南京邮电大学

实 验 报 告

（2024 / 2025 学年 第 一 学期）

课程名称	数据结构
实验名称	线性结构
实验时间	2024 年 9 月 25 日
指导单位	计算机学院
指导教师	孙海安

学生姓名	于明宏	班级学号	B23041011
学院(系)	计算机学院	专　　业	信息安全

实 验 报 告

实验名称	线性结构			指导教师	孙海安
实验类型	设计	实验学时	2	实验时间	2024.9.25

一、 实验目的和要求

1. 掌握线性表的顺序存储和链式存储这两种基本存储结构及其应用场合。
2. 掌握顺序表和链表的各种基本操作算法。
3. 理解线性表应用于多项式的实现算法。

二、实验环境(实验设备)

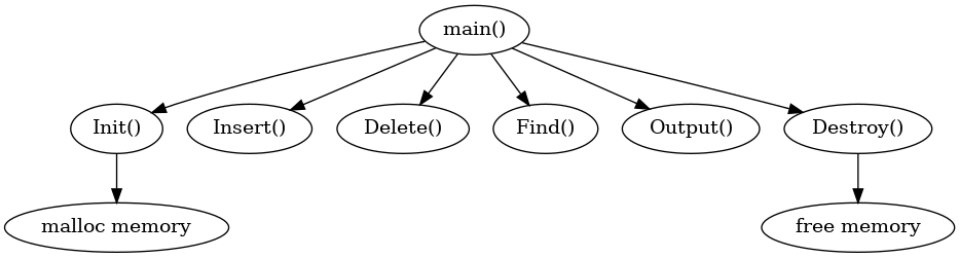
硬件：微型计算机

软件：Windows 11 Professional Edition 23H2、Microsoft Visual C++ 2022

三、实验原理及内容

题目 1：

1、算法设计



2、算法实现

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

// Define types for element and status
```

```
typedef int ElemType; // Element type in the sequence list
typedef int Status;      // Status for return values

// Define the structure for the sequential list
typedef struct {
    int n;                  // Current number of elements
    int maxLength;       // Maximum allowed number of elements
    ElemType* element; // Pointer to the element array
} SeqList;

// Initialize the sequence list with a given size
Status Init(SeqList* L, int mSize) {
    L->maxLength = mSize;    // Set maximum length
    L->n = 0;                  // Set current number of elements to 0
    L->element = (ElemType*)malloc(sizeof(ElemType) * mSize); // Allocate memory for elements
    if (L->element)
        return 1;              // Return success if memory allocation succeeds
    exit(0);                   // Exit if memory allocation fails
}

// Find the element at position i in the list
Status Find(SeqList seqList, int i, ElemType* x) {
    if (i < 0 || i > seqList.n - 1) {
        return 0;              // Return failure if index is out of bounds
    }
    *x = seqList.element[i]; // Set the value at position i to x
    return 1;                  // Return success
}

// Insert an element at position i in the list
Status Insert(SeqList* seqList, int i, ElemType x) {
    int j;
    if (i < -1 || i > seqList->n - 1)
        return 0;              // Return failure if index is invalid
    if (seqList->n == seqList->maxLength)
```

```
        return 0;                   // Return failure if the list is full
    for (j = seqList->n - 1; j > i; j--) {
        seqList->element[j + 1] = seqList->element[j]; // Shift elements to the right
    }
    seqList->element[i + 1] = x; // Insert the new element
    seqList->n++;                    // Increase the number of elements
    return 1;                        // Return success
}

// Delete the element at position i in the list
Status Delete(SeqList* seqList, int i) {
    int j;
    if (i < 0 || i > seqList->n - 1) {
        return 0;                   // Return failure if index is invalid
    }
    if (!seqList->n) {
        return 0;                   // Return failure if the list is empty
    }
    for (j = i + 1; j < seqList->n; j++) {
        seqList->element[j - 1] = seqList->element[j]; // Shift elements to the left
    }
    seqList->n--;                   // Decrease the number of elements
    return 1;                       // Return success
}

// Output all the elements in the list
int Output(SeqList seqList) {
    int i;
    if (!seqList.n)
        return 0;                   // Return failure if the list is empty
    for (i = 0; i <= seqList.n - 1; i++)
        printf("%d ", seqList.element[i]); // Print each element
    return 1;                       // Return success
}
```

```
// Destroy the sequence list and free the allocated memory
void Destroy(SeqList* L) {
    (*L).n = 0;                    // Reset the number of elements to 0
    (*L).maxLength = 0;            // Reset the maximum length to 0
    free((*L).element);            // Free the allocated memory
}

// Main function
int main(){
    int i, j, delPos, findPos, n, findResult;
    SeqList list;

    printf("Number of elements:\n");
    scanf_s("%d", &n);              // Input the number of elements
    Init(&list, n);                 // Initialize the list
    printf("Elements: \n");
    for (i = 0; i < n; i++) {
        scanf_s("%d", &j);          // Input elements
        Insert(&list, i - 1, j); // Insert elements into the list
    }
    printf("Sequence list: ");
    Output(list);                   // Output the current list
    printf("\n");

    printf("Delete element at this position: \n");
    scanf_s("%d", &delPos);      // Input the position to delete
    Delete(&list, delPos);       // Delete the element at that position
    printf("Sequence list after deleting this element: \n");
    Output(list);                   // Output the list after deletion
    printf("\n");

    printf("Find element at this position: \n");
    scanf_s("%d", &findPos);    // Input the position to find
    Find(list, findPos, &findResult); // Find the element at that position
    printf("This element: %d", findResult); // Print the found element
```

```
    printf("\n");

