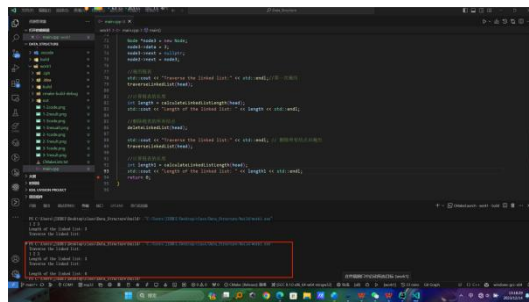


1. 单链表的建立:

RESULT:



CODE:

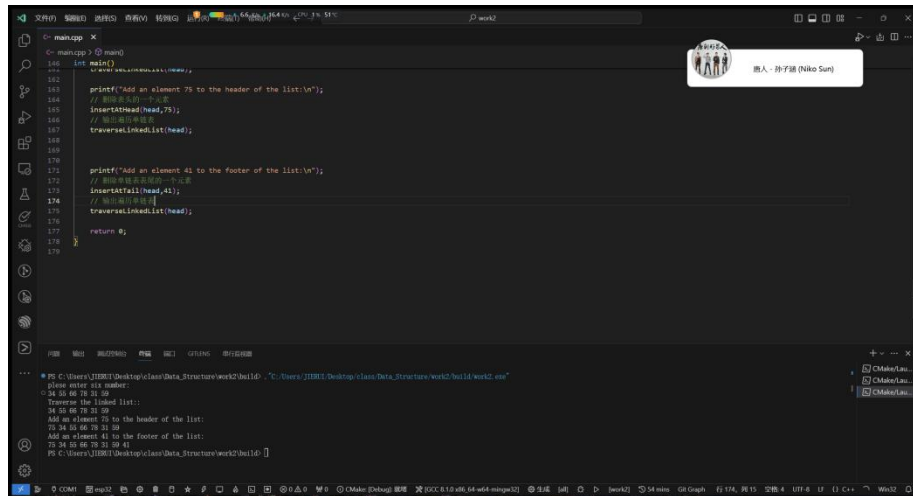
```

1 #include <iostream>
2 // 定义链表的结点结构体
3 struct Node
4 {
5     int data;
6     Node *next;
7 };
8 /**
9  * @brief 遍历链表
10  *
11  * @param head 链表头结点
12  */
13 void traverseLinkedList(Node *head)
14 {
15     Node *current = head;
16     while (current != nullptr) // 当前结点非空
17     {
18         std::cout << current->data << " ";
19         current = current->next;
20     }
21     std::cout << std::endl;
22 }
23 /**
24  * @brief 计算链表的长度
25  *
26  * @param head 链表头结点
27  * @return int 链表长度
28  */
29 int calculateLinkedListLength(Node *head)
30 {
31     int length = 0;
32     Node *current = head;
33     while (current != nullptr)
34     {
35         length++; // 长度自增
36         current = current->next; // 指针指向下一个结点
37     }
38     return length;
39 }
40 /**
41  * @brief 删除链表的所有结点
42  *
43  * @param head 链表头结点
44  */
45 void deleteLinkedList(Node *&head)
46 {
47     Node *current = head;
48     while (current != nullptr)
49     {
50         Node *temp = current;
51         current = current->next; // 指针指向下一个结点
52         delete temp;
53     }
54     head = nullptr;
55 }
56
57 int main()
58 {
59     Node *head = nullptr;
60
61     // 建立一个单链表
62     Node *node1 = new Node;
63     node1->data = 1;
64     node1->next = nullptr;
65     head = node1;
66
67     Node *node2 = new Node;
68     node2->data = 2;
69     node2->next = nullptr;
70     node1->next = node2;
71
72     Node *node3 = new Node;
73     node3->data = 3;
74     node3->next = nullptr;
75     node2->next = node3;
76
77     // 遍历链表
78     std::cout << "Traverse the linked list:" << std::endl; // 第一次遍历
79     traverseLinkedList(head);
80
81     // 计算链表的长度
82     int length = calculateLinkedListLength(head);
83     std::cout << "Length of the linked list: " << length << std::endl;
84
85     // 删除链表的所有结点
86     deleteLinkedList(head);
87
88     std::cout << "Traverse the linked list:" << std::endl; // 删除所有结点后遍历
89     traverseLinkedList(head);
90
91     // 计算链表的长度
92     int length1 = calculateLinkedListLength(head);
93     std::cout << "Length of the linked list: " << length1 << std::endl;
94     return 0;
95 }
96

