链表与顺序表达的区别：

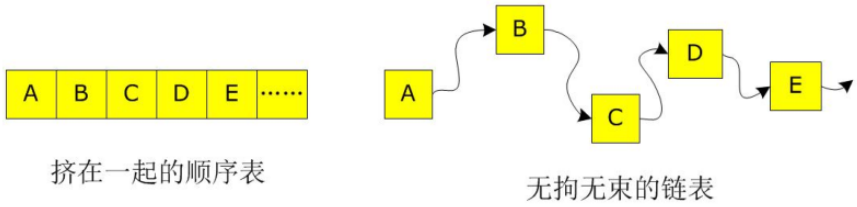
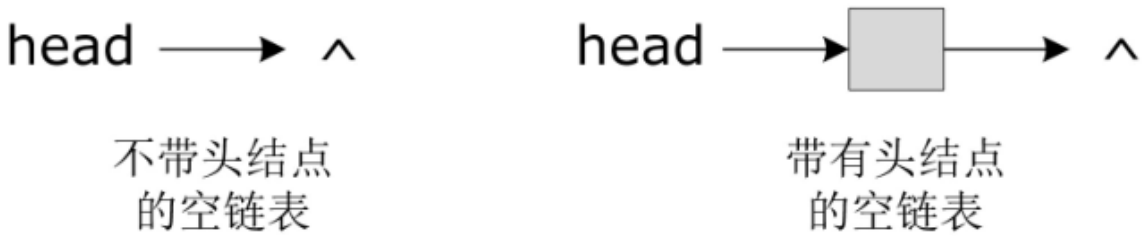
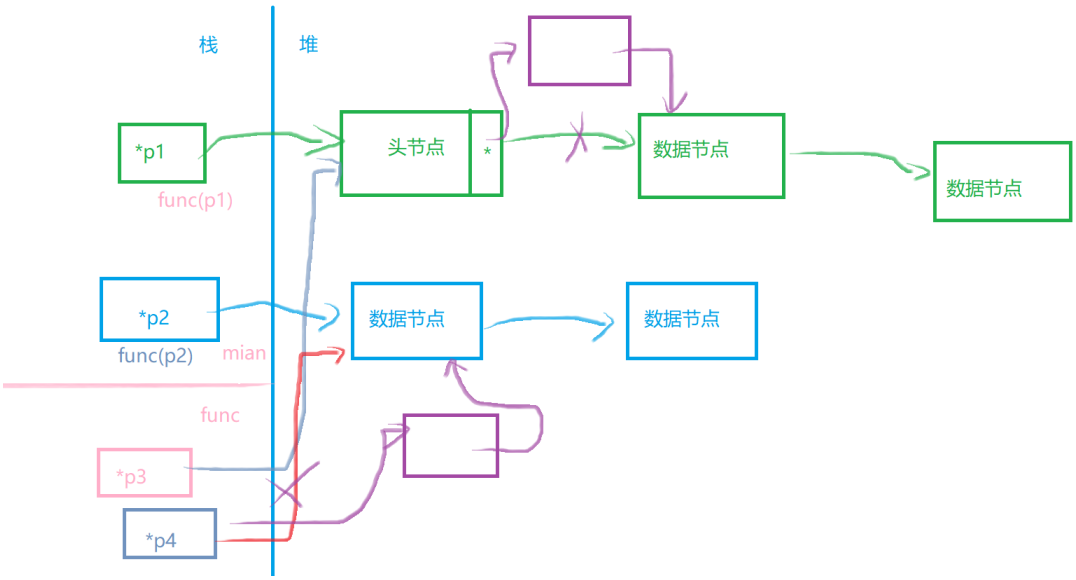


图 3-2 顺序表和链表

- 顺序表在存储数据的时候需要一整片连续的内存，而链表不需要连续的内存。
- 如果需要插入或者删除数据，对于顺序表来说很可能需要挪动大量的数据，而链表则不需要只需要改一下指针的指向即可。