    Destroy(&list);              // Destroy the list and free memory
    return 0;                    // End the program
}
```

3、复杂度分析

（1）Init 函数

时间复杂度：O(1)

该函数进行初始化操作，分配内存和设置初始值，这些操作都只进行一次，时间复杂度为 O(1)。

（2）Find 函数

时间复杂度：O(1)

查找操作通过数组的随机访问，直接根据索引找到元素，访问数组元素的时间复杂度为 O(1)。

（3）Insert 函数

时间复杂度：O(n)

插入操作最坏情况是在数组的头部插入，导致所有元素需要向后移动，因此在最坏情况下，时间复杂度为 O(n)，其中 n 为顺序表的当前元素个数。

（4）Delete 函数

时间复杂度：O(n)

删除操作最坏情况下是在数组的头部删除，导致所有元素需要向前移动，因此最坏情况的时间复杂度为 O(n)。

（5）Output 函数

时间复杂度：O(n)

遍历顺序表中的所有元素并输出，每个元素都会被访问一次，因此时间复杂度为 O(n)。

（6）Destroy 函数

时间复杂度：O(1)

销毁操作只需要释放内存并重置属性，不涉及遍历操作，时间复杂度为 O(1)。

4、实验结果与结论

```
Number of elements:
10
Elements:
1 2 3 4 5 6 7 8 9 0
Sequence list: 1 2 3 4 5 6 7 8 9 0
Delete element at this position:
4
Sequence list after deleting this element:
```

1 2 3 4 6 7 8 9 0

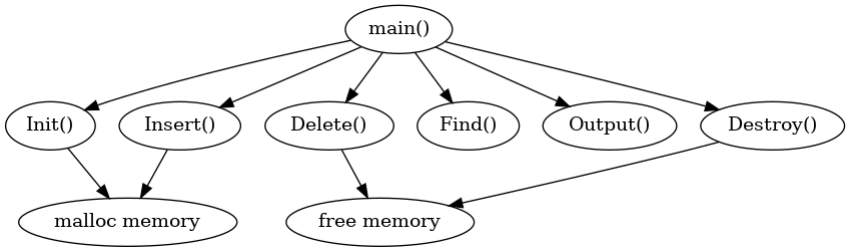
Find element at this position:

5

This element: 7

题目 2：

1、 算法设计



2、算法实现

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

typedef int ElemType;    // Define ElemType as int to represent the type of elements stored in the list nodes
typedef int Status;      // Define Status as int to represent the status of function returns (success or failure)

// Define the structure for a list node
typedef struct Node {
    ElemType element;            // The element stored in the node
    struct Node* link;           // Pointer to the next node in the list
} Node;

// Define the structure for the list header
typedef struct {
    struct Node* head;    // Pointer to the head node of the list
    int n;                       // Number of elements in the list
} ListHeader;

// Function to initialize the list by creating a head node
Status Init(ListHeader* h) {
```

```
        h->head = (Node*)malloc(sizeof(Node));    // Allocate memory for the head node
        if (!h->head) {                           // Check if memory allocation was successful
            return 0;                             // Return 0 if allocation failed
        }
        h->head->link = NULL;                     // Initialize the link pointer of the head node to NULL
        h->n = 0;                                 // Initialize the number of elements in the list to 0
        return 1;                                 // Return 1 to indicate successful initialization
    }

    // Function to find the element at position i in the list
    Status Find(ListHeader* h, int i, ElemType* x) {
        Node* p;
        int j;
        if (i < 0 || i > h->n - 1) {    // Check if the index i is within valid range
            return 0;                   // Return 0 if the index is invalid
        }
        p = h->head->link;              // Set p to point to the first node (after the head)
        for (j = 0; j < i; j++) {       // Traverse the list to the ith node
            p = p->link;                // Move p to the next node
        }
        *x = p->element;                // Assign the element at the ith node to *x
        return 1;                       // Return 1 to indicate success
    }

    // Function to insert an element x at position i in the list
    Status Insert(ListHeader* h, int i, ElemType x) {
        Node* p, * q;
        int j;
        if (i < -1 || i > h->n - 1) {    // Check if the index i is valid for insertion
            return 0;                    // Return 0 if the index is invalid
        }
        p = h->head;                     // Set p to the head node
        for (j = 0; j <= i; j++) {       // Move p to the ith node
```

```
            p = p->link;
        }
        q = (Node*)malloc(sizeof(Node));    // Allocate memory for the new node
        q->element = x;                     // Set the new node's element to x
        q->link = p->link;                  // Set the new node's link to point to the next node
        p->link = q;                        // Link the previous node to the new node
        h->n++;                             // Increment the number of elements in the list
        return 1;                           // Return 1 to indicate success
    }

    // Function to delete the element at position i in the list
    Status Delete(ListHeader* h, int i) {
        int j;
        Node* p, * q;
        if (!h->n) {                        // If the list is empty, return failure
            return 0;
            if (i < 0 || i > h->n - 1) {    // Check if the index i is valid for deletion
                return 0;
            }
        }
        q = h->head;                        // Set q to the head node
        for (j = 0; j < i; j++) {           // Move q to the node just before the ith node
            q = q->link;
        }
        p = q->link;                        // Set p to the ith node (the one to be deleted)
        q->link = q->link->link;            // Link the previous node to the next node, skipping the ith node
        free(p);                            // Free the memory of the deleted node
        h->n--;                             // Decrement the number of elements in the list
        return 1;                           // Return 1 to indicate success
    }

    // Function to output the elements of the list
    Status Output(ListHeader* h) {
        Node* p;
```

```
    if (!h->n) {                        // If the list is empty, return failure
        return 0;
    }
    p = h->head->link;                  // Set p to the first node (after the head)
    while (p) {                         // Traverse the list
        printf("%d ", p->element); // Print the element of the current node
        p = p->link;                    // Move to the next node
    }
    printf("\n");                       // Print a newline after the list output
    return 1;                           // Return 1 to indicate success
}

// Function to destroy the list and free all allocated memory
void Destroy(ListHeader* h) {
    Node* p, * q;
    while (h->head->link) {             // While the list is not empty
        q = h->head->link;              // Set q to the first node
        p = h->head->link->link;        // Set p to the next node
        free(h->head->link);            // Free the memory of the current node
        h->head = q;                    // Move the head pointer to the next node
    }
}

// Main function to test the list functions (insert, delete, find, output)
int main() {
    int i, j, delPos, findPos, n, findResult;
    ListHeader list;    // Define a ListHeader to hold the list

    printf("Number of elements:\n");
    scanf_s("%d", &n);    // Input the number of elements in the list
    Init(&list);              // Initialize the list

    printf("Elements: \n");
    for (i = 0; i < n; i++) {
        scanf_s("%d", &j);              // Input each element
```

```
            Insert(&list, i - 1, j);        // Insert the element into the list
        }

        printf("Sequence list: ");
        Output(&list);    // Output the elements of the list
        printf("\n");

        printf("Delete element at this position: \n");
        scanf_s("%d", &delPos);                // Input the position of the element to delete
        Delete(&list, delPos);                // Delete the element at the specified position
        printf("Sequence list after deleting this element: \n");
        Output(&list);                        // Output the list after deletion
        printf("\n");

        printf("Find element at this position: \n");
        scanf_s("%d", &findPos);               // Input the position of the element to find
        Find(&list, findPos, &findResult);    // Find the element at the specified position
        printf("This element: %d", findResult);    // Output the found element
        printf("\n");

        Destroy(&list);    // Destroy the list and free the allocated memory

        return 0;              // Program ends successfully
    }
```

