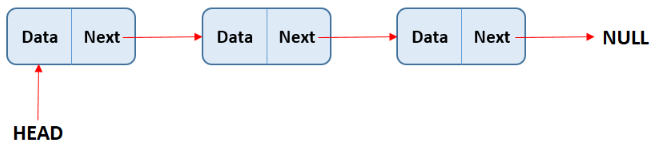

Figure 2:

3.3.2 Circular Linked List

The Major difference between circular linked list and singly linked list is that the last node of the circular linked list points to the first node of the list.

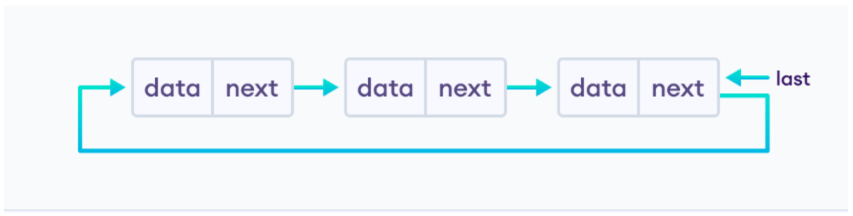

Figure 3:

3.3.3 Doubly Linked List

The major difference between doubly linked list and singly linked list is that the doubly linked list has two pointers: one pointer to the next node and another pointer to the previous node.

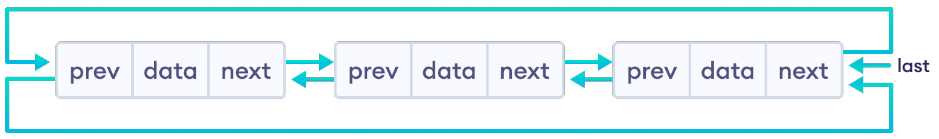

Figure 4:

3.4 Various operations on CLL

All the operations that can be performed on any data structure can be performed on a circular linked list. Some of the operations are:

1. Insertion
2. Deletion
3. Traversal
4. Search
5. Sorting
6. Merging
7. Reversing

The ADT is a set of operations that can be performed on the data structure. It is given further in the FAQ section. The Pseudo Code for these operations is given in the following sections.

4 Platform

Operating System: Arch Linux x86-64
IDEs or Text Editors Used: Visual Studio Code
Compilers : g++ and gcc on linux for C++

5 Input

1. The Choice for what to do
2. The Coefficients and Exponents of the Polynomials
3. Testcase: Addition of the Following 2 Polynomials

$$3x^2 + 5x + 9 \tag{1}$$
$$4x^6 + 8x \tag{2}$$

6 Output

1. The Resultant Polynomial Represented as a Circular Linked List
2. The Sum of the Given 2 Polynomials.
3. The Menu for what to do.
4. Testcase Output:

$$4x^6 + 8x + 3x^2 + 5x + 9 \tag{3}$$

7 Test Conditions

1. Input at least five nodes.
2. Addition of two polynomials with at least 5 terms.
3. Evaluate polynomial with floating values.

8 Pseudo Code

8.1 Pseudo Code for Creating a Circular Linked List

```c
// Create a circular linked list
void create()
{
  struct node *temp, *ptr;
  int i, n;
  printf("Enter the number of nodes: ");
  scanf("%d", &n);
  for (i = 0; i < n; i++)
  {
    temp = (struct node *)malloc(sizeof(struct node));
    printf("Enter the data: ");
    scanf("%d", &temp->data);
    temp->next = NULL;
    if (head == NULL)
    {
      head = temp;
      ptr = temp;
    }
    else
    {
      ptr->next = temp;
      ptr = temp;
    }
  }
  ptr->next = head;
}
```

8.2 Pseudo Code for Displaying a Circular Linked List

```
// Display a circular linked list

void display(struct Node *head){
  struct Node *ptr = head;
  do{
    printf("%d ", ptr->data);
    ptr = ptr->next;
  }while(ptr != head);
  printf("\n");
}
```

8.3 Pseudo Code for Addition of 2 Polynomials using Circular Linked List

```
struct Node *add_polynomials(struct Node *head1, struct Node *head2)
{
  // Pointers for the result polynomial.
  struct Node *result_head = ASSIGN MEMORY
  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 = ASSIGN MEMORY
      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 = ASSIGN MEMORY
      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 = ASSIGN MEMORY
      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;
    }
    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)
  {
```

```c
    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;
}
```