单向链表的头节点设计：





注意：

在使用无头节点的链表时，如果新数需要插入到链表的开头位置，要注意返回p4指针（头指针），在使用有头节点的链表时则不需要注意该问题。

节点设计：

数据域用于存储数据， 指针域用来寻找下一个数据的入口地址。


```c
// 节点设计
typedef struct list
{
    Data_Type Num ;   //数据
    struct list * Next ; // 后继指针
}List , *P_List;
```

初始化:

在堆空间申请一个新的节点的内存， 并把指针指向NULL

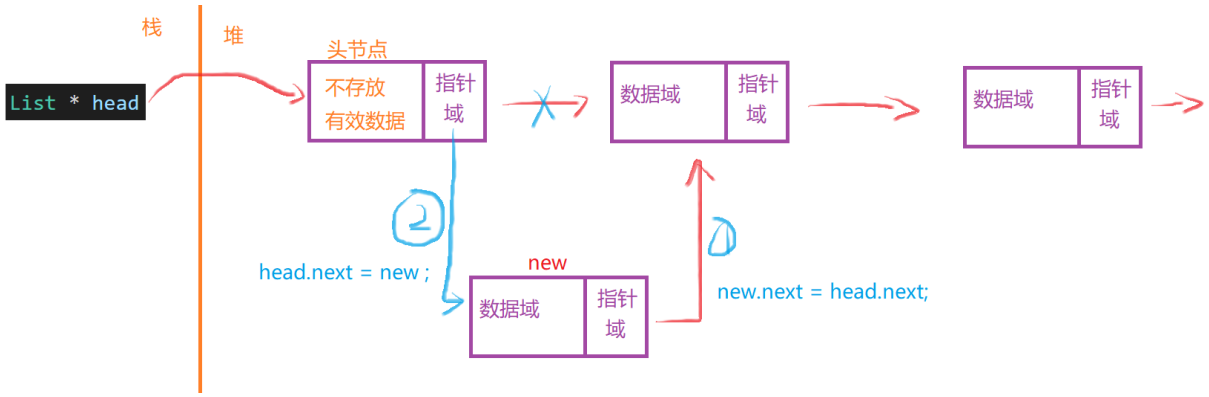
返回该堆空间的内存地址。

```c
P_List init_new_node(Data_Type new_data)
{
    // 申请一个头节点
    List * head = calloc(1,sizeof(List) );

    head->Num = new_data ;
    head->Next = NULL ; // 初始化后继指针指向空

    return head ;
}
```

插入数据：




```c
//头插数据
int add_2_list(P_List head , P_List new_node)
{
    if( head == NULL )
    {
        printf("链表头异常！！！\n");
        return -1 ;
    }

    //1 让新节点的后继指针指向头结点的后记节点
    new_node->Next = head->Next ;

    //2 让头节点的后继指针指向新节点
    head->Next = new_node ;

    return 0 ;
}


int add_2_list_tail(P_List head , P_List new_node)
{
    if( head == NULL )
    {
        printf("链表头异常！！！\n");
        return -1 ;
    }

    //遍历找到链表的末尾
    P_List tmp ;
    // while(tmp->Next != NULL )
    // {
```

