Addition using Linked List

Aim:
The aim of this program is to add two numbers represented as linked lists, where each node of the linked list contains a single digit, and return the result as a linked list.

Theory:
This program utilizes the concept of adding two numbers represented by linked lists. It follows the elementary school arithmetic approach of adding digits from the least significant to the most significant, handling carry over when the sum exceeds 9.

Code:
```
#include <stdio.h>
#include <stdlib.h>

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

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

void push(Node **head_ref, int new_data)
{
   Node *new_node = newNode(new_data);
   new_node->next = (*head_ref);
   (*head_ref) = new_node;
}

Node *addTwoLists(Node *first, Node *second)
{
   Node *res = NULL;
   Node *temp, *prev = NULL;
   int carry = 0, sum;

   while (first != NULL || second != NULL)
   {
      sum = carry + (first ? first->data : 0) + (second ? second->data : 0);
      carry = (sum >= 10) ? 1 : 0;
      sum = sum % 10;
      temp = newNode(sum);
      if (res == NULL)
         res = temp;
      else
         prev->next = temp;
```

```
        prev = temp;

        if (first)
            first = first->next;
        if (second)
            second = second->next;
    }
    if (carry > 0)
        temp->next = newNode(carry);
    return res;
}

Node *reverse(Node *head)
{
    if (head == NULL || head->next == NULL)
        return head;
    Node *rest = reverse(head->next);
    head->next->next = head;
    head->next = NULL;
    return rest;
}

void printList(Node *node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
    printf("\n");
}

void freeList(Node *head)
{
    Node *temp;
    while (head != NULL)
    {
        temp = head;
        head = head->next;
        free(temp);
    }
}

int main(void)
{
    Node *res = NULL;
    Node *first = NULL;
    Node *second = NULL;

    int num1, num2;
    printf("Enter the first number: ");
    scanf("%d", &num1);
```

```
    printf("Enter the second number: ");
    scanf("%d", &num2);

    while (num1 > 0)
    {
        int digit = num1 % 10;
        push(&first, digit);
        num1 /= 10;
    }

    while (num2 > 0)
    {
        int digit = num2 % 10;
        push(&second, digit);
        num2 /= 10;
    }

    printf("First list is ");
    printList(first);
    printf("Second list is ");
    printList(second);

    first = reverse(first);
    second = reverse(second);

    res = addTwoLists(first, second);
    res = reverse(res);

    printf("Resultant list is ");
    printList(res);

    // Free allocated memory
    freeList(first);
    freeList(second);
    freeList(res);

    return 0;
}
```

Output:

```
go-d-code@code-valley:~/Roger/College/DAA_Lab_Codes/DAA_2$ gcc addition_LL.c -o addition_LL && ./addition_LL
Enter the first number: 999
Enter the second number: 1
First list is 9 9 9
Second list is 1
Resultant list is 1 0 0 0
```

Algorithm Analysis

Algorithm steps:

1. Initialize two empty linked lists to represent the input numbers.
2. Read the input numbers from the user and push their digits onto the respective linked lists.

3. Reverse both linked lists to facilitate addition from the least significant digit.
4. Add the two linked lists digit by digit, handling carry over as needed.
5. Reverse the result linked list to obtain the final result.
6. Print the final result.

Example:

1->2->3

+4->5->6

5->7->9

Time Complexity:

1. **Traversing Input Lists:**

- The algorithm traverses both input linked lists once to add corresponding digits. This traversal takes O(n) time, where n is the length of the longer linked list.

2. **Addition and Carry Over:**

- At each step, the algorithm performs constant-time operations such as addition, modulo, and checking for carry over. These operations don't depend on the size of the input but happen for each digit in the linked lists.

3. **Reversing the Result:**

- After adding the numbers, the algorithm reverses the result linked list, which again takes O(n) time, where n is the length of the result linked list.

Space Complexity:

1. **Input Lists:**

- The algorithm uses two linked lists to store the input numbers. The space required for these linked lists is proportional to the lengths of the input numbers, denoted as m and n.

2. **Result List:**

- Another linked list is used to store the result. The space required for this list is proportional to the length of the result, which could be at most max(m, n) + 1.

