知识点总结

· 在C语言中，单链表是一种常见的数据结构，它由一系列结点（又称节点）组成，每个结点包含两部分：数据域和指针域。

· 数据域：储存结点的实际数据，类型可以是任意数据类型（如整型、字符型等）。

· 指针域：储存下一个结点的地址。

· 单链表的各个结点在内存中并非连续存放，通过指针连接形成有序的链式结构。

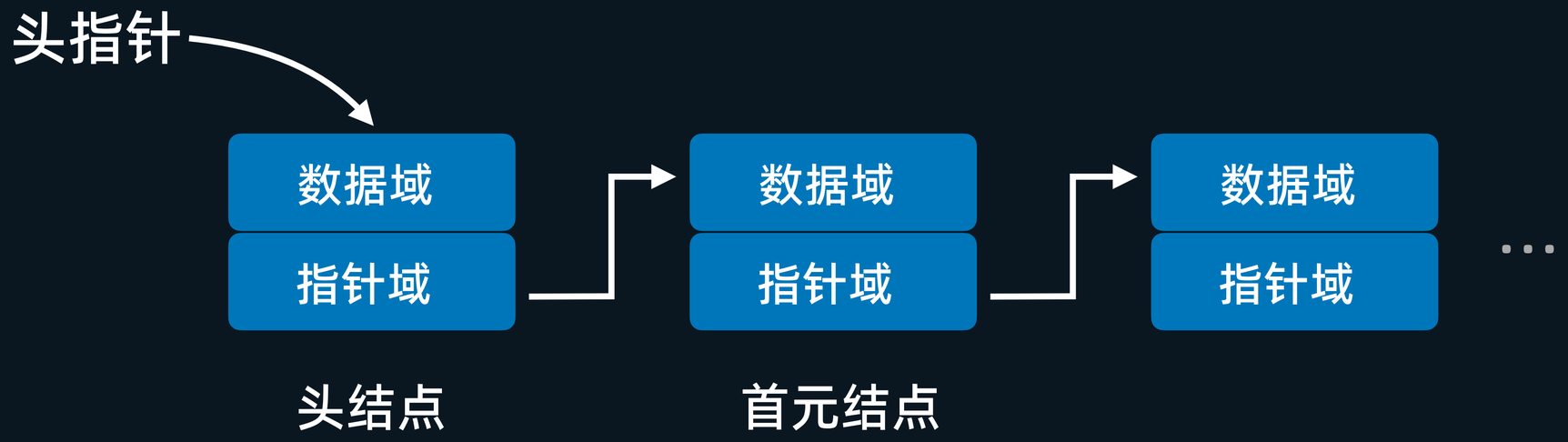
· 单链表一般可以分为带头结点和不带头结点两种。

· 在C语言中，头结点是单链表中的一个特殊结点，位于单链表的第一个实际数据结点（首元结点）之前。头结点通常不储存有效数据，主要作为辅助结点用于简化单链表操作。

- 带头结点的单链表：

头指针指向头结点，头结点的next指针指向首元结点。即使链表为空，头结点仍然存在，其next指针为NULL。

图示如下：

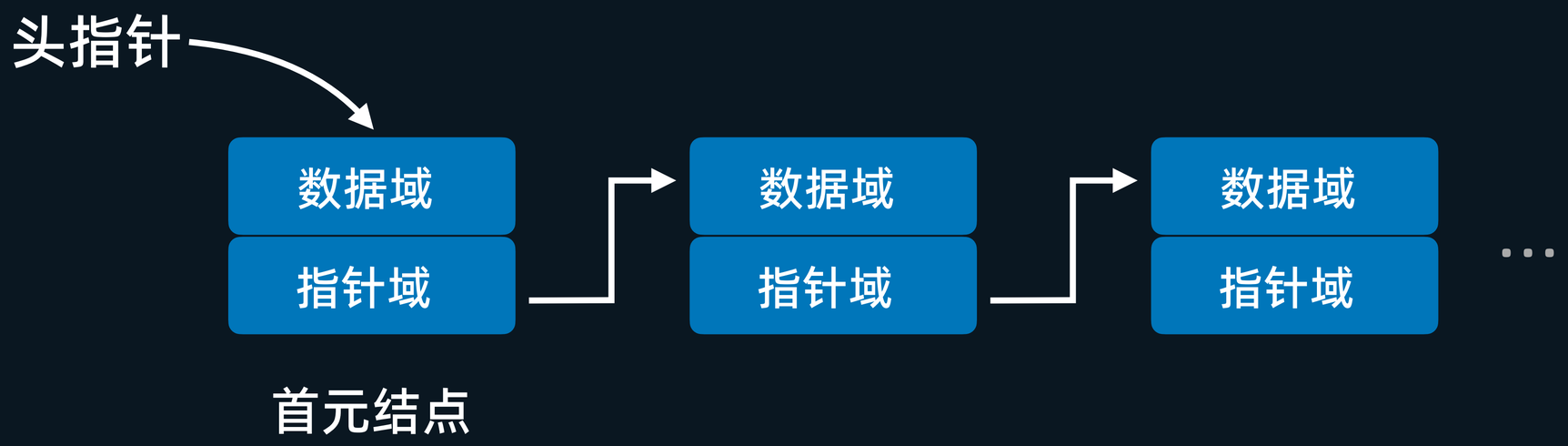


头指针 -> 头结点 -> 首元结点 -> 结点2 -> ... -> NULL

· 不带头结点的单链表：

头指针直接指向首元结点。若链表为空，头指针为NULL。

图示如下：



头指针 -> 首元结点 -> 结点2 -> ... -> NULL

· 单链表的创建与应用：

一、带头结点的单链表

1、**定义结点结构体**：通常使用结构体来定义单链表的结点。

例：
```
// 定义单链表结点
typedef struct Node {
    int data; // 数据域
    struct Node* next; // 指针域，指向下一个结点
} Node;
```

其中，成员data用于存储数据，这里假设为int类型，实际应用中可以是任意数据类型。

成员next是一个指针，它指向同类型的下一个结点。

2、创建并初始化空链表

```
// 必须动态分配内存创建头结点
// head是指向头结点的头指针
Node* head = (Node*)malloc(sizeof(Node));
// 将头结点的data成员初始化为0
head->data = 0;
// 将头结点的next指针初始化为NULL，表明链表为空
head->next = NULL;
```

3、插入结点：

1）头插法：在链表头部插入新结点，链表的结点顺序与输入顺序相反。

例：
```
void insert_at_head(Node* head, int data) {
    Node* new_node  = (Node*)malloc(sizeof(Node));
    new_node->data = data;
    // 以下两步指针操作的顺序不能反！
    // 新结点指向原第一个结点
    new_node->next = head->next;
    // 头结点指向新结点
    head->next = new_node;
}
```

2）**尾插法**：在链表尾部插入新结点，链表的结点顺序与输入顺序一致。

例：

```
void insert_at_tail(Node* head, int data) {
    Node* new_node  = (Node*)malloc(sizeof(Node));
    new_node->data = data;
    new_node->next = NULL;
    // 找到最后一个结点
    Node* current = head;
    // 注意判断的是current->next
    while (current->next != NULL) {
        current = current->next;
    }
    // 尾结点连接新结点
    current->next = new_node;
}
```