8.4 Pseudo Code for Evaluation of a Polynomial using a Circular Linked List

```c
// evaluate a polynomial using circular linked list
void evaluate(struct Node *head, int x)
{
  struct Node *temp = head->next;
  int result = 0;

  // Loop to evaluate the polynomial.
  while (temp != head)
  {
    result += temp->coeff * pow(x, temp->exp);
    temp = temp->next;
  }

  printf("The value of the polynomial is %d", result);
}
```

9 Code

9.1 Program

```c
// Circular linked List
// Implementing the following polynomial operations using circular linked list.
    Create Dislay and Add.

// Krishnaraj Thadesar
// Jan 22nd 2023
// Assignment 1
// Advanced Data Structures
// Semester 4

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

// The default way to represent circular linked lists using structures.
struct Node
```

```c
{
    int coeff;
    int exp;
    struct Node *next;
};

// Function to accept the polynomial from the user.
void accept_polynomial(struct Node *head)
{
    struct Node *temp = head;
    int choice = 0;
    do
    {
        struct Node *curr = (struct Node *)malloc(sizeof(struct Node));
        printf("\nEnter the Coefficient: ");
        scanf("%d", &curr->coeff);
        printf("\nEnter the Exponent: ");
        scanf("%d", &curr->exp);

        // Main Logic of inserting node at the end and making it point to the head
    .
        curr->next = head;
        temp->next = curr;
        temp = temp->next;

        printf("Do you want to enter more terms? (0 for no, 1 for yes) \n");
        scanf("%d", &choice);
    } while (choice != 0);
}

// Function to display the polynomial.
int display_polynomial(struct Node *head)
{
    // Edge Case of empty list.
    if (head->next == head)
    {
        printf("\nNo terms in the polynomial");
        return -1;
    }

    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");
}

// Function to add two polynomials, Returns the head of the added polynomial.
// Takes as input the heads of the other 2 polynomials that you want to add.
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;
        }
        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;
    printf("Hello! What do you want to do? \n Remember to enter linked list in the
 descending order of exponents. \n");
    struct Node *head = (struct Node *)malloc(sizeof(struct Node));
    struct Node *head2 = (struct Node *)malloc(sizeof(struct Node));
    struct Node *head3 = (struct Node *)malloc(sizeof(struct Node));
    struct Node *added;

```

```c
    while (choice != 4)
    {
        printf("\n\
1. Create a Polynomial to represent it in a circular linked list. \n\
2. Add 2 Polynomials\n\
3. Display your Polynomial\n\
4. Quit\n");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1:
            accept_polynomial(head);
            display_polynomial(head);
            break;
        case 2:
            printf("Please enter the first polynomial");
            accept_polynomial(head2);
            printf("\nThe First Polynomial you entered is:  \n");
            display_polynomial(head2);

            printf("Please enter the second polynomial");
            accept_polynomial(head3);
            printf("\nThe Second Polynomial you entered is:  \n");
            display_polynomial(head3);

            printf("\nThe Added Polynomial:  \n");
            added = add_polynomials(head2, head3);
            display_polynomial(added);
            break;
        case 3:
            display_polynomial(head);
        case 4:
            break;
        default:
            printf("\nInvalid\n");
            break;
        }
    }

    return 0;
}
```

9.2 Input and Output

```
Hello! What do you want to do?
 Remember to enter linked list in the descending order of exponents.

1. Create a Polynomial to represent it in a circular linked list.
2. Add 2 Polynomials
3. Display your Polynomial
4. Quit
1

Enter the Coefficient: 1

Enter the Exponent: 1
Do you want to enter more terms? (0 for no, 1 for yes)
1

```

```
Enter the Coefficient: 2

Enter the Exponent: 3
Do you want to enter more terms? (0 for no, 1 for yes)
0
1x^1 + 2x^3

1. Create a Polynomial to represent it in a circular linked list.
2. Add 2 Polynomials
3. Display your Polynomial
4. Quit
2
Please enter the first polynomial
Enter the Coefficient: 1

Enter the Exponent: 3
Do you want to enter more terms? (0 for no, 1 for yes)
1

Enter the Coefficient: 2

Enter the Exponent: 2
Do you want to enter more terms? (0 for no, 1 for yes)
1

Enter the Coefficient: 1

Enter the Exponent: 1
Do you want to enter more terms? (0 for no, 1 for yes)
0

The First Polynomial you entered is:
1x^3 + 2x^2 + 1x^1
Please enter the second polynomial
Enter the Coefficient: 2

Enter the Exponent: 3
Do you want to enter more terms? (0 for no, 1 for yes)
1

Enter the Coefficient: 5

Enter the Exponent: 2
Do you want to enter more terms? (0 for no, 1 for yes)
1

Enter the Coefficient: 1

Enter the Exponent: 1
Do you want to enter more terms? (0 for no, 1 for yes)
0

The Second Polynomial you entered is:
2x^3 + 5x^2 + 1x^1

The Added Polynomial:
3x^3 + 7x^2 + 2x^1

1. Create a Polynomial to represent it in a circular linked list.
```

```
2. Add 2 Polynomials
3. Display your Polynomial
4. Quit
4
```

10 Conclusion

Thus, implemented different operations on CLL.

