数据结构作业

1、d1—>d2—>d3—>d4

2、B

3、A

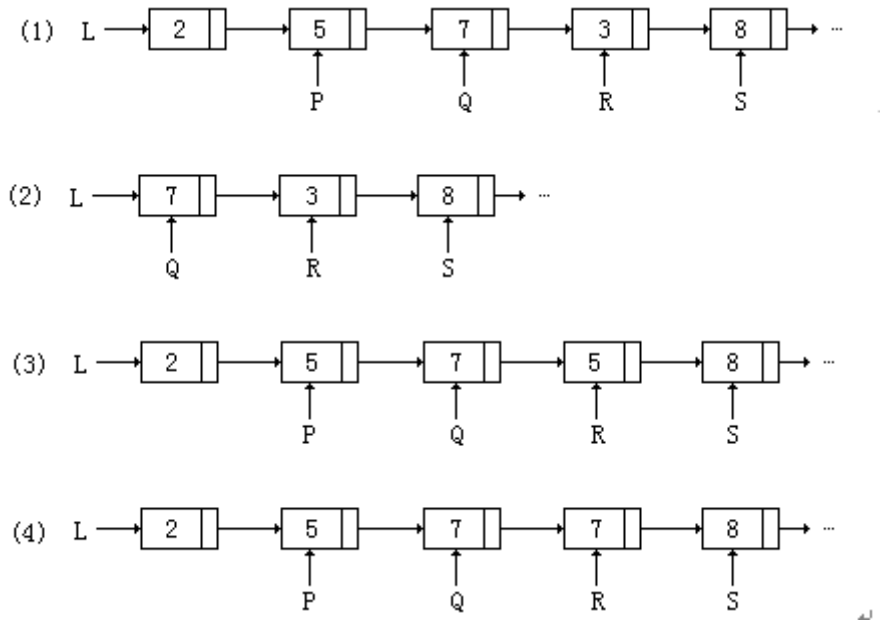
4、（1）一半　表长和插入（或删除）位置

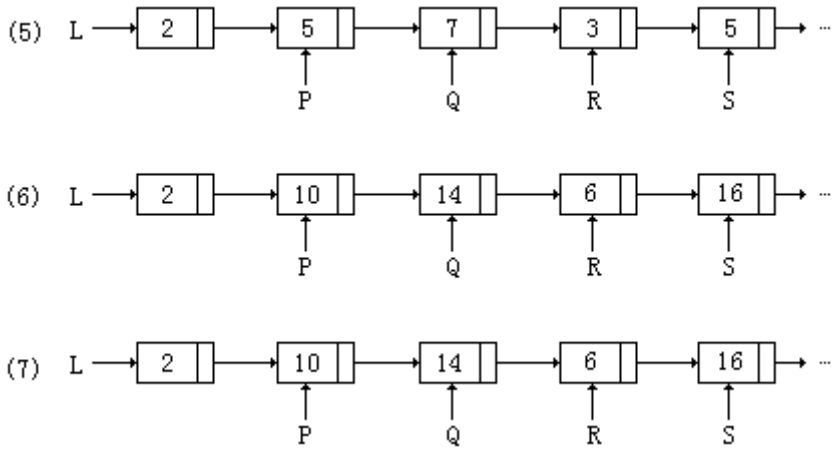
（2）一定　不一定

（3）其前驱结点的 next 域

（4）方便操作

5、





6、(1)　(11)　(3)　(14)

(2)　(10)　(12)　(8)　(3)　(14)

(3)　(10)　(12)　(7)　(3)　(14)

(4)　(12)　(11)　(3)　(14)

(5)　(9)　(11)　(3)　(14)

7、

```cpp
#include <iostream>

// 定义链表节点
struct ListNode {
    int data;
    ListNode* next;
    ListNode(int val) : data(val), next(nullptr) {}
};

// 就地逆置单链表的函数
void reverseLinkedList(ListNode*& head) {
    ListNode* prev = nullptr;
    ListNode* current = head;
    ListNode* next = nullptr;

    while (current != nullptr) {
        next = current->next;  // 保存下一个节点的指针

        current->next = prev;  // 将当前节点的 next 指针指向前一个节点

```

```
        prev = current;         // 更新 prev 指针为当前节点
        current = next;         // 更新 current 指针为下一个节点
    }

    head = prev;  // 更新链表头指针
}

// 打印链表的函数
void printLinkedList(ListNode* head) {
    while (head != nullptr) {
        std::cout << head->data << " ";
        head = head->next;
    }
    std::cout << std::endl;
}

// 主函数
int main() {
    // 创建一个示例链表: 1 -> 2 -> 3 -> 4 -> 5
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
    head->next->next = new ListNode(3);
    head->next->next->next = new ListNode(4);
    head->next->next->next->next = new ListNode(5);

    std::cout << "Original Linked List: ";
    printLinkedList(head);

    // 调用就地逆置函数
    reverseLinkedList(head);

    std::cout << "Reversed Linked List: ";
    printLinkedList(head);

    // 释放链表内存
    while (head != nullptr) {
        ListNode* temp = head;
        head = head->next;
        delete temp;
    }

    return 0;
}
```