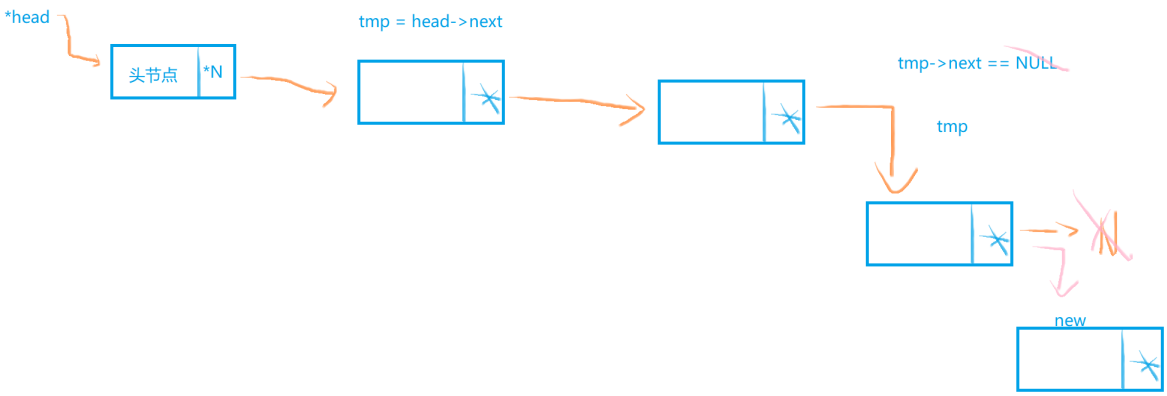
```
    //      tmp = tmp->Next;
    // }
    //      初始化              判断只要tmp->next==NULL 则表示到达末尾节点
    for ( tmp = head ; tmp->Next != NULL ; tmp = tmp->Next);

    //1 让新节点的后继指针指向头结点的后记节点
    new_node->Next = tmp->Next ;

    //2 让头节点的后继指针指向新节点
    tmp->Next = new_node ;


    return 0 ;

}
```



遍历显示数据：

```

int display_list(List * head)
{
    if(head->Next == NULL )
    {
        printf("该表为空！！！！\n");
        return -1 ;
    }

    List * tmp = head->Next ; // 把头结点的NEXT （第一个数据）

    while( tmp != NULL ) // 只要tmp 不等于NULL 就循环
```

```
    {
        printf("data:%d\n" , tmp->Num);
        tmp = tmp->Next ; // tmp 往后走一个
    }

    return 0 ;
}
```

查找节点:

```
//                  链表头          需要移除的内容如果为NULL 则只是单纯的显示链表
List * list_4_each(List * head , Data_Type * Num)
{
    if(head->Next == NULL )
    {
        printf("该表为空! ! ! ! \n");
        return NULL ;
    }

    List * tmp = head ; // 初始化tmp

    while( tmp->Next != NULL ) // 只要tmp 不等于NULL 就循环
    {
        if(Num == NULL ) // 如果为NULL 则直接输出
        {
            printf("data:%d\n" , tmp->Next->Num);
        }
        else // 否则需要进行的是查找数据
        {
            if(tmp->Next->Num == *Num)
            {
                return tmp ; // 返回的是需要删除节点的前一个节点
            }
        }
        tmp = tmp->Next ; // tmp 往后走一个
    }

    return NULL  ;
```

```
}
```

删除节点

```
List * del_4_list( List * head , Data_Type del_num)
{
    if( head->Next == NULL )
    {
        printf("该表已空！！\n");
        return NULL  ;
    }

    // 遍历查找需要移除的节点
    List * tmp = list_4_each(head , &del_num ); // 需要删除的前一个
    if(tmp == NULL )
    {
        printf("没有你要的豆腐！！\n");
        return NULL ;
    }
    List * del = tmp->Next ; // 需要删除的节点

    // 把找到的节点从链表中剔除
    tmp->Next = del->Next ;  // 让需要删除节点的前一个节点的后继指针指向需要删除节点的后继节点
    del->Next = NULL ; // 让del 的next不要再指向链表，而是指向空

    return del ;
}
```

移动数据：

思路：

1. 使用剔除节点的函数把需要移动的节点从表中剔除
2. 使用头插（在某个节点的后面插入）把刚才剔除的节点进行插入

```
void move_node(List * head ,  Data_Type move_data , Data_Type move_tag )
```

```
{
    // 找到需要移动的数据并把它从链表中剔除
    List* move_node = del_4_list(  head , move_data );

    // 找到目标位置
    List* tag_node = list_4_each( head , &move_tag ) ;

    // 把剔除的数据插入到 目标位置
    add_2_list_head( tag_node->Next  , move_node );

}
```

作业：

1. 先把单向不循环链表代码看懂（新建、插入、显示）
2. 尝试添加 查找、删除 、 修改 、排序等功能、
3. 输入数据，保持链表有序

预习：

双向

循环