3. **Additional Variables:**

- The algorithm uses a few additional variables such as pointers and loop counters, which require constant space.

Multiplication using Linked List

Aim:
The aim of this program is to multiply two numbers represented as linked lists, where each node of the linked list contains a single digit, and return the result as a linked list.

Theory:
This program aims to multiply two numbers represented as linked lists, where each node of the linked list contains a single digit. It utilizes the concept of multiplying two numbers digit by digit, with carry-over handled appropriately.

Code:
```
#include <stdio.h>
#include <stdlib.h>

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

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

void push(struct Node **head_ref, int new_data)
{
   Node *new_node = newNode(new_data);
   new_node->next = (*head_ref);
   (*head_ref) = new_node;
}

long long multiplyTwoLists(Node *first, Node *second)
{
   long long N = 1000000007;
   long long num1 = 0, num2 = 0;

   while (first || second)
   {
      if (first)
      {
         num1 = ((num1) * 10) % N + first->data;
         first = first->next;
      }

      if (second)
      {
         num2 = ((num2) * 10) % N + second->data;
```

```
            second = second->next;
        }
    }
    return ((num1 % N) * (num2 % N)) % N;
}

void printList(struct Node *node)
{
    while (node != NULL)
    {
        printf("%d", node->data);
        if (node->next)
            printf("->");
        node = node->next;
    }
    printf("\n");
}

void freeList(struct Node *head)
{
    Node *temp;
    while (head != NULL)
    {
        temp = head;
        head = head->next;
        free(temp);
    }
}

int main()
{
    Node *first = NULL;
    Node *second = NULL;

    int num1, num2;
    printf("Enter the first number: ");
    scanf("%d", &num1);
    printf("Enter the second number: ");
    scanf("%d", &num2);

    while (num1 > 0)
    {
        int digit = num1 % 10;
        push(&first, digit);
        num1 /= 10;
    }

    while (num2 > 0)
    {
        int digit = num2 % 10;
        push(&second, digit);
        num2 /= 10;
```

```
    }

    printf("First List is: ");
    printList(first);
    printf("Second List is: ");
    printList(second);

    printf("Result is: ");
    printf("%lld", multiplyTwoLists(first, second));

    // Free allocated memory
    freeList(first);
    freeList(second);
    printf("\n");
    return 0;
}
```

Output:

```
go-d-code@code-valley:~/Roger/College/DAA_Lab_Codes/DAA_2$ gcc mult_LL.c -o mult_LL && ./mult_LL
Enter the first number: 110
Enter the second number: 11
First List is: 1->1->0
Second List is: 1->1
Result is: 1210
```

Algorithm Analysis

Algorithm steps:

1. Initialize two empty linked lists to represent the input numbers.
2. Read the input numbers from the user and push their digits onto the respective linked lists.
3. Traverse both linked lists simultaneously, multiplying corresponding digits and updating the intermediate product.
4. Handle carry-over by adding the appropriate digit to the result or carrying over to the next digit.
5. Print the final product.

Example:

1->1->0

*1->1

1->2->1->0

Time Complexity:

1. **Traversing Input Lists:**

- The algorithm traverses both input linked lists once to multiply corresponding digits. This traversal takes O(n) time, where n is the length of the longer linked list.

2. **Multiplication and Carry Over:**

- At each step, the algorithm performs constant-time operations such as multiplication, modulo, and checking for carry over. These operations don't depend on the size of the input but happen for each digit in the linked lists.

Space Complexity:

1. **Input Lists:**
- The algorithm uses two linked lists to store the input numbers. The space required for these linked lists is proportional to the lengths of the input numbers, denoted as m and n.

2. **Intermediate Product:**
- The algorithm computes the product of the two numbers digit by digit. The intermediate product may have at most m + n digits, where m and n are the lengths of the input numbers. Thus, the space required for the intermediate product is O(m + n).

3. **Result List:**
- The space required for the result linked list is also proportional to the size of the intermediate product, which is O(m + n).

4. **Additional Variables:**
- The algorithm uses a few additional variables such as pointers and loop counters, which require constant space.