8、有序：O(n)

无序：O(n)

9、不一定正确。访问线性表的第 i 个元素的时间复杂度与线性表的实现方式有关。在顺序存储结构中，可以通过数组下标直接访问第 i 个元素，时间复杂度为 O(1)；而在链式存储结构中，需要从头结点开始逐个遍历到第 i 个元素，时间复杂度为 O(i)。因此，这个说法在链式存储结构下是正确的，但在顺序存储结构下不成立。

10、在循环单链表表示的链队列中，可以不设置队头指针。在链队列中，队头指针用于指向队列的首元素，而队尾指针用于指向队列的尾元素。在循环单链表中，队头和队尾的位置是相邻的，可以通过队尾指针的 next 指针找到队头。因此，如果不设置队头指针，可以通过队尾指针的 next 指针访问队头元素。

11、线性表顺序存储和链式存储的特色比较：

顺序存储：

特色：使用数组顺序存储元素，支持随机访问。

优点：访问速度快，对 CPU 缓存友好。

缺点：插入和删除元素可能涉及元素的移动，导致操作耗时。

插入操作时间复杂度：O(n)（平均情况下，需要移动 n/2 个

元素)。

链式存储:

特色:使用指针将元素链接在一起,支持动态内存分配。

优点:插入和删除元素方便,不需要移动其他元素。

缺点:访问速度相对较慢,由于指针引用可能导致缓存不友好。

插入操作时间复杂度:O(1)(在已知插入位置的情况下)。

12、

```cpp
#include <iostream>

// 定义链表结点
struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};

// 删除小于 a 的元素
void deleteLessThanA(Node*& head, int a) {
    // 添加头结点简化删除操作
    Node* dummy = new Node(0);
    dummy->next = head;

    Node* current = dummy;

    while (current->next != nullptr) {
        if (current->next->data < a) {
            Node* temp = current->next;
            current->next = current->next->next;
            delete temp;
        } else {
            current = current->next;
        }
    }

    // 更新原始头结点
```

```
    head = dummy->next;
    delete dummy;
}

// 打印链表
void printList(Node* head) {
    while (head != nullptr) {
        std::cout << head->data << " ";
        head = head->next;
    }
    std::cout << std::endl;
}

int main() {
    // 示例使用
    Node* head = new Node(3);
    head->next = new Node(5);
    head->next->next = new Node(2);
    head->next->next->next = new Node(8);

    std::cout << "Original List: ";
    printList(head);

    int a = 5;
    deleteLessThanA(head, a);

    std::cout << "List after deleting elements less than " << a << ":
    ";
printList(head);

    return 0;
}
```

13、

（1）123 231 321 213 132

（2）不能得到 435612 的出站序列。因为 4356 出站说明 12 已经在栈中，1 不可能先于 2 出栈。可以得到 135426 的出站序列，其相应操作为：SXSSXSSXXXSX。

14、删除栈中特定元素

15、

```
void test(int &sum) {
    int x;
    sum = 0;

    do {
        cin >> x;
        if (x != 0) {
            sum += x;
        }
    } while (x != 0);

    cout << sum;
}
```

16、rhcae

17、将队列 Q 中的元素逐个出队，并压入栈 S，然后将栈 S 中的元素逐个出栈并入队列 Q。

18、栈顶　队尾和队头

19、不正确。栈和队列都是线性结构，只是操作的限制不同。

20、

```
/*判空*/
```

```
int SeQueue_Empty(SeQueue *Q)
{
    return Q->rear==Q->front;
}

/*判满*/
int SeQueue_Full(SeQueue *Q)
{
    return (Q->rear+1)%MAXSIZE==Q->front;
}
```

21、

```
StrLength(s): 14
StrLength(t): 4
SubString(s, 8, 7): STUDENT
SubString(t, 2, 1): O
Index(s, 'A'): 2
Index(s, t): 18446744073709551615
Replace(s, 'STUDENT', q): I AM A WORKER
Concat(SubString(s, 6, 2), Concat(t, SubString(s, 7, 8))): A GOOD WORKER
```

22、d(4k,l)=l/(4k)