3、复杂度分析

(1) Init 函数

时间复杂度：O(1)

Init 函数初始化链表，分配内存并设置初始值。这是一个常数时间的操作，因此时间复杂度为 O(1)。

(2) Find 函数

时间复杂度：O(i)

Find 函数查找链表中第 i 个位置的元素。由于单链表不支持随机访问，必须从头遍历到第 i 个节点，最坏情况下需要遍历整个链表。时间复杂度为 O(i)，如果 i 是最后一个元素，时间复杂度为 O(n)，其中 n 为链表中的元素个数。

(3) Insert 函数

时间复杂度：O(i)

Insert 函数需要遍历链表，找到第 i 个位置进行插入。遍历过程的时间复杂度为 O(i)。在最坏情况下（即插入到最后一个元素或最后一个元素后面），时间复杂度为 O(n)，其中 n 为链表中的元素个数。

(4) Delete 函数

时间复杂度：O(i)

与 Insert 函数类似，Delete 函数需要遍历到第 i 个元素，这一过程的时间复杂度为 O(i)。删除节点的操作是常数时间的（调整指针并释放内存）。因此总体的时间复杂度是 O(i)，最坏情况下（删除最后一个元素），时间复杂度为 O(n)。

(5) Output 函数

时间复杂度：O(n)

Output 函数需要遍历整个链表，并输出每个元素。由于需要访问链表中的每一个节点，因此时间复杂度为 O(n)。

(6) Destroy 函数

时间复杂度：O(n)

Destroy 函数逐一释放链表中的每个节点的内存。因为它需要遍历所有 n 个节点，因此时间复杂度为 O(n)。

4、实验结果与结论

Number of elements:

10

Elements:

1 2 3 4 5 6 7 8 9 10

Sequence list: 1 2 3 4 5 6 7 8 9 10

Delete element at this position:

5

Sequence list after deleting this element: 1 2 3 4 5 7 8 9 10

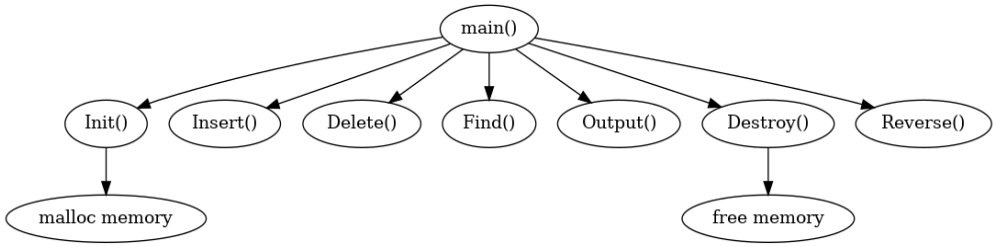
Find element at this position:

6

This element: 8

题目 3：

1、 算法设计



2、算法实现

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

typedef int ElemType;    // Define ElemType as int to represent the type of elements stored in the list nodes
typedef int Status;      // Define Status as int to represent the status of function returns (success or failure)

// Define the structure for a list node
typedef struct Node {
    ElemType element;            // The element stored in the node
    struct Node* link;           // Pointer to the next node in the list
} Node;

// Define the structure for the list header
typedef struct {
    struct Node* head;    // Pointer to the head node of the list
    int n;                // Number of elements in the list
} ListHeader;

// Function to initialize the list by creating a head node
Status Init(ListHeader* h) {
    h->head = (Node*)malloc(sizeof(Node));    // Allocate memory for the head node
    if (!h->head) {                           // Check if memory allocation was successful
        return 0;                             // Return 0 if allocation failed
    }
    h->head->link = NULL;                     // Initialize the link pointer of the head node to NULL
    h->n = 0;                                 // Initialize the number of elements in the list to
```

```
        return 1;                                   // Return 1 to indicate successful initialization
    }