3）指定位置插入：在单链表的指定位置pos插入新结点。

```
void insert_at_position(Node* head, int data, unsigned int pos) {
    // 创建当前指针current并使其指向头结点，用于遍历单链表，寻找插入位置的上一个结点
    Node* current = head;
    // 用于记录当前遍历到的位置
    int count = 0;
    // 寻找插入位置的前驱结点（即第pos-1个结点）
    while(current != NULL && count < pos) {
        current = current->next;
        count++;
    }
    // 如果当前结点为空，说明要插入的位置超出了链表的长度范围
    if (current == NULL) {
        printf("插入位置无效！");
        // 结束函数，不再执行后续插入操作
        return;
    }

    Node* new_node  = (Node*)malloc(sizeof(Node));
    new_node->data = data;
    // 让新结点的next指针指向当前结点（前驱结点）的下一个结点，即原后继结点
    new_node->next = current->next;
    // 让当前结点（前驱结点）的next指针指向新结点，完成插入操作
    current->next = new_node ;
}
```

4、**遍历单链表**：依次访问链表中的每个结点。

例：

```
void traverse_list(Node* head) {
    // 从第一个实际数据结点开始遍历，即从头结点的下一个结点开始遍历
    Node* current = head->next;
    // 只要当前结点不为空，就继续循环
    while (current!= NULL) {
        printf("%d -> ", current->data);
        // 将指针current移动到下一个结点
        current = current->next;
    }
    printf("\n");
}
```

5、删除结点：

1）**删除首元结点**：删除单链表的第一个实际数据结点。

例：

```
void delete_first(Node* head) {
    // 如果是空链表，直接返回，不执行后续的删除操作
    if (head->next == NULL) {
        return;
    }
    // 创建一个临时指针temp，使其指向要删除的首元结点
    Node* temp = head->next;
    // 将头结点的next指针绕过首元结点，直接指向首元结点的下一个结点
    head->next = temp->next;
    // 释放temp指针所指结点（即首元结点）占用的内存空间
    free(temp);
}
```

2）**删除尾结点**：删除单链表的最后一个实际数据结点。

例：

```
void delete_last(Node* head) {
    // 如果是空链表，直接返回，不执行后续的删除操作
    if (head->next == NULL)
        return;
    // 创建当前指针current并使其指向头结点，用于遍历单链表，找到倒数第二个结点
    Node* current = head;
    // 通过while循环找到单链表中的倒数第二个结点
    // 注意循环条件是当前结点的下一个结点的下一个结点不为空
    while(current->next->next != NULL) {
        current = current->next;
    }
    // 创建一个临时指针temp，使其指向要删除的尾结点（即 current 指针所指结点的下一个结点）
    Node* temp = current->next;
    // 将倒数第二个结点的next指针设为NULL，变成新的尾结点
    current->next = NULL;
    // 释放temp指针所指结点（即尾结点）占用的内存空间
    free(temp);
}
```

3）删除指定结点：删除单链表中值为target的结点。

例：

```
void delete_node(Node* head, int target) {
    // 创建当前指针current并使其指向头结点，用于遍历单链表，寻找目标结点的前驱结点
    Node* current = head;
    // 通过while循环寻找目标结点的前驱结点
    // 只要当前结点的下一个结点存在（即还未到达单链表末尾），并且下一个结点的数据值不等于目标值target，就继续循环
    while (current->next != NULL && current->next->data != target) {
        // 将指针current移动到下一个结点
        current = current->next;
    }

    if (current->next != NULL) {
        // 创建临时指针temp，使其指向目标结点（即 current指针所指结点的下一个结点）
        Node* temp = current->next;
        // 将当前结点（即目标结点的前驱结点）的next指针绕过目标结点，直接指向目标结点的下一个结点
        current->next = temp->next;
        // 释放temp指针所指结点占用的内存空间
        free(temp);
    }
}
```

二、不带头结点的单链表

1、定义结点结构体。

例：

```
// 定义链表结点结构体
typedef struct Node {
    int data;               // 数据域
    struct Node* next;      // 指针域
} Node;
```

2、创建并初始化空链表。

```
// 通过将头指针head赋值为NULL，表示链表当前为空，没有任何结点
Node* head = NULL;
```

3、插入结点。

1）头插法。

```
// 注意参数head是一个二级指针，用来传递和修改链表的头指针
void insert_at_head(Node** head, int data) {
    // 创建新结点
    Node* new_node = (Node*)malloc(sizeof(Node));
    new_node->data = data;
    // 新结点的next指针指向原来的第一个实际数据结点
    new_node->next = *head;
    // 更新头指针，使其指向新结点
    *head = new_node;
}
```

