数据结构作业

1、d1—>d2—>d3—>d4

2、B

3、A

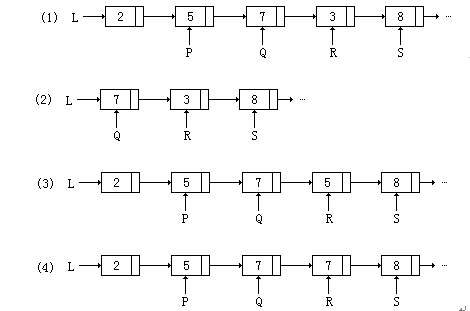
4、（1）一半 表长和插入（或删除）位置

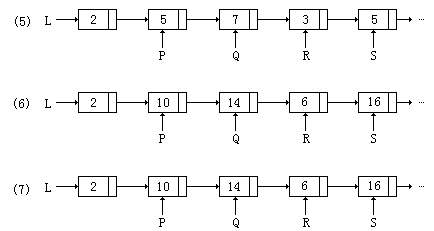
（2）一定 不一定

（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、

1. #include <iostream>
3. // 定义链表节点
4. **struct** ListNode {
5. **int** data;
6. ListNode\* next;
7. ListNode(**int** val) : data(val), next(nullptr) {}
8. };
10. // 就地逆置单链表的函数
11. **void** reverseLinkedList(ListNode\*& head) {
12. ListNode\* prev = nullptr;
13. ListNode\* current = head;
14. ListNode\* next = nullptr;
16. **while** (current != nullptr) {
17. next = current->next;  // 保存下一个节点的指针
19. current->next = prev;  // 将当前节点的next指针指向前一个节点
21. prev = current;        // 更新prev指针为当前节点
22. current = next;        // 更新current指针为下一个节点
23. }
25. head = prev;  // 更新链表头指针
26. }
28. // 打印链表的函数
29. **void** printLinkedList(ListNode\* head) {
30. **while** (head != nullptr) {
31. std::cout << head->data << " ";
32. head = head->next;
33. }
34. std::cout << std::endl;
35. }
37. // 主函数
38. **int** main() {
39. // 创建一个示例链表: 1 -> 2 -> 3 -> 4 -> 5
40. ListNode\* head = **new** ListNode(1);
41. head->next = **new** ListNode(2);
42. head->next->next = **new** ListNode(3);
43. head->next->next->next = **new** ListNode(4);
44. head->next->next->next->next = **new** ListNode(5);
46. std::cout << "Original Linked List: ";
47. printLinkedList(head);
49. // 调用就地逆置函数
50. reverseLinkedList(head);
52. std::cout << "Reversed Linked List: ";
53. printLinkedList(head);
55. // 释放链表内存
56. **while** (head != nullptr) {
57. ListNode\* temp = head;
58. head = head->next;
59. **delete** temp;
60. }
62. **return** 0;
63. }

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、

1. #include <iostream>
3. // 定义链表结点
4. **struct** Node {
5. **int** data;
6. Node\* next;
7. Node(**int** val) : data(val), next(nullptr) {}
8. };
10. // 删除小于 a 的元素
11. **void** deleteLessThanA(Node\*& head, **int** a) {
12. // 添加头结点简化删除操作
13. Node\* dummy = **new** Node(0);
14. dummy->next = head;
16. Node\* current = dummy;
18. **while** (current->next != nullptr) {
19. **if** (current->next->data < a) {
20. Node\* temp = current->next;
21. current->next = current->next->next;
22. **delete** temp;
23. } **else** {
24. current = current->next;
25. }
26. }
28. // 更新原始头结点
29. head = dummy->next;
30. **delete** dummy;
31. }
33. // 打印链表
34. **void** printList(Node\* head) {
35. **while** (head != nullptr) {
36. std::cout << head->data << " ";
37. head = head->next;
38. }
39. std::cout << std::endl;
40. }
42. **int** main() {
43. // 示例使用
44. Node\* head = **new** Node(3);
45. head->next = **new** Node(5);
46. head->next->next = **new** Node(2);
47. head->next->next->next = **new** Node(8);
49. std::cout << "Original List: ";
50. printList(head);
52. **int** a = 5;
53. deleteLessThanA(head, a);
55. std::cout << "List after deleting elements less than " << a << ": ";
56. printList(head);
58. **return** 0;
59. }

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、

1. **void** test(**int** &sum) {
2. **int** x;
3. sum = 0;
5. **do** {
6. cin >> x;
7. **if** (x != 0) {
8. sum += x;
9. }
10. } **while** (x != 0);
12. cout << sum;
13. }

