一、链表

结构体定义：

```
struct SLinkedNode
{
	int m_Val;
	SLinkedNode* m_pNext;

	SLinkedNode() : m_Val(0), m_pNext(nullptr) {}
};
```

1. 计算单链表长度

```
//************************************************************
//FUNCTION:: 计算链表长度
unsigned int getListLength(SLinkedNode* vHeadNode)
{
	unsigned int uiLength = 0;
	SLinkedNode* pCurrentNode = vHeadNode;
	while (pCurrentNode)
	{
		uiLength++;
		pCurrentNode = pCurrentNode->m_pNext;
	}

	return uiLength;
}
```

2.反转单链表

```
//************************************************************
//FUNCTION::反转单链表
SLinkedNode* reverseList(SLinkedNode* vHeadNode)
{
	if (!vHeadNode || !vHeadNode->m_pNext)
		return vHeadNode;

	SLinkedNode* pPrevNode = nullptr;
	SLinkedNode* pCurrNode = vHeadNode;
	SLinkedNode* pReverseNode = vHeadNode;
	while (pCurrNode)
	{
		SLinkedNode* pNextNode = pCurrNode->m_pNext;
		if (pNextNode == nullptr)
			pReverseNode = pCurrNode;

		pCurrNode->m_pNext = pPrevNode;
		pPrevNode = pCurrNode;
		pCurrNode = pNextNode;
	}

	return pReverseNode;
}
```

3. 单链表倒数第 K 个节点

```
//************************************************************
//FUNCTION:: 查找链表倒数第K个节点
SLinkedNode* findBackKthNode(SLinkedNode* vHeadNode, unsigned int vK)
{
	if (vHeadNode == nullptr || vK == 0)
		return nullptr;

	SLinkedNode* pBeforeNode = vHeadNode, *pBehindNode = vHeadNode;

	for (int i=0; i < vK - 1; i++)
	{
		if (pBeforeNode->m_pNext == nullptr)
		{
			return nullptr;
		}

		pBeforeNode = pBeforeNode->m_pNext;
	}

	while (pBeforeNode->m_pNext)
	{
		pBeforeNode = pBeforeNode->m_pNext;
		pBehindNode = pBehindNode->m_pNext;
	}

	return pBehindNode;
}
```

4. 查找链表中间节点

```
//************************************************************
//FUNCTION::查找链表中间节点
SLinkedNode* findMiddleNode(SLinkedNode* vHeadNode)
{
	if (vHeadNode == nullptr) return nullptr;

	SLinkedNode* pBeforeNode = vHeadNode, *pBehindNode = vHeadNode;

	while (pBeforeNode->m_pNext)
	{
		pBeforeNode = pBeforeNode->m_pNext;
		pBehindNode = pBehindNode->m_pNext;

		if (pBeforeNode->m_pNext)
		{
			pBeforeNode = pBeforeNode->m_pNext;
		}
	}

	return pBehindNode;
}
```

5. 从尾到头打印链表

```
//************************************************************
//FUNCTION::从尾到头打印链表_1
void reversePrintList_1(SLinkedNode* vHeadNode)
{
	std::stack<SLinkedNode*> ListStack;
	SLinkedNode* pCurrNode = vHeadNode;
	while (pCurrNode)
	{
		ListStack.push(pCurrNode);
		pCurrNode = pCurrNode->m_pNext;
	}

	while (!ListStack.empty())
	{
		SLinkedNode* pTopNode = ListStack.top();
		std::cout << pTopNode->m_Val << " ";
		ListStack.pop();
	}
	std::cout << std::endl;
}
//************************************************************
//FUNCTION::从尾到头打印链表_2
void reversePrintList_2(SLinkedNode* vHeadNode)
{
	if (vHeadNode == nullptr)
		return;

	reversePrintList_2(vHeadNode->m_pNext);

	std::cout << vHeadNode->m_Val << " ";
}
```

6. 合并两个有序链表

```
//*********************************************************
//FUNCTION::合并两个有序链表
SLinkedNode* mergeTwoSortedList(SLinkedNode* vHeadNode1, SLinkedNode*
vHeadNode2)
{
	if (!vHeadNode1)
		return vHeadNode2;
	else if (!vHeadNode2)
		return vHeadNode1;

	SLinkedNode* pMergeNode = nullptr;

	if (vHeadNode1->m_Val < vHeadNode2->m_Val)
	{
		pMergeNode = vHeadNode1;
		pMergeNode->m_pNext = mergeTwoSortedList(vHeadNode1->m_pNext,
vHeadNode2);
	}
	else
	{
		pMergeNode = vHeadNode2;
```

```
            pMergeNode->m_pNext = mergeTwoSortedList(vHeadNode1,
vHeadNode2->m_pNext);
        }

        return pMergeNode;
}
```

7. 如何判断单链表是否有环？

```
//************************************************************
//FUNCTION::判断单链表有环
bool hasCircle(SLinkedNode* vHeadNode)
{
        SLinkedNode* pFastNode = vHeadNode;
        SLinkedNode* pSlowNode = vHeadNode;

        while (pFastNode != nullptr && pFastNode->m_pNext != nullptr)
        {
                pFastNode = pFastNode->m_pNext->m_pNext;
                pSlowNode = pSlowNode->m_pNext;

                if (pFastNode == pSlowNode)
                        return true;
        }

        return false;
}
```

