CSE2225

Project #1 Report

Detailed explanation of multiplication and base conversion algorithms.

Burak Karayağlı
150121824

Contents

Linked List

I created a node struct. This struct holds the data, the next node and previous node.

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

I've implemented few methods for creating linked list with these nodes.

```
void push(int data, Node **head, Node **tail) {

    Node* newNode = createNode(data);

    //Checking linked list is empty or not
    if(*head == NULL) {
        (*head) = newNode;
        (*tail) = newNode;
        return;
    }

    (*head)->prev = newNode;
    newNode->next = (*head);
    (*head) = newNode;
    return;

}

void append(int data, Node **head, Node **tail) {
    Node* newNode = createNode(data);

    //Checking linked list is empty or not
    if(*head == NULL) {
        (*head) = newNode;
        (*tail) = newNode;
        return;
    }


    (*tail)->next = newNode;
    newNode->prev = (*tail);
    (*tail) = newNode;
    return;
}
```

The push method adding the node beginning of the linked list and the append method adding the node end of the linked list.

I added printing methods for linked lists.

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

Mutliplication

I created a function for multiplication. This function takes number as linked list. Multiplier, multiplicand, result and base. Base argument in int data type. I gave 2 argument for each number because I'm holding the head and tail of the linked lists.

```
void multiplyLL(Node* headMultiplier, Node* tailMultiplier, Node*
headMultiplicand, Node* tailMultiplicand, Node **headResult, Node
**tailResult, int base)
```

Then I created a lookup table for not calculating same things again and again. I'm initilazing an array of size base.

```
Node *lookupHead[base];
Node *lookupTail[base];
int i;
for(i = 0; i < base; i++){
    lookupHead[i] = NULL;
    lookupTail[i] = NULL;
}
```

My algorithm uses elementary multiplication, I'm starting to read digits from the end of the number. I'm multiplying multiplier digit by digit. So if a digits repats in the multiplicand I'm not recalculating it, I'm just calling from the lookup table. Before the starting to the while loop I'm initializing three nodes, currentMultiplicand for iterating multiplicand, currentResult and ShiftPtr for adding operations in multiplication.

```
Node *currentMultiplicand = tailMultiplicand;
Node *currentResult = NULL;
Node *shiftPtr = NULL;
```

I'm starting the loop with the last digit of multiplicand, then I create nodes for multiplier * digit. Firstly, I'm checking if this calculation in the lookup table or not. If it is in I'm passing the multiplication part.

```
Node *headProduct = NULL;
Node *tailProduct = NULL;

if (lookupHead[currentMultiplicand->data] != NULL) {
    headProduct = lookupHead[currentMultiplicand->data];
    tailProduct = lookupTail[currentMultiplicand->data];
}
```

Then I started to multiplier loop for iterating it with same method. I initialized carry and value.

```
else {
    int multiplyCarry = 0;
    int value;
    Node *currentMultiplier = tailMultiplier;
    while(currentMultiplier) {

        int temp = ((currentMultiplier->data)*(currentMultiplicand-
>data)+multiplyCarry);
        value = temp % base;
        multiplyCarry = temp / base;
        push(value, &headProduct, &tailProduct);
```

I'm multiplying current digits and adding carry for calculating temp value. First carry value initilazed as 0, so at first it is ineffective element. After calculated temp value I'm calculating push value with mod base, and calculating carry with division to base. I'm adding these values to the product linked list. For the last digit I added an if statement for adding carry value.

```
if (currentMultiplier == headMultiplier && temp >= base) {
    push(multiplyCarry, &headProduct, &tailProduct);
}
```

After this I'm storing this product to the lookup table and exiting from the inner loop. Multiplication part of the method is done. Now I'm adding these products one by one and getting final result.

```
lookupHead[currentMultiplicand->data] = headProduct;
lookupTail[currentMultiplicand->data] = tailProduct;
```

For summation part I first declared result, if this multiplication is the first one I'm declaring result as product. Then I'm declaring shift ptr too. Shift ptr is provides the shifting in summation.

```
if((*headResult) == NULL) {
    Node *currentProduct = headProduct;
    while(currentProduct != NULL) {
        append(currentProduct->data, headResult, tailResult);
        currentProduct = currentProduct->next;
    }
    shiftPtr = *tailResult;
}
```

If this is not the first product I'm doing elementary summation. It's look like multiplication algorithm. I'm initializing some values, and I'm shifting the number. So I'm not doing anything to the right sides of the numbers while summation.

```
else {
    int carry = 0;
    int value;
    int temp;
    /*
    printLinkedList(headResult);
    printLinkedList(headProduct);
    printf("================\n");
    */
    currentResult = shiftPtr->prev;
```

I'm adding values in the result with carry, and I'm iterating in the both result and product numbers.

```
while (currentResult != NULL) {
    //printf("Result: %d, Product: %d\n", currentResult->data, tailProduct-
>data);
    temp = currentResult->data + tailProduct->data + carry;
    value = temp % base;
    carry = temp / base;
    currentResult->data = value;


    tailProduct = tailProduct->prev;
    currentResult = currentResult->prev;
}
```

Lastly I'm adding the extra digits because of the shifting.

```
while (tailProduct != NULL) {
    temp = tailProduct->data + carry;
    value = temp % base;
    carry = temp / base;
    push(value, headResult, tailResult);
    tailProduct = tailProduct -> prev;
}

if (temp >= base) push(carry, headResult, tailResult);
```

After this I'm reassigning shiftPtr and currentMultiplicand with ->prev. And multiplication part is done.