```

2. 单链表的插入:

RESULT:



```
140 int main()
141 {
142     // Create a singly linked list
143     print("Add an element 75 to the header of the list.\n");
144     // Insert at head
145     InsertAtHead(head, 75);
146     // Output the singly linked list
147     traverseLinkedList(head);
148
149     print("Add an element 41 to the footer of the list.\n");
150     // Insert at tail
151     InsertAtTail(head, 41);
152     // Output the singly linked list
153     traverseLinkedList(head);
154     return 0;
155 }
```

Output:

```
PS C:\Users\JH861\Desktop\class_Data_Structure\work2\build> ".\work2.exe"
please enter the number:
traverse the linked list::
34 55 66 78 31 59
Add an element 75 to the header of the list:
75 34 55 66 78 31 59
Add an element 41 to the footer of the list:
75 34 55 66 78 31 59 41
PS C:\Users\JH861\Desktop\class_Data_Structure\work2\build>
```

CODE:

```

class Node {
// 存储节点的数据域,第一个元素为节点数据域指向下一个节点的指针
struct Node {
{
int data;
Node *next;
};
};

// Node 类为单链表
// Node head 存储单链表的头指针

void TraverseLinkList(Node *head)
{
Node *current = head;
while (current != nullptr)
{
std::cout << current->data << " "; // 输出当前节点的数据域,并换行
current = current->next; // 让current指向下一个节点
}
std::cout << std::endl; // 遍历到head执行

// Node 向单链表添加数据域为一个元素
// Node head 存储单链表的头指针
// Node value 存储添加元素

void InsertAtHead(Node *head, int value)
{
Node *NodeNode = new Node();
NodeNode->data = value; // 当前节点的数据域赋值value
NodeNode->next = head; // 将原节点head的next置空
head = NodeNode; // 将原节点head指向新添加的头节点
head = NodeNode;

// Node 向单链表添加数据域为一个元素
// Node head 存储单链表的头指针
// Node value 存储添加元素

void InsertAtTail(Node *head, int value)
{
Node *NodeNode = new Node();
NodeNode->data = value; // 当前节点的数据域赋值value
NodeNode->next = nullptr; // 将原节点的next置空
if (head == nullptr) // 如果为空链表,直接将该节点作为头节点
{
head = NodeNode;
}
else
{
Node *current = head;
while (current->next != nullptr)
{
current = current->next;
}
current->next = NodeNode; // 将原节点next指向下一个节点并返回
}
}

// Node 删除单链表的数据域元素
// Node head 存储单链表的头指针

void DeleteAtHead(Node *head)
{
if (head == nullptr) // 如果为空链表,直接返回
{
return;
}
Node *temp = head; // 将单链表的头指针赋值
head = head->next; // 头指针指向下一个节点
delete temp; // 删除原表的头节点

// Node 删除单链表的数据域元素
// Node head 存储单链表的头指针

void DeleteAtTail(Node *head)
{
if (head == nullptr) // 如果为空链表,直接返回
{
return;
}
if (head->next == nullptr) // 如果只有一个节点,直接删除
{
delete head;
head = nullptr;
return;
}
Node *current = head;
while (current->next->next != nullptr) // 找到倒数第二个节点
{
current = current->next;
}
current->next = nullptr; // 将倒数第二个节点
current->next = nullptr; // 将倒数第二个节点的next置空

// Node 删除单链表中间数据域元素
// Node head 存储单链表的头指针
// Node value 存储删除元素

void DeleteAtIndex(Node *head, int value)
{
if (head == nullptr) // 如果为空链表,直接返回
{
return;
}
// 如果删除数据域为头节点
if (head->data == value)
{
deleteAtHead(head); // 直接删除头节点并返回
}
Node *current = head; // 指向头节点,将单链表的头节点赋值
Node *previous = nullptr; // 指向头节点的前一个节点

// 删除单链表中间数据域的元素
while (current != nullptr && current->data != value) // 当前节点为头节点且数据域不等于value,则指向下一个节点
{
previous = current;
current = current->next;
}
// 如果删除数据域的元素
if (current != nullptr)
{
previous->next = current->next; // 将当前节点的next指向next节点的next节点并返回
delete current; // 删除当前节点,并删除数据域元素
}
}

int main()
{
// Node head 存储单链表的头指针,头指针为nullptr
Node *head = nullptr; // 存储单链表的头指针,头指针为nullptr
// 添加头节点(头节点, 存储数据域为单链表)
printf("Enter enter its number:\n");
for (int i = 0; i < 4; i++)
{
int data;
std::cout << "data: ";
scanf("%d", &data);
InsertAtTail(head, data); // 在单链表末尾添加元素
}

// 删除数据域元素
printf("Traverse the linked list:\n");
TraverseLinkList(head);

printf("Add an element 75 to the header of the list:\n");
InsertAtHead(head, 75);
TraverseLinkList(head);

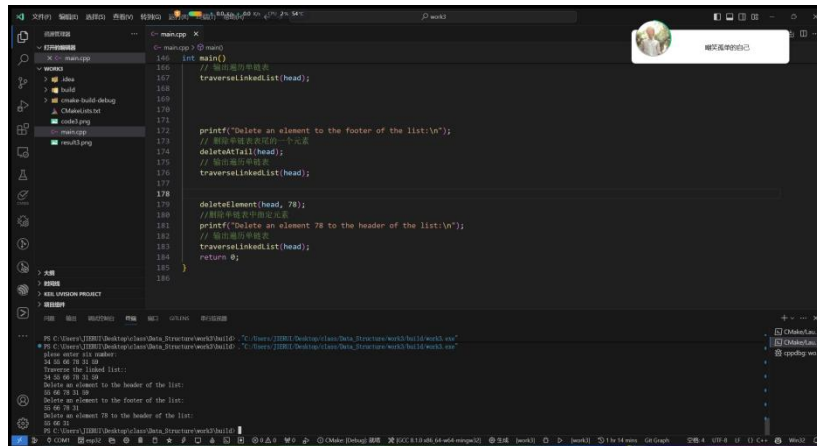
// 删除数据域元素
printf("Delete an element 41 to the footer of the list:\n");
DeleteAtTail(head, 41);
TraverseLinkList(head);

return 0;
}