    // Function to find the element at position i in the list
    Status Find(ListHeader* h, int i, ElemType* x) {
        Node* p;
        int j;
        if (i < 0 || i > h->n - 1) {    // Check if the index i is within valid range
            return 0;                   // Return 0 if the index is invalid
        }
        p = h->head->link;              // Set p to point to the first node (after the head)
        for (j = 0; j < i; j++) {       // Traverse the list to the ith node
            p = p->link;                // Move p to the next node
        }
        *x = p->element;                // Assign the element at the ith node to *x
        return 1;                       // Return 1 to indicate success
    }

    // Function to insert an element x at position i in the list
    Status Insert(ListHeader* h, int i, ElemType x) {
        Node* p, * q;
        int j;
        if (i < -1 || i > h->n - 1) {    // Check if the index i is valid for insertion
            return 0;                    // Return 0 if the index is invalid
        }
        p = h->head;                     // Set p to the head node
        for (j = 0; j <= i; j++) {       // Move p to the ith node
            p = p->link;
        }
        q = (Node*)malloc(sizeof(Node));    // Allocate memory for the new node
        q->element = x;                     // Set the new node's element to x
        q->link = p->link;                  // Set the new node's link to point to the next node
        p->link = q;                        // Link the previous node to the new node
        h->n++;                             // Increment the number of elements in the list
```

```
        return 1;                                   // Return 1 to indicate success
    }

    // Function to delete the element at position i in the list
    Status Delete(ListHeader* h, int i) {
        int j;
        Node* p, * q;
        if (!h->n) {                        // If the list is empty, return failure
            return 0;
            if (i < 0 || i > h->n - 1) {    // Check if the index i is valid for deletion
                return 0;
            }
        }
        q = h->head;                                // Set q to the head node
        for (j = 0; j < i; j++) {           // Move q to the node just before the ith node
            q = q->link;
        }
        p = q->link;                                // Set p to the ith node (the one to be deleted)
        q->link = q->link->link;            // Link the previous node to the next node, skipping the ith node
        free(p);                                    // Free the memory of the deleted node
        h->n--;                                     // Decrement the number of elements in the list
        return 1;                                   // Return 1 to indicate success
    }

    // Function to output the elements of the list
    Status Output(ListHeader* h) {
        Node* p;
        if (!h->n) {                                // If the list is empty, return failure
            return 0;
        }
        p = h->head->link;                          // Set p to the first node (after the head)
        while (p) {                                 // Traverse the list
            printf("%d ", p->element); // Print the element of the current node
            p = p->link;                            // Move to the next node
```

```
    }
    printf("\n");                          // Print a newline after the list output
    return 1;                              // Return 1 to indicate success
}

// Function to destroy the list and free all allocated memory
void Destroy(ListHeader* h) {
    Node* p, * q;
    while (h->head->link) {                // While the list is not empty
        q = h->head->link;                 // Set q to the first node
        p = h->head->link->link;           // Set p to the next node
        free(h->head->link);               // Free the memory of the current node
        h->head = q;                       // Move the head pointer to the next node
    }
}

// Function to reverse the linked list without additional storage
Status Reverse(ListHeader* h) {
    if (!h->head->link || !h->head->link->link) {
        return 1;    // If the list is empty or has only one node, no need to reverse
    }

    Node* prev = NULL;                     // Initialize the previous pointer to NULL
    Node* curr = h->head->link;            // Start from the first node (after the head)
    Node* next = NULL;                     // This will hold the next node temporarily

    while (curr) {                         // Traverse the list
        next = curr->link;                 // Store the next node
        curr->link = prev;                 // Reverse the link
        prev = curr;                       // Move prev to the current node
        curr = next;                       // Move curr to the next node
    }

    h->head->link = prev;                  // Set the new head of the reversed list
    return 1;                              // Return success
```

```
}

// Main function to test the list functions (insert, delete, find, output)
int main() {
    int i, j, delPos, findPos, n, findResult;
    ListHeader list;    // Define a ListHeader to hold the list

    printf("Number of elements:\n");
    scanf_s("%d", &n);    // Input the number of elements in the list
    Init(&list);              // Initialize the list

    printf("Elements: \n");
    for (i = 0; i < n; i++) {
        scanf_s("%d", &j);                // Input each element
        Insert(&list, i - 1, j);       // Insert the element into the list
    }

    printf("Sequence list: ");
    Output(&list);    // Output the elements of the list
    printf("\n");

    printf("Delete element at this position: \n");
    scanf_s("%d", &delPos);              // Input the position of the element to delete
    Delete(&list, delPos);                // Delete the element at the specified position
    printf("Sequence list after deleting this element: ");
    Output(&list);                        // Output the list after deletion
    printf("\n");

    printf("Find element at this position: \n");
    scanf_s("%d", &findPos);              // Input the position of the element to find
    Find(&list, findPos, &findResult);    // Find the element at the specified position
    printf("This element: %d", findResult);    // Output the found element
    printf("\n");

    // Reverse the list
```

```
    Reverse(&list);
    printf("\nSequence list after reversing: \n");
    Output(&list);    // Output the reversed list