11 FAQ

1. **Write an ADT for CLL.**

ADT for CLL is as follows

```c

// Structure for a node in the circular linked list.
struct Node
{
  int data;
  struct Node *next;
};

```

Function to create a new node.

```c

struct Node *create_node(int data)
{
  struct Node *new_node = (struct Node *)malloc(sizeof(struct Node));
  new_node->data = data;
  new_node->next = NULL;
  return new_node;
}

```

Function to create a circular linked list.

```c

struct Node *create_circular_linked_list(int data)
{
  struct Node *head = create_node(data);
  head->next = head;
  return head;
}


```

Function to insert a node at the end of the circular linked list.

```c

struct Node *insert_at_end(struct Node *head, int data)
{
  struct Node *new_node = create_node(data);
  struct Node *temp = head;
  while (temp->next != head)
  {
    temp = temp->next;
  }
  temp->next = new_node;
  new_node->next = head;
  return head;
}


```

Function to insert a node at a given position in the circular linked list.

```c

struct Node *insert_at_position(struct Node *head, int data, int position)
{
  struct Node *new_node = create_node(data);
  struct Node *temp = head;
  int i = 1;
  while (i < position - 1)
  {
    temp = temp->next;
    i++;
  }
  new_node->next = temp->next;
  temp->next = new_node;
  return head;
}

```

Function to delete a node from the beginning of the circular linked list.

```c

struct Node *delete_from_beginning(struct Node *head)
{
  struct Node *temp = head;
  while (temp->next != head)
  {
    temp = temp->next;
  }
  temp->next = head->next;
  free(head);
  head = temp->next;
  return head;
}

```

Function to delete a node from a given position in the circular linked list.

```c

struct Node *delete_from_position(struct Node *head, int position)
{
  struct Node *temp = head;
  int i = 1;
  while (i < position - 1)
  {
    temp = temp->next;
    i++;
  }
  struct Node *temp2 = temp->next;
  temp->next = temp2->next;
  free(temp2);
  return head;
}

```

Function to display the circular linked list.

```c

void display(struct Node *head)
{
  struct Node *temp = head;
  while (temp->next != head)
```

```
    {
      printf("%d ", temp->data);
      temp = temp->next;
    }
    printf("%d ", temp->data);
    printf(" ");
  }

```

Function to search for a node in the circular linked list.

```

  int search(struct Node *head, int data)
  {
    struct Node *temp = head;
    int position = 1;
    while (temp->next != head)
    {
      if (temp->data == data)
      {
        return position;
      }
      temp = temp->next;
      position++;
    }
    if (temp->data == data)
    {
      return position;
    }
    return -1;
  }

```

Function to sort the circular linked list.

```

  struct Node *sort(struct Node *head)
  {
    struct Node *temp = head;
    struct Node *temp2 = NULL;
    int temp_data;
    while (temp->next != head)
    {
      temp2 = temp->next;
      while (temp2 != head)
      {
        if (temp->data > temp2->data)
        {
          temp_data = temp->data;
          temp->data = temp2->data;
          temp2->data = temp_data;
        }
        temp2 = temp2->next;
      }
      temp = temp->next;
    }
    return head;
  }


```

2. **How to perform multiplication of two polynomials?**

   Multiplication of two polynomials is similar to multiplication of two numbers.

   (a) For each term in the first polynomial, multiply it with each term in the second polynomial.

   (b) While multiplying, add the exponents of the two terms. and Multiply the Coefficients.

3. **Write polynomial addition algorithm if terms are not sorted.**

   In case the terms are not sorted, we can either sort both the polynomials, and then proceed to add them in the usual way, or we can follow this algorithm

   (a) For each polynomial in the First Polynomial, check for the presence of a term with the same exponent in the other polynomial by looping over it.

   (b) If the term is present, add the coefficients of the two terms, and store the result. Thereby Incrementing the index of the first polynomial.

   (c) If the term is not present, store the term in the result polynomial. Thereby Incrementing the index of the first polynomial.