23、

```
#include <iostream>
#include <cstring>
using namespace std;

// 定长顺序存储表示串
struct String {
    char data[100];  // 假设最大长度为100
    int length;
};

void DeleteAllSubStrings(String& S, const String& T) {
    int i = 0;
    while (i <= S.length - T.length) {
        bool match = true;
        for (int j = 0; j < T.length; ++j) {
```

```
            if (S.data[i + j] != T.data[j]) {
                match = false;
                break;
            }
        }
        if (match) {
            // 删除匹配的子串
            memmove(S.data + i, S.data + i + T.length, S.length - i - T.length + 1);
            S.length -= T.length;
        } else {
            ++i;
        }
    }
}

int main() {
    String S = { "abcbcdabcbcda", 13 };
    String T = { "abc", 3 };

    DeleteAllSubStrings(S, T);

    cout << "Result: " << S.data << endl;

    return 0;
}
```

24、a)　6*8*6=288 字节

b)　1000+（1*8+4）*6=1072

c)　1000+（2*6+3）*6=1090

25、d)　100

100+（1*3*5*8+1*5*8+1*8+1）*4=676

100+（3*3*5*8+1*5*8+2*8+5）*4=1784

100+（8*3*5*8+2*5*8+4*8+7）*4=4416

e) $a_{0,0,0,0}$ $a_{1,0,0,0}$ $a_{2,0,0,0}$ $a_{3,0,0,0}$

26、f) $k=2i+j-3$

g) $i = [(k+1)/3] + 1$

$j = k - 2i + 3$

27、$k=i+j-i\%2-1$

28、

```
1.  spmatrix * Add(spmatrix *A, spmatrix *B)
2.  {
3.      void Out(spmatrix *S);
4.      spmatrix *C;
5.      int i = 0;
6.      int j = 0;
7.      int t = 0;
8.      C = (spmatrix *)malloc(sizeof(spmatrix));
9.      if (!C)
10.     {
11.         printf("分配空间失败!\n");
12.         exit(-1);
13.     }
14.     C->m = A->m;
15.     C->n = A->n;
16.
17.     if (A->m != B->m || A->n != B->n)
18.     {
19.         printf("矩阵形状不一致!无法进行加法运算!\n");
20.     }
21.
22.     while (i < A->t && j < B->t)
23.     {
24.         if (A->data[i].i > B->data[j].i)
25.         {
26.             C->data[t].i = B->data[j].i;
```

```
            C->data[t].j = B->data[j].j;
            C->data[t].val = B->data[j].val;
            j++;
            t++;
        }
        else if (A->data[i].i < B->data[j].i)
        {
            C->data[t].i = A->data[i].i;
            C->data[t].j = A->data[i].j;
            C->data[t].val = A->data[i].val;
            i++;
            t++;
        }
        else
        {
            if (A->data[i].j < B->data[j].j)
            {
                C->data[t].i = A->data[i].i;
                C->data[t].j = A->data[i].j;
                C->data[t].val = A->data[i].val;
                i++;
                t++;
            }
            else if (A->data[i].j > B->data[j].j)
            {
                C->data[t].i = A->data[i].i;
                C->data[t].j = A->data[i].j;
                C->data[t].val = A->data[i].val;
                i++;
                t++;
            }
            else
            {
                C->data[t].i = A->data[i].i;
                C->data[t].j = A->data[i].j;
                C->data[t].val = A->data[i].val + B->data[j].val;
                j++;
                i++;
                t++;
            }
        }
    }

    while (i < A->t)
```

```
    {
        C->data[t].i = A->data[i].i;
        C->data[t].j = A->data[i].j;
        C->data[t].val = A->data[i].val;
        i++;
        t++;
    }
    while (j < B->t)
    {
        C->data[t].i = B->data[j].i;
        C->data[t].j = B->data[j].j;
        C->data[t].val = B->data[j].val;
        j++;
        t++;
    }

    C->t = t;
    return C;

}
```

29、

```
Status GetElem(T2SMatrix M, int i, int j, ElemType &e)
/* 求二元组矩阵的元素 A[i][j]的值 e  */
{
    int cur,next;
    cur = M.cpot[i];//当前一行的起始位置
    next = M.cpot[i + 1];//下一行的起始位置
    e = 0;//需要注意，稀疏矩阵中的 e 不是非零元素就是零元素
    if(i <= 0 || j <= 0 || i > M.mu || j > M.nu)
        return ERROR;
    for( ;cur < next; ++cur){
        if(M.data[cur].j == j){//匹配成功
            e = M.data[cur].e;//返回非零元素 e
            return OK;
        }
    }
    return OK; //匹配失败，返回零元素 e
}
```

这种存储结构的优点是可以随机存取稀疏矩阵任意一行的非零元，而三元组顺序表只能按行进行顺序存储。