    Destroy(&list);    // Destroy the list and free the allocated memory

    return 0;              // Program ends successfully
}
```

3、复杂度分析

Reverse 函数

时间复杂度：O(n)

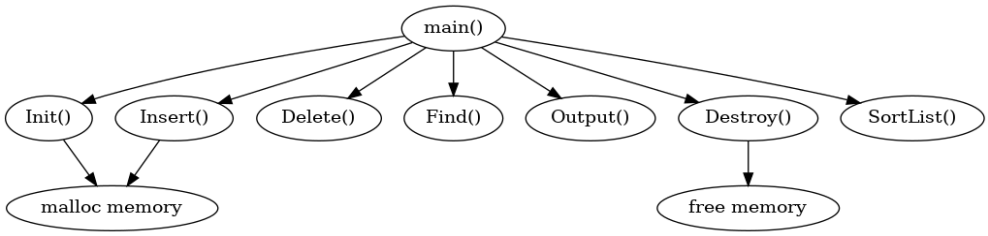
对于链表中的每一个节点，需要执行一次指针反转操作。假设链表中有 n 个节点，Reverse 函数需要遍历整个链表一次，操作每个节点。因此，Reverse 函数的时间复杂度为 O(n)。

4、实验结果与结论

```
Number of elements:
10
Elements:
1 2 3 4 5 6 7 8 9 10
Sequence list: 1 2 3 4 5 6 7 8 9 10

Delete element at this position:
5
Sequence list after deleting this element: 1 2 3 4 5 7 8 9 10

Find element at this position:
6
This element: 8

Sequence list after reversing:
10 9 8 7 5 4 3 2 1
```

题目 4：

1、 算法设计



2、算法实现

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

typedef int ElemType;    // Define ElemType as int to represent the type of elements stored in the list nodes
typedef int Status;      // Define Status as int to represent the status of function returns (success or failure)

// Define the structure for a list node
typedef struct Node {
    ElemType element;            // The element stored in the node
    struct Node* link;           // Pointer to the next node in the list
} Node;

// Define the structure for the list header
typedef struct {
    struct Node* head;    // Pointer to the head node of the list
    int n;                // Number of elements in the list
} ListHeader;

// Function to initialize the list by creating a head node
Status Init(ListHeader* h) {
    h->head = (Node*)malloc(sizeof(Node));    // Allocate memory for the head node
    if (!h->head) {                           // Check if memory allocation was successful
        return 0;                             // Return 0 if allocation failed
    }
    h->head->link = NULL;                     // Initialize the link pointer of the head node to NULL
```

```
        h->n = 0;                                   // Initialize the number of elements in the list to
0
        return 1;                                   // Return 1 to indicate successful initialization
    }

    // Function to find the element at position i in the list
    Status Find(ListHeader* h, int i, ElemType* x) {
        Node* p;
        int j;
        if (i < 0 || i > h->n - 1) {    // Check if the index i is within valid range
            return 0;                   // Return 0 if the index is invalid
        }
        p = h->head->link;              // Set p to point to the first node (after the head)
        for (j = 0; j < i; j++) {       // Traverse the list to the ith node
            p = p->link;                // Move p to the next node
        }
        *x = p->element;                // Assign the element at the ith node to *x
        return 1;                       // Return 1 to indicate success
    }

    // Function to insert an element x at position i in the list
    Status Insert(ListHeader* h, int i, ElemType x) {
        Node* p, * q;
        int j;
        if (i < -1 || i > h->n - 1) {    // Check if the index i is valid for insertion
            return 0;                    // Return 0 if the index is invalid
        }
        p = h->head;                     // Set p to the head node
        for (j = 0; j <= i; j++) {       // Move p to the ith node
            p = p->link;
        }
        q = (Node*)malloc(sizeof(Node));    // Allocate memory for the new node
        q->element = x;                     // Set the new node's element to x
        q->link = p->link;                  // Set the new node's link to point to the next node
        p->link = q;                        // Link the previous node to the new node
```

```
        h->n++;                                     // Increment the number of elements in the list
        return 1;                                   // Return 1 to indicate success
    }

    // Function to delete the element at position i in the list
    Status Delete(ListHeader* h, int i) {
        int j;
        Node* p, * q;
        if (!h->n) {                    // If the list is empty, return failure
            return 0;
        }
        if (i < 0 || i > h->n - 1) {    // Check if the index i is valid for deletion
            return 0;
        }
        q = h->head;                    // Set q to the head node
        for (j = 0; j < i; j++) {       // Move q to the node just before the ith node
            q = q->link;
        }
        p = q->link;                    // Set p to the ith node (the one to be deleted)
        q->link = q->link->link;        // Link the previous node to the next node, skipping the ith
node
        free(p);                        // Free the memory of the deleted node
        h->n--;                         // Decrement the number of elements in the list
        return 1;                       // Return 1 to indicate success
    }

    // Function to output the elements of the list
    Status Output(ListHeader* h) {
        Node* p;
        if (!h->n) {                    // If the list is empty, return failure
            return 0;
        }
        p = h->head->link;              // Set p to the first node (after the head)
        while (p) {                     // Traverse the list
            printf("%d ", p->element); // Print the element of the current node
```

```
            p = p->link;                    // Move to the next node
        }
        printf("\n");                       // Print a newline after the list output
        return 1;                           // Return 1 to indicate success
    }

    // Function to sort the linked list using insertion sort
    void SortList(ListHeader* h) {
        if (!h->n) return;    // If the list is empty, no need to sort

        Node* sorted = NULL;    // Sorted linked list
        Node* current = h->head->link;    // Start with the first element (after head)

        while (current) {
            Node* next = current->link;    // Store the next node to process
            // Insert current into the sorted linked list
            if (!sorted || sorted->element >= current->element) {
                // Insert at the beginning or before the first element in the sorted list
                current->link = sorted;
                sorted = current;
            }
            else {
                // Find the correct position in the sorted part of the list
                Node* temp = sorted;
                while (temp->link && temp->link->element < current->element) {
                    temp = temp->link;
                }
                current->link = temp->link;
                temp->link = current;
            }
            current = next;    // Move to the next node in the original list
        }
        h->head->link = sorted;    // Update the head to point to the new sorted list
    }
```

```
// Function to destroy the list and free all allocated memory
void Destroy(ListHeader* h) {
    Node* p, * q;
    while (h->head->link) {             // While the list is not empty
        q = h->head->link;              // Set q to the first node
        p = h->head->link->link;        // Set p to the next node
        free(h->head->link);            // Free the memory of the current node
        h->head = q;                    // Move the head pointer to the next node
    }
}

// Main function to test the list functions (insert, delete, find, output, and sort)
int main() {
    int i, j, delPos, findPos, n, findResult;
    ListHeader list;    // Define a ListHeader to hold the list

    printf("Number of elements:\n");
    scanf_s("%d", &n);    // Input the number of elements in the list
    Init(&list);              // Initialize the list

    printf("Elements: \n");
    for (i = 0; i < n; i++) {
        scanf_s("%d", &j);              // Input each element
        Insert(&list, i - 1, j);        // Insert the element into the list
    }

    printf("Sequence list: ");
    Output(&list);    // Output the unsorted list

    printf("\nDelete element at this position: \n");
    scanf_s("%d", &delPos);             // Input the position of the element to delete
    Delete(&list, delPos);              // Delete the element at the specified position
    printf("Sequence list after deleting this element: \n");
    Output(&list);                      // Output the list after deletion
    printf("\n");
```

```
    printf("Find element at this position: \n");
    scanf_s("%d", &findPos);              // Input the position of the element to find
    Find(&list, findPos, &findResult);    // Find the element at the specified position
    printf("This element: %d", findResult);    // Output the found element
    printf("\n");

    // Sort the list
    SortList(&list);
    printf("\nSequence list after sorting: ");
    Output(&list);    // Output the sorted list

    Destroy(&list);    // Destroy the list and free the allocated memory

    return 0;              // Program ends successfully
}
```