8.判断两个链表是否相交

```
//************************************************************
//FUNCTION::判断两个单链表是否相交
bool isIntersect(SLinkedNode* vHeadNode1, SLinkedNode* vHeadNode2)
{
        if (vHeadNode1 == nullptr || vHeadNode2 == nullptr)
                return false;

        SLinkedNode* pTailNode1 = vHeadNode1;
        while (pTailNode1->m_pNext)
        {
                pTailNode1 = pTailNode1->m_pNext;
        }

        SLinkedNode* pTailNode2 = vHeadNode2;
        while (pTailNode2->m_pNext)
        {
                pTailNode2 = pTailNode2->m_pNext;
        }

        return pTailNode1 == pTailNode2;
}
```

9. 求两个链表相交的第一个节点

```
/************************************************************
//FUNCTION::两个链表相交的第一个相同节点
SLinkedNode* getFirstCommonNode(SLinkedNode* vHeadNode1, SLinkedNode* vHeadNode2)
{
	if (vHeadNode1 == nullptr || vHeadNode2 == nullptr)
		return nullptr;
	unsigned int uiLength1 = 0;
	SLinkedNode* pTailNode1 = vHeadNode1;
	while (pTailNode1->m_pNext)
	{
		pTailNode1 = pTailNode1->m_pNext;
		uiLength1++;
	}

	unsigned int uiLength2 = 0;
	SLinkedNode* pTailNode2 = vHeadNode2;
	while (pTailNode2->m_pNext)
	{
		pTailNode2 = pTailNode2->m_pNext;
		uiLength2++;
	}

	if (pTailNode1 != pTailNode2)
		return nullptr;

	if (uiLength1 < uiLength2)
	{
		int uiOffset = uiLength2 - uiLength1;
		while (uiOffset)
		{
			vHeadNode2 = vHeadNode2->m_pNext;
			uiOffset--;
		}
	}
	else
	{
		int uiOffset = uiLength1 - uiLength2;
		while (uiOffset)
		{
			vHeadNode1 = vHeadNode1->m_pNext;
			uiOffset--;
		}
	}

	while (vHeadNode1 != vHeadNode2)
	{
		vHeadNode1 = vHeadNode1->m_pNext;
		vHeadNode2 = vHeadNode2->m_pNext;
	}

	return vHeadNode1;
}
```

10. 求单链表环中第一个入口节点

```
//************************************************************
//FUNCTION::单链表存在环，计算进入环的第一个交点
SLinkedNode* getFirstNodeInCircle(SLinkedNode* vHeadNode)
{
	if (vHeadNode == nullptr || vHeadNode->m_pNext == nullptr)
		return nullptr;

	SLinkedNode* pFastNode = vHeadNode;
	SLinkedNode* pSlowNode = vHeadNode;
	while (pFastNode != nullptr && pFastNode->m_pNext != nullptr)
	{
		pFastNode = pFastNode->m_pNext->m_pNext;
		pSlowNode = pSlowNode->m_pNext;

		if (pFastNode == pSlowNode)
			break;
	}

	if (pFastNode == nullptr || pFastNode->m_pNext == nullptr)
		return nullptr;

	//环中此节点作为假设的尾节点，变成两个单链表相交问题
	SLinkedNode* pAssumedTail = pSlowNode;
	SLinkedNode* pHead1 = vHeadNode;
	SLinkedNode* pHead2 = pAssumedTail->m_pNext;

	//SLinkedNode* pFirstCommonNode = getFirstCommonNode(pHead1, pHead2);
	unsigned int uiLength1 = 1;
	SLinkedNode* pTailNode1 = pHead1;
	while (pTailNode1 != pAssumedTail)
	{
		pTailNode1 = pTailNode1->m_pNext;
		uiLength1++;
	}

	unsigned int uiLength2 = 1;
	SLinkedNode* pTailNode2 = pHead2;
	while (pTailNode2 != pAssumedTail)
	{
		pTailNode2 = pTailNode2->m_pNext;
		uiLength2++;
	}

	pTailNode1 = pHead1;
	pTailNode2 = pHead2;
	if (uiLength1 < uiLength2)
	{
		int uiOffset = uiLength2 - uiLength1;
		while (uiOffset)
		{
			pTailNode2 = pTailNode2->m_pNext;
			uiOffset--;
		}
	}
	else
	{
```

```
		int uiOffset = uiLength1 - uiLength2;
		while (uiOffset)
		{
			pTailNode1 = pTailNode1->m_pNext;
			uiOffset--;
		}
	}

	while (pTailNode1 != pTailNode2)
	{
		pTailNode1 = pTailNode1->m_pNext;
		pTailNode2 = pTailNode2->m_pNext;
	}

	return pTailNode1;
}
```

11. O(1)时间复杂度删除指定链表节点

```
//****************************************************
//FUNCTION::O(1)的时间复杂度删除指定链表节点
void deleteListNode(SLinkedNode* vHeadNode, SLinkedNode* vToDeletedNode)
{
	if (vToDeletedNode == nullptr) return;

	if (vToDeletedNode->m_pNext)
	{
		//复制下一个节点数据到当前节点，然后删除下一个节点
		vToDeletedNode->m_Val = vToDeletedNode->m_pNext->m_Val;
		SLinkedNode* pTempNode = vToDeletedNode->m_pNext;
		vToDeletedNode->m_pNext = vToDeletedNode->m_pNext->m_pNext;
		delete pTempNode;
		pTempNode = nullptr;
	}
	else //删除节点是最后一个节点
	{
		if (vHeadNode == vToDeletedNode)
		{
			delete vToDeletedNode;
			vToDeletedNode = nullptr;
		}
		else
		{
			SLinkedNode* pCurrNode = vHeadNode;
			while (pCurrNode->m_pNext != vToDeletedNode)
			{
				pCurrNode = pCurrNode->m_pNext;
			}

			pCurrNode->m_pNext = nullptr;
			delete vToDeletedNode;
			vToDeletedNode = nullptr;
		}
	}
}
```