```

3. 单链表的删除:

RESULT:



```
166 int main()
167 {
168     // 输出遍历单链表
169     traverseLinkedList(head);
170
171     printf("Delete an element to the footer of the list:\n");
172     // 删除单链表的最后一个元素
173     deleteAtTail(head);
174     // 输出遍历单链表
175     traverseLinkedList(head);
176
177     deleteElement(head, 78);
178     // 删除单链表中指定元素
179     printf("Delete an element 78 to the header of the list:\n");
180     // 输出遍历单链表
181     traverseLinkedList(head);
182     return 0;
183 }
```

Output:

```
Traverse the linked list:
14 46 66 78 21 39
Delete an element to the footer of the list:
14 46 78 21
Delete an element 78 to the header of the list:
14 46 21
```

CODE:

```

1 #include <iostream>
2 // 链表节点的定义结构体, 第一个节点包含数据和指向下一个节点的指针
3 struct Node
4 {
5     int data;
6     Node *next;
7 };
8 /**
9  * @brief 遍历单链表
10  *
11  * @param head 单链表的头指针
12  */
13 void traverseLinkedList(Node *head)
14 {
15     Node *current = head;
16     while (current != nullptr)
17     {
18         std::cout << current->data << " "; // 输出当前节点的数据, 并换
19         current = current->next; // 将current指向下一个节点
20     }
21     std::cout << std::endl; // 输出std::endl换行
22 }
23 /**
24  * @brief 向单链表的表头添加一个元素
25  *
26  * @param head 单链表的头指针
27  * @param value 要添加的元素
28  */
29 void insertAtHead(Node *head, int value)
30 {
31     Node *newNode = new Node();
32     newNode->data = value; // 将新节点的数据域赋值为value
33     newNode->next = head; // 将新节点的next指针指向原来的头节点
34     head = newNode;
35 }
36 /**
37  * @brief 向单链表的表尾添加一个元素
38  *
39  * @param head 单链表的头指针
40  * @param value 要添加的元素
41  */
42 void insertAtTail(Node *head, int value)
43 {
44     Node *newNode = new Node();
45     newNode->data = value; // 将新节点的数据域赋值为value
46     newNode->next = nullptr; // 将新节点的next指针置空
47     if (head == nullptr) // 如果为空链表, 直接将新节点作为头节点
48     {
49         head = newNode;
50     }
51     else
52     {
53         Node *current = head;
54         while (current->next != nullptr)
55         {
56             current = current->next;
57         }
58         current->next = newNode; // 将新节点添加到最后一个节点的后面
59     }
60 }
61 /**
62  * @brief 删除单链表的头元素
63  *
64  * @param head 单链表的头指针
65  */
66 void deleteHead(Node *head)
67 {
68     if (head == nullptr) // 如果为空链表, 直接返回
69     {
70         return;
71     }
72     Node *temp = head; // 保存要删除的节点的地址
73     head = head->next; // 头指针指向下一个节点