3、复杂度分析

SortList 函数

时间复杂度：$O(n^2)$

在 SortList 函数中，假设链表中有 n 个节点。为了对链表进行排序，需要多次遍历链表。在最坏情况下，对于每个节点，需要比较和交换它与其他节点的值，这涉及 n 次遍历，每次遍历时需要访问接下来还未排序的所有节点，最多为 n - 1 个节点。因此，比较和交换操作会在整个链表上进行 n 次，每次需要遍历大约 n 次节点。

4、实验结果与结论

```
Number of elements:
10
Elements:
1 3 5 7 9 0 2 4 6 8
Sequence list: 1 3 5 7 9 0 2 4 6 8

Delete element at this position:
5
Sequence list after deleting this element:
1 3 5 7 9 2 4 6 8
```

Find element at this position:

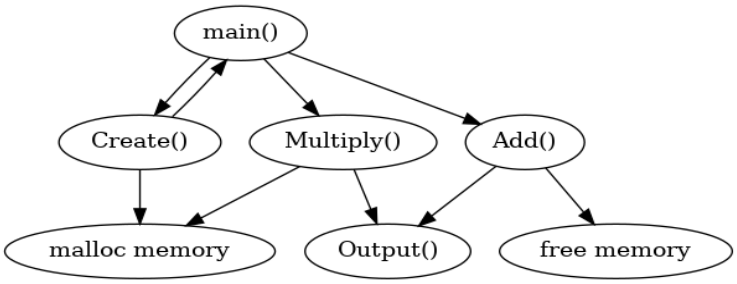
6

This element: 4

Sequence list after sorting: 1 2 3 4 5 6 7 8 9

题目 5：

1、 算法设计



2、算法实现

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

// Define a structure for polynomial nodes
typedef struct PNode {
    int coef; // Coefficient of the polynomial term
    int exp;    // Exponent of the polynomial term
    struct PNode* link; // Pointer to the next node
} PNode;

// Define a structure for polynomial
typedef struct {
    struct PNode* head; // Head pointer of the polynomial
} Polynominal;

// Function to create a polynomial
void Create(Polynominal* p) {
    PNode* pn, * pre, * q; // Pointers for new node, previous node, and current node
    p->head = (PNode*)malloc(sizeof(PNode)); // Allocate head node
    p->head->exp = -1; // Initialize head node exponent
    p->head->link = p->head; // Point to itself
```

```
printf("Enter the number of terms in the expression:\n");
int nn = 0;
scanf_s("%d", &nn); // Read number of terms

// Loop to read each term of the polynomial
for (int i = 0; i < nn; i++) {
    pn = (PNode*)malloc(sizeof(PNode)); // Allocate new node
    printf("\nEnter the coefficient:\n");
    scanf_s("%d", &pn->coef); // Read coefficient
    printf("Enter the exponent:\n");
    scanf_s("%d", &pn->exp); // Read exponent

    if (pn->exp == -1) break; // Stop if exponent is -1

    pre = p->head; // Set previous node to head
    q = p->head->link; // Set current node to the first node

    // Find the correct position to insert the new node
    while (q != p->head && q->exp > pn->exp) {
        pre = q; // Move previous pointer
        q = q->link; // Move current pointer
    }

    // If the exponent already exists, combine coefficients
    if (q != p->head && q->exp == pn->exp) {
        q->coef += pn->coef; // Combine coefficients
        if (q->coef == 0) { // If coefficient becomes zero
            pre->link = q->link; // Remove the node
            free(q); // Free memory
        }
        free(pn); // Free temporary node
    }
    else { // Insert new node
        pn->link = q; // Link new node to current node
```

```
                pre->link = pn; // Link previous node to new node
            }
        }
    }