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。

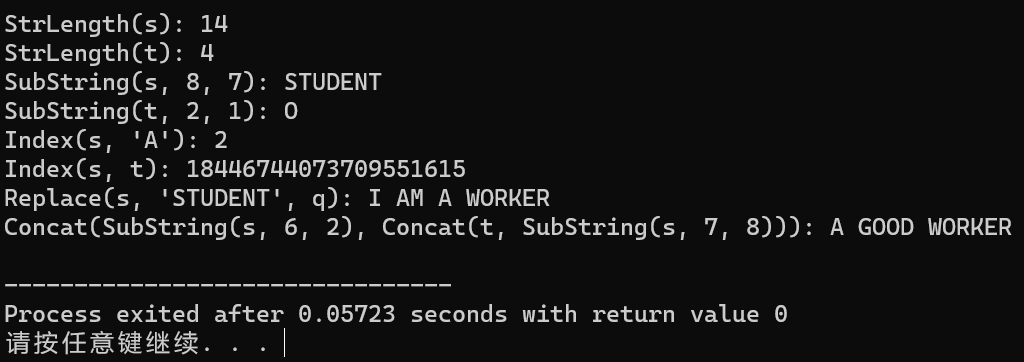
18、栈顶 队尾和队头

19、不正确。栈和队列都是线性结构，只是操作的限制不同。

20、

1. /\*判空\*/
2. **int** SeQueue\_Empty(SeQueue \*Q)
3. {
4. **return** Q->rear==Q->front;
5. }
7. /\*判满\*/
8. **int** SeQueue\_Full(SeQueue \*Q)
9. {
10. **return** (Q->rear+1)%MAXSIZE==Q->front;
11. }

21、



22、d(4k,l)=l/(4k)

23、

1. #include <iostream>
2. #include <cstring>
3. **using** **namespace** std;
5. // 定长顺序存储表示串
6. **struct** String {
7. **char** data[100];  // 假设最大长度为100
8. **int** length;
9. };
11. **void** DeleteAllSubStrings(String& S, **const** String& T) {
12. **int** i = 0;
13. **while** (i <= S.length - T.length) {
14. **bool** match = **true**;
15. **for** (**int** j = 0; j < T.length; ++j) {
16. **if** (S.data[i + j] != T.data[j]) {
17. match = **false**;
18. **break**;
19. }
20. }
21. **if** (match) {
22. // 删除匹配的子串
23. memmove(S.data + i, S.data + i + T.length, S.length - i - T.length + 1);
24. S.length -= T.length;
25. } **else** {
26. ++i;
27. }
28. }
29. }
31. **int** main() {
32. String S = { "abcbcdabcbcda", 13 };
33. String T = { "abc", 3 };
35. DeleteAllSubStrings(S, T);
37. cout << "Result: " << S.data << endl;
39. **return** 0;
40. }

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) a0,0,0,0 a1,0,0,0 a2,0,0,0 a3,0,0,0

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;
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. }
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;
27. C->data[t].j = B->data[j].j;
28. C->data[t].val = B->data[j].val;
29. j++;
30. t++;
31. }
32. **else** **if** (A->data[i].i < B->data[j].i)
33. {
34. C->data[t].i = A->data[i].i;
35. C->data[t].j = A->data[i].j;
36. C->data[t].val = A->data[i].val;
37. i++;
38. t++;
39. }
40. **else**
41. {
42. **if** (A->data[i].j < B->data[j].j)
43. {
44. C->data[t].i = A->data[i].i;
45. C->data[t].j = A->data[i].j;
46. C->data[t].val = A->data[i].val;
47. i++;
48. t++;
49. }
50. **else** **if** (A->data[i].j > B->data[j].j)
51. {
52. C->data[t].i = A->data[i].i;
53. C->data[t].j = A->data[i].j;
54. C->data[t].val = A->data[i].val;
55. i++;
56. t++;
57. }
58. **else**
59. {
60. C->data[t].i = A->data[i].i;
61. C->data[t].j = A->data[i].j;
62. C->data[t].val = A->data[i].val + B->data[j].val;
63. j++;
64. i++;
65. t++;
66. }
67. }
68. }
70. **while** (i < A->t)
71. {
72. C->data[t].i = A->data[i].i;
73. C->data[t].j = A->data[i].j;
74. C->data[t].val = A->data[i].val;
75. i++;
76. t++;
77. }
78. **while** (j < B->t)
79. {
80. C->data[t].i = B->data[j].i;
81. C->data[t].j = B->data[j].j;
82. C->data[t].val = B->data[j].val;
83. j++;
84. t++;
85. }
87. C->t = t;
88. **return** C;
90. }

29、

1. Status GetElem(T2SMatrix M, **int** i, **int** j, ElemType &e)
2. /\* 求二元组矩阵的元素A[i][j]的值e  \*/
3. {
4. **int** cur,next;
5. cur = M.cpot[i];//当前一行的起始位置
6. next = M.cpot[i + 1];//下一行的起始位置
7. e = 0;//需要注意，稀疏矩阵中的e不是非零元素就是零元素
8. **if**(i <= 0 || j <= 0 || i > M.mu || j > M.nu)
9. **return** ERROR;
10. **for**( ;cur < next; ++cur){
11. **if**(M.data[cur].j == j){//匹配成功
12. e = M.data[cur].e;//返回非零元素e
13. **return** OK;
14. }
15. }
16. **return** OK; //匹配失败，返回零元素e
17. }

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。