74     delete temp; // 释放旧的头节点
75 }
76 /**
77  * @brief 删除单链表的表尾元素
78  *
79  * @param head 单链表的头指针
80  */
81 void deleteTail(Node *head)
82 {
83     if (head == nullptr) // 如果为空链表, 直接返回
84     {
85         return;
86     }
87     if (head->next == nullptr) // 如果只有一个节点, 直接删除
88     {
89         delete head;
90         head = nullptr;
91         return;
92     }
93     Node *current = head;
94     while (current->next->next != nullptr) // 找到倒数第二个节点
95     {
96         current = current->next;
97     }
98     delete current->next; // 删除最后一个节点
99     current->next = nullptr; // 将倒数第二个节点的next指针置空
100 }
101 /**
102  * @brief 删除单链表中的指定元素
103  *
104  * @param head 单链表的头指针
105  * @param value 要删除的元素
106  */
107 void deleteElement(Node *head, int value)
108 {
109     if (head == nullptr) // 如果为空链表, 直接返回
110     {
111         return;
112     }
113     // 如果要删除的元素是头节点
114     if (head->data == value)
115     {
116         deleteHead(head);
117         return;
118     }
119     Node *current = head;
120     Node *previous = nullptr;
121     // 遍历链表直到找到要删除的元素
122     while (current != nullptr && current->data != value)
123     {
124         previous = current;
125         current = current->next;
126     }
127     // 如果找到要删除的元素
128     if (current != nullptr)
129     {
130         previous->next = current->next;
131         delete current;
132     }
133 }
134 int main()
135 {
136     Node *head = nullptr; // 单链表的头指针, 初始化为nullptr
137     // 从键盘上读取5个整数, 并依次保存在单链表中
138     print("Please enter 5 numbers:\n");
139     for (int i = 0; i < 5; i++)
140     {
141         int num;
142         std::cin >> num;
143         insertAtTail(head, num); // 在单链表的表尾添加元素
144     }
145     // 遍历单链表
146     print("Traverse the linked list:\n");
147     traverseLinkedList(head);
148     print("Delete an element to the header of the list.\n");
149     // 删除头元素
150     deleteHead(head);
151     // 遍历单链表
152     traverseLinkedList(head);
153     print("Delete an element to the footer of the list.\n");
154     // 删除尾元素
155     deleteTail(head);
156     // 遍历单链表
157     traverseLinkedList(head);
158     deleteElement(head, 78);
159     print("Delete an element 78 to the header of the list.\n");
160     // 删除指定元素
161     deleteElement(head, 78);
162     // 遍历单链表
163     traverseLinkedList(head);
164     return 0;
165 }

```

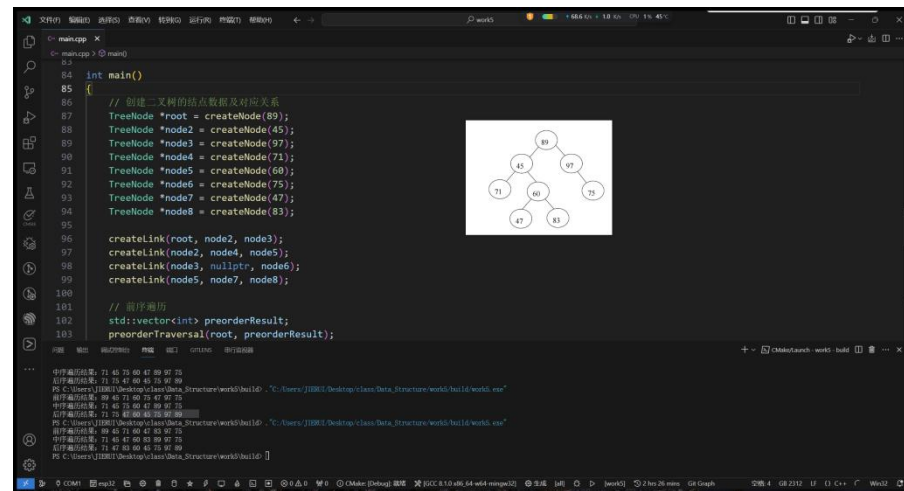
RESULT:

[illegible]

CODE:

[illegible]

RESULT:



CODE:

```

1 #include <iostream>
2 #include <vector>
3 // 树的结构体
4 struct TreeNode
5 {
6     int data; // 数据
7     TreeNode *left; // 左子节点
8     TreeNode *right; // 右子节点
9 };
10
11 /**
12  * @brief 创建一个二叉树节点
13  * @param data 节点数据
14  * @return 新创建的节点指针
15  */
16 TreeNode *createNode(int data)
17 {
18     TreeNode *newNode = new TreeNode();
19     if (newNode)
20     {
21         newNode->data = data;
22         newNode->left = newNode->right = nullptr;
23     }
24     return newNode;
25 }
26
27 /**
28  * @brief 创建节点之间的链接
29  * @param parent 父节点
30  * @param leftChild 左子节点
31  * @param rightChild 右子节点
32  */
33 void createLink(TreeNode *parent, TreeNode *leftChild, TreeNode *rightChild)
34 {
35     parent->left = leftChild;
36     parent->right = rightChild;
37 }
38
39 /**
40  * @brief 前序遍历二叉树
41  * @param root 根节点
42  * @param result 存储遍历结果的向量
43  */
44 void preorderTraversal(TreeNode *root, std::vector<int> &result)
45 {
46     if (root)
47     {
48         result.push_back(root->data);
49         preorderTraversal(root->left, result);
50         preorderTraversal(root->right, result);
51     }
52 }
53
54 /**
55  * @brief 中序遍历二叉树
56  * @param root 根节点
57  * @param result 存储遍历结果的向量
58  */
59 void inorderTraversal(TreeNode *root, std::vector<int> &result)
60 {
61     if (root)
62     {
63         inorderTraversal(root->left, result);
64         result.push_back(root->data);
65         inorderTraversal(root->right, result);
66     }
67 }
68
69 /**
70  * @brief 后序遍历二叉树
71  * @param root 根节点
72  * @param result 存储遍历结果的向量
73  */
74 void postorderTraversal(TreeNode *root, std::vector<int> &result)
75 {
76     if (root)
77     {
78         postorderTraversal(root->left, result);
79         postorderTraversal(root->right, result);
80         result.push_back(root->data);
81     }
82 }
83
84 int main()
85 {
86     // 创建二叉树的节点数据及对应关系
87     TreeNode *root = createNode(89);
88     TreeNode *node2 = createNode(45);
89     TreeNode *node3 = createNode(97);
90     TreeNode *node4 = createNode(71);
91     TreeNode *node5 = createNode(68);
92     TreeNode *node6 = createNode(75);
93     TreeNode *node7 = createNode(47);
94     TreeNode *node8 = createNode(83);
95
96     createLink(root, node2, node3);
97     createLink(node2, node4, node5);
98     createLink(node3, nullptr, node6);
99     createLink(node5, node7, node8);
100
101     // 前序遍历
102     std::vector<int> preorderResult;
103     preorderTraversal(root, preorderResult);
104     std::cout << "前序遍历结果: ";
105     for (int num : preorderResult)
106     {
107         std::cout << num << " ";
108     }
109     std::cout << std::endl;
110
111
112     // 中序遍历
113     std::vector<int> inorderResult;
114     inorderTraversal(root, inorderResult);
115     std::cout << "中序遍历结果: ";
116     for (int num : inorderResult)
117     {
118         std::cout << num << " ";
119     }
120     std::cout << std::endl;
121
122
123     // 后序遍历
124     std::vector<int> postorderResult;
125     postorderTraversal(root, postorderResult);
126     std::cout << "后序遍历结果: ";
127     for (int num : postorderResult)
128     {
129         std::cout << num << " ";
130     }
131     std::cout << std::endl;
132
133     return 0;
134 }
135
136

```