    // Function to output the polynomial
    void Output(Polynominal p) {
        printf("expression = ");
        PNode* q; // Pointer for traversal
        int flag = 1; // Flag for formatting
        q = p.head->link; // Start from the first node
        if (!q) { // If the list is empty
            return;
        }
        // Loop to print each term of the polynomial
        while (q != p.head) {
            if (!flag && (q->coef > 0)) printf("+"); // Print '+' for positive coefficients
            flag = 0; // Reset flag after first term
            if (q->coef == 0) { // Skip if coefficient is zero
                return;
            }
            if (q->coef != 1) { // Print coefficient if not 1
                printf("%d", q->coef);
            }

            // Print exponent part
            switch (q->exp) {
            case 0: break; // No variable for exponent 0
            case 1: printf("X"); break; // Print X for exponent 1
            default: printf("X^%d", q->exp); break; // Print X raised to exponent
            }
            q = q->link; // Move to the next node
        }
        printf("\n"); // New line after output
    }
```

```
// Function to add two polynomials
void Add(Polynominal* px, Polynominal* qx) {
    PNode* p, * q, * q1 = qx->head, * temp; // Pointers for traversal
    p = px->head->link; // Start from the first node of first polynomial
    q = qx->head->link; // Start from the first node of second polynomial

    // Loop through first polynomial
    while (p != px->head) {
        q1 = qx->head; // Reset q1 to the head of second polynomial
        q = q1->link; // Start from the first node of second polynomial

        // Find position to add or combine terms
        while (q != qx->head && q->exp > p->exp) {
            q1 = q; // Move previous pointer
            q = q->link; // Move current pointer
        }

        // Combine coefficients if exponents match
        if (q != qx->head && p->exp == q->exp) {
            q->coef += p->coef; // Combine coefficients
            if (q->coef == 0) { // Remove node if coefficient is zero
                q1->link = q->link; // Link previous to next node
                free(q); // Free memory
                q = q1->link; // Update current pointer
            }
        }
        else { // Insert new term
            temp = (PNode*)malloc(sizeof(PNode)); // Allocate new node
            temp->coef = p->coef; // Set coefficient
            temp->exp = p->exp; // Set exponent
            temp->link = q1->link; // Link new node to next
            q1->link = temp; // Link previous to new node
        }
        p = p->link; // Move to the next node in first polynomial
```

```
        }

        // Combine duplicate exponents in the result
        q1 = qx->head; // Reset to head
        q = q1->link; // Start from the first node
        while (q != qx->head && q->link != qx->head) {
            if (q->exp == q->link->exp) { // If exponents match
                q->coef += q->link->coef; // Combine coefficients
                PNode* duplicate = q->link; // Duplicate node to be removed
                q->link = duplicate->link; // Link to next node
                free(duplicate); // Free memory
            }
            else {
                q1 = q; // Move previous pointer
                q = q->link; // Move current pointer
            }
        }
    }

    // Function to multiply two polynomials
    void Multiply(Polynominal* px, Polynominal* qx) {
        Polynominal qx1, qx2; // Temporary polynomials for multiplication
        PNode* q1, * q2, * q3, * q4, * pre = (PNode*)malloc(sizeof(PNode)), * q; // Pointers for nodes
        qx1.head = (PNode*)malloc(sizeof(PNode)); // Allocate head for qx1
        qx1.head->exp = -1; // Initialize exponent
        qx1.head->link = qx1.head; // Point to itself
        q1 = px->head; // Start from the first node of first polynomial
        q2 = qx->head; // Start from the first node of second polynomial

        // Loop to multiply each term in the first polynomial with the second polynomial
        while (q2->exp != -1) {
            q3 = (PNode*)malloc(sizeof(PNode)); // Allocate new node for result
            q3->coef = q1->coef * q2->coef; // Multiply coefficients
            q3->exp = q1->exp + q2->exp; // Add exponents
```

```
        // Insert the new node into the result polynomial
        if (qx1.head->link->exp == -1) {
            q3->link = qx1.head->link; // Link to head
            qx1.head->link = q3; // Set as first node
            pre = qx1.head->link; // Update previous pointer
        }
        else {
            q3->link = qx1.head; // Link to head
            pre->link = q3; // Link previous to new node
            pre = pre->link; // Update previous pointer
        }

        q2 = q2->link; // Move to the next node in second polynomial
    }
    q1 = q1->link; // Move to the next node in first polynomial

    // Loop through remaining terms in first polynomial
    while (q1->exp != -1) {
        q2 = q2->link; // Reset to start of second polynomial
        qx2.head = (PNode*)malloc(sizeof(PNode)); // Allocate head for temporary polynomial
        qx2.head->exp = -1; // Initialize exponent
        qx2.head->link = qx2.head; // Point to itself

        // Multiply current term of first polynomial with all terms of the second polynomial
        while (q2->exp != -1) {
            q4 = (PNode*)malloc(sizeof(PNode)); // Allocate new node for result
            q4->coef = q1->coef * q2->coef; // Multiply coefficients
            q4->exp = q1->exp + q2->exp; // Add exponents
            // Insert the new node into the temporary polynomial
            if (qx2.head->link->exp == -1) {
                q4->link = qx2.head->link; // Link to head
                qx2.head->link = q4; // Set as first node
                pre = qx2.head->link; // Update previous pointer
            }
            else {
```

```
                    q4->link = qx2.head; // Link to head
                    pre->link = q4; // Link previous to new node
                    pre = pre->link; // Update previous pointer
                }
                q2 = q2->link; // Move to the next node in second polynomial
            }
            Add(&qx2, &qx1); // Add temporary result to overall result
            q1 = q1->link; // Move to the next node in first polynomial
        }
        Output(qx1); // Output the final result
}