Summation

In this method I used same logic in the multiplication summation. I'm iterating both the number while both of them not equal to NULL I'm adding these digits. Then if one of them is equal I'm just iterating over the the other number and continuing to adding.

```
int carry = 0;
int value;
int temp;
Node *currentAddend = tailAddend;
Node *currentAdder = tailAdder;
while (currentAddend != NULL && currentAdder != NULL) {
    temp = currentAddend->data + currentAdder->data + carry;
    value = temp % base;
    carry = temp / base;
    push(value, headResult, tailResult);
    currentAddend = currentAddend->prev;
    currentAdder = currentAdder->prev;
}

while (currentAddend != NULL) {
    temp = currentAddend->data + carry;
    value = temp % base;
    carry = temp / base;
    push(value, headResult, tailResult);
    currentAddend = currentAddend->prev;
}

while (currentAdder != NULL) {
    temp = currentAdder->data + carry;
    value = temp % base;
    carry = temp / base;
    push(value, headResult, tailResult);
    currentAdder = currentAdder->prev;
}

if (temp >= base) push(carry, headResult, tailResult);
```

Base Conversion

Firstly, I implemented the basic base conversion algorithm. Multiplying digits with the base^n, later I discovered better way. It starts to iterate from the beginning. I initialized the result number with 0. The algorithm multiplies results with base, then adding the current digit. Lastly reassign this to result. When iteration is done, the result is the decimal base. I created the linked list holds the base.

```
Node *headBaseNumber = NULL;
Node *tailBaseNumber = NULL;
append(base, &headBaseNumber, &tailBaseNumber);
```

Then iterators and results:

```
Node* currentMultiplier = headMultiplier;
Node* currentMultiplicand = headMultiplicand;

append(0, headDecimalMultiplier, tailDecimalMultiplier);
append(0, headDecimalMultiplicand, tailDecimalMultiplicand);
```

After this I started to iteration and make both operations in one while loop.

```
while (currentMultiplier != NULL || currentMultiplicand != NULL) {


    if (currentMultiplier != NULL) {
        Node* headDigit =NULL ; Node* tailDigit = NULL;
        append(currentMultiplier->data, &headDigit, &tailDigit);

        Node *headProduct = NULL; Node *tailProduct = NULL;
        multiplyLL(*headDecimalMultiplier,*tailDecimalMultiplier,
headBaseNumber, tailBaseNumber, &headProduct, &tailProduct, 10);
        Node *headTempResult = NULL; Node *tailTempResult = NULL;
        addLL(headProduct, tailProduct, headDigit, tailDigit,
&headTempResult, &tailTempResult, 10);

        *headDecimalMultiplier = headTempResult;
        *tailDecimalMultiplier = tailTempResult;

        currentMultiplier = currentMultiplier->next;
    }
```

In these operations I implemented the algorithm and get the result. For the multiplicand part I did the same things.

After the iteration I multiplied these decimal numbers in base ten. Because multiplication is faster than conversion.

```
multiplyLL(*headDecimalMultiplier, *tailDecimalMultiplier,
*headDecimalMultiplicand, *tailDecimalMultiplicand, headDecimalResult,
tailDecimalResult, 10);
```

File Input and Output

I implemented input in the main function but I created function for output. I'm reading files line by line.

```
//File reading
FILE* ptr;
char ch;

ptr = fopen(argv[1], "r");

if (NULL == ptr) {
    printf("file can't be opened \n");
}

do {
    ch = fgetc(ptr);

    if(ch != '\n') append(ch - '0', &headMultiplier, &tailMultiplier);


    // Checking if character is not EOF.
    // If it is EOF stop eading.
} while (ch != '\n');

do {
    ch = fgetc(ptr);
    //printf("%c", ch);
    if(ch != '\n') append(ch - '0', &headMultiplicand, &tailMultiplicand);
    // Checking if character is not EOF.
    // If it is EOF stop eading.
} while (ch != '\n');

base = fgetc(ptr) - '0';
if(base==1) base = 10;

// Closing the file
fclose(ptr);
```

I'm reading one char in the last line. If this char equals to 1, I'm making base 1. In the output function I'm taking arguments and writing it to text file.

Lastly I'm initializing necessary nodes and calling multiplying and conversion functions. Then print them to the console and calling output function.

```
multiplyLL(headMultiplier, tailMultiplier, headMultiplicand,
tailMultiplicand, &headResult, &tailResult, base);
```

```
baseConverter(headMultiplier, tailMultiplier, headMultiplicand,
tailMultiplicand,
              &headDecimalMultiplier, &tailDecimalMultiplier,
              &headDecimalMultiplicand, &tailDecimalMultiplicand,
              &headDecimalResult, &tailDecimalResult, base);
```

```
createOutputFile(headMultiplier, headMultiplicand, headResult,
headDecimalMultiplier, headDecimalMultiplicand, headDecimalResult);
```

Conclusion

While developing this project I learnt much. I implemented different algorithms to make program more efficient. I implemented two base conversion method, two summation method etc. I compared their Big O to decide. The other thing which I learnt is. Making program modular with functions. In the beginning of the project I implemented multiplication in the main method without function. But when I realized that I will need multiplication in the conversion too. I converted it to function. Then I created the other functions. While creating linked list I used doubly linked list, so I stored head and tails of the linked lists. In the end of the project I realized I should create another struct for holding this head and tail in the struct. This makes my job easier and makes the code more readable.