2）尾插法。

例：

```
void insert_at_tail(Node** head, int data) {
    // 创建新结点
    Node* new_node = (Node*)malloc(sizeof(Node));
    new_node->data = data;
    new_node->next = NULL;
    // 处理空链表特殊情况
    if (*head == NULL) {
        *head = new_node;
        return;
    }
    // 找到最后一个结点
    Node* current = *head;
    while (current->next != NULL) {
        current = current->next;
    }
    // 连接新结点
    current->next = new_node;
}
```

3）指定位置插入：在单链表的指定位置pos插入新结点。

```c
void insert_at_position(Node** head, int data, unsigned int pos) {
    // 处理pos=0的情况（头插法）
    if (pos == 0) {
        insert_at_head(head, data);
        return;
    }
    // 寻找前驱结点
    Node* current = *head;
    int count = 0;
    while(current != NULL && count < pos - 1) {
        current = current->next;
        count++;
    }
    // 检查位置有效性
    if (current == NULL) {
        printf("插入位置无效！\n");
        // 结束函数，不再执行后续插入操作
        return;
    }
    // 创建新结点
    Node* new_node = (Node*)malloc(sizeof(Node));
    new_node->data = data;
    // 插入新结点
    new_node->next = current->next;
    current->next = new_node;
}
```

4、遍历单链表：依次访问链表中的每个结点。

例：

```
void traverse_list(Node* head) {
    // 从第一个实际数据结点开始遍历，即从头结点的下一个结点开始遍历
    Node* current = head->next;
    // 只要当前结点不为空，就继续循环
    while (current!= NULL) {
        printf("%d -> ", current->data);
        // 将指针current移动到下一个结点
        current = current->next;
    }
    printf("\n");
}
```

5、删除结点:

1）删除首元结点。

例:

```
void delete_first(Node** head) {
     if (*head == NULL)
          return;
     Node* temp = *head;
     *head = temp->next;
     free(temp);
}
```

2）删除尾结点。

例：

```
void delete_last(Node** head) {
    // 处理空链表的情况
    if (*head == NULL)
        return;
    // 处理只有一个结点的情况
    if ((*head)->next == NULL) {
        free(*head);
        *head = NULL;
        return;
    }
    // 找到倒数第二个结点
    Node* current = *head;
    while (current->next->next != NULL) {
        current = current->next;
    }
    Node* temp = current->next;
    // 删除尾结点
    current->next = NULL;
    free(temp);
}
```

3）删除指定结点：
删除单链表中值为target的结点。

例：

```
void delete_node(Node** head, int target) {
    // 处理空链表的情况
    if (*head == NULL)
        return;

    // 处理头指针所指结点是目标的情况
    if ((*head)->data == target) {
        delete_first(head);
        return;
    }

    // 寻找目标结点的前驱
    Node* current = *head;
    while (current->next != NULL && current->next->data != tar
        current = current->next;
    }

    // 找到目标结点
    if (current->next != NULL) {
        Node* temp = current->next;
        current->next = temp->next;
        free(temp);
    }
}
```

头结点是链表里的一个辅助结点，它在链表的第一个数据结点之前，头结点本身不储存实际的数据，它的主要作用是指向链表里的第一个数据结点

头结点是需要初始化的，一般会把头结点的next指针初始化为NULL，用来表示链表此时是空的

题目1、关于单链表的头结点，以下说法正确的是（ B ）

A. 头结点是链表的第一个数据结点

B. 头结点可以简化插入/删除操作

C. 头结点的data成员必须存储链表长度

D. 头结点不需要初始化

题目2、有以下程序:

```
#include <stdio.h>
main()
{
    struct node
    {
        int n;
        struct node *next;
    } *p;
    struct node x[3] = {{2, x+1}, {4, x+2}, {6, NULL}};
    p = x;
    printf("%d,", p->n);
    printf("%d\n", p->next->n);
}
```