// Main function
int main() {
        Polynominal p, q; // Declare two polynomial variables
        int x; // Variable for user choice
        printf("Enter the first expression:\n");
        Create(&p); // Create the first polynomial
        Output(p); // Output the first polynomial
        printf("\n\nEnter the second expression:\n");
        Create(&q); // Create the second polynomial
        Output(q); // Output the second polynomial
        printf("\nEnter your choice:\n1: Addition      2: Multiplication\n");
        scanf_s("%d", &x); // Read user choice
        switch (x) {
        case 0: break; // No action for choice 0
        case 1: printf("\nAddition result:\n");
            Add(&p, &q); // Add the polynomials
            Output(q); // Output the result
            break;
        case 2: printf("\nMultiplication result:\n");
            Multiply(&p, &q); // Multiply the polynomials
            break;
        default: break; // No action for invalid choice
        }
```

return 0; // Exit the program

}

3、复杂度分析

(1)Create 函数

时间复杂度：O(n)

函数读取用户输入的 n 项多项式。每一项的系数和指数处理时间为常数时间，可能需要遍历链表以找到插入位置，因此总复杂度为 O(n)。

(2)Output 函数

时间复杂度：O(n)

函数从头遍历链表，输出每一项的多项式。每一项的处理时间为常数，导致线性遍历，复杂度为 O(n)。

(3)Add 函数

时间复杂度：O(m*n)

如果第一个多项式有 m 项，第二个多项式有 n 项，函数需要为每一项在第二个多项式中查找合并位置。合并和插入新项的时间与结果中的现有项数成线性关系，因此整体复杂度为 O(m*n)。

(4)Multiply 函数

时间复杂度：O(m*n+k)

乘法涉及两个嵌套循环：对于第一个多项式中的每一项，乘以第二个多项式中的每一项，总复杂度为 O(m*n)。之后，Add 函数将结果合并，可能增加 O(k)的复杂度，其中 k 是中间结果中的项数。因此，整体复杂度可近似为 O(m*n)。

4、实验结果与结论

Enter the first expression:

Enter the number of terms in the expression:

5

Enter the coefficient:

3

Enter the exponent:

4

Enter the coefficient:

2

Enter the exponent:

2

```
Enter the coefficient:
6
Enter the exponent:
2

Enter the coefficient:
7
Enter the exponent:
1

Enter the coefficient:
8
Enter the exponent:
0
expression = 3X^4+8X^2+7X+8



Enter the second expression:
Enter the number of terms in the expression:
3

Enter the coefficient:
4
Enter the exponent:
3

Enter the coefficient:
2
Enter the exponent:
1

Enter the coefficient:
8
Enter the exponent:
0
```

```
expression = 4X^3+2X+8

Enter your choice:
1: Addition      2: Multiplication
2

Multiplication result:
expression = 12X^7+38X^5+52X^4+48X^3+78X^2+72X+64
```

四、实验小结（包括问题和解决方法、心得体会、意见与建议等）

(一)实验中遇到的主要问题及解决方法

1.问题：顺序表插入操作时数组越界

解决方法：初次运行程序时，插入操作导致数组越界。通过增加插入位置的边界检查，确保插入位置合法后，问题得到解决。

2.问题：链表节点删除后内存泄漏

解决方法：链表操作中，删除节点后没有及时释放内存，导致内存泄漏。经过调试，发现是指针处理不当。通过在删除节点后及时释放相关内存，成功解决问题。

3.问题：链表遍历时出现空指针异常

解决方法：在遍历链表时，某些节点由于错误的指针连接导致空指针访问。通过在遍历过程中增加空指针检查，并修正指针连接逻辑，避免了程序崩溃。

4.问题：多项式链表合并时重复项处理不当

解决方法：在实现多项式合并时，对于相同指数的项没有正确处理，导致系数叠加错误。通过在合并过程中添加相同指数项的判定逻辑，确保系数合并正确。

（二）实验心得

本次实验使我深入理解了顺序表与链表的存储结构及其操作方法。通过亲自动手编写代码，我不仅加深了对这些数据结构的掌握，还提高了调试和解决问题的能力。尤其是在链表操作中，指针的管理要求极高，这促使我对指针的使用有了更加清晰的认识。此外，算法设计的合理性和高效性也是本次实验中需要重点考虑的部分。

（三）意见与建议（没有可省略）

可以提供更多的时间上机操作，以确保更多程序设计思路得以实现，提升面向对象语言的掌握程度和编程能力。

五、支撑毕业要求指标点

《数据结构》课程支撑毕业要求的指标点为：

1.2-M 掌握计算机软硬件相关工程基础知识，能将其用于分析计算机及应用领域的相关工程问题。

3.2-H 能够根据用户需求，选取适当的研究方法和技术手段，确定复杂工程问题的解决方案。

4.2-H 能够根据实验方案，配置实验环境、开展实验，使用定性或定量分析方法进行数据分析与处理，综合实验结果以获得合理有效的结论。

实验内容	支撑点 1.2	支撑点 3.2	支撑点 4.2
线性表及多项式的运算	√		
二叉树的基本操作及哈夫曼编码译码系统的实现		√	√
图的基本运算及智能交通中的最佳路径选择问题		√	√
各种内排序算法的实现及性能比较	√		√

六、指导教师评语 (含学生能力达成度的评价)

如评分细则所示

成　绩	XX	批阅人	XX	日　期	XXX

评分细则	评分项	优秀	良好	中等	合格	不合格
评分细则	遵守实验室规章制度					
	学习态度					
	算法思想准备情况					
	程序设计能力					
	解决问题能力					
	课题功能实现情况					
	算法设计合理性					
	算法效能评价					
	回答问题准确度					
	报告书写认真程度					
	内容详实程度					
	文字表达熟练程度					
	其它评价意见					
	本次实验能力达成评价（总成绩）					