程序运行后的输出结果是（ B ）

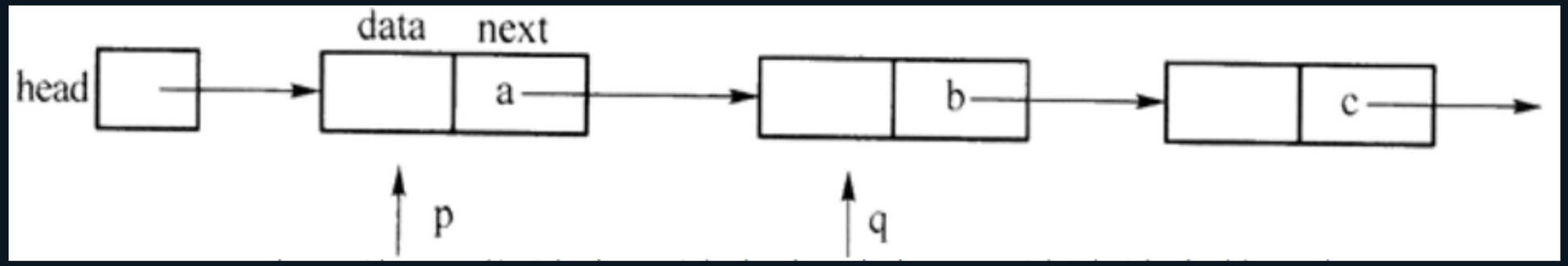
A. 2,3

B. 2,4

C. 3,4

D. 4,6

题目3、假定已建立以下数据链表结构，且指针p和q已指向如下图所示的结点：



则以下选项中可将q所指结点从链表中删除并释放该结点的语句是（D）

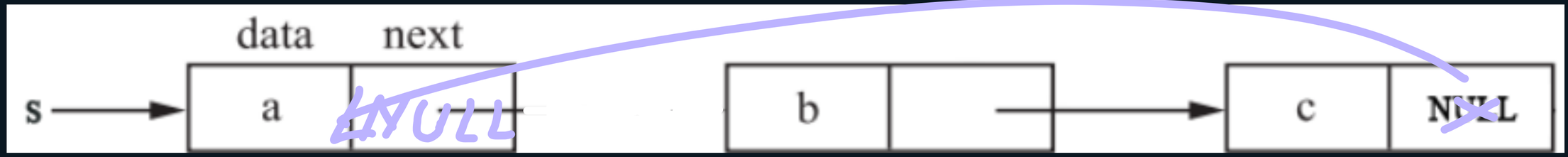
A．(*)p.next = (*q).next; free(p);

B．b = q->next; free(q);

C．p = q; free(q);

D．p->next = q->next; free(q);

题目4、程序中已构成如下图所示的不带头结点的单向链表结构，指针变量s、p、q均已正确定义，并用于指向链表结点，指针变量s总是作为头指针指向链表的第一个结点。



若有以下程序段

```
q = s;
s = s->next;
p = s;
while (p->next) p = p->next;
p->next = q;
q->next = NULL;
```

a -> b -> c ➡ b -> c ->a

该程序段实现的功能是（C）

A. 删除首结点

B. 尾结点成为首结点

C. 首结点成为尾结点

D. 删除尾结点

题目5、补全尾插法代码：

```
void insert_tail(Node* head, int value) {
    Node* new_node = (Node*)malloc(sizeof(Node));
    new_node->data = value;
    new_node->next = ____NULL____; //填空 1
    Node* p = head;
    while (p->next != NULL) {
        p = ____p->next____;  //填空 2
    }
    ____p->next____ = new_node;  //填空 3
}
```