链表

> 请忽视我的画工~~

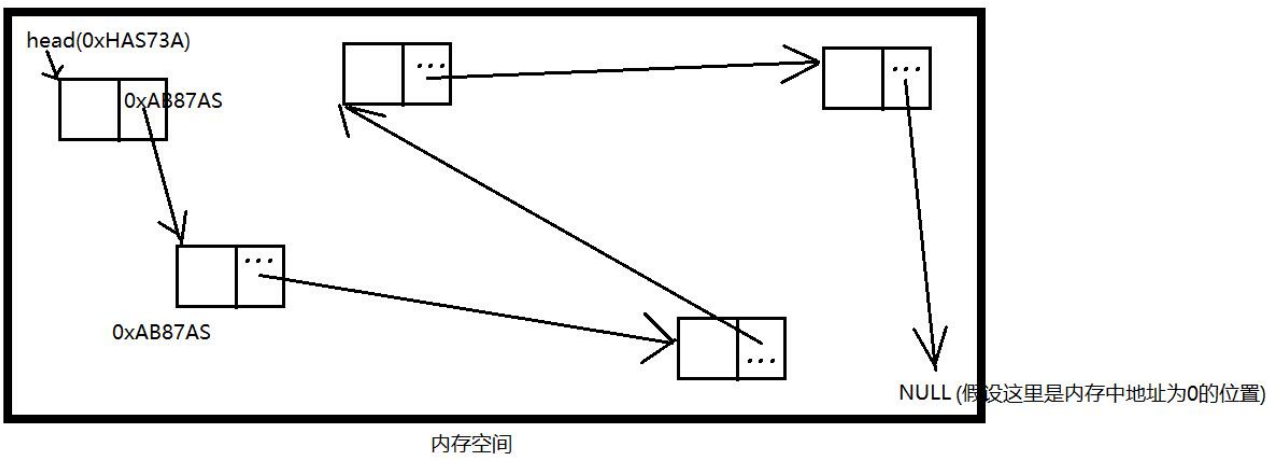
一、概论

1. 基本概念

1.1 概念

逻辑结构上一个挨一个的数据元素，但是物理存储却没有占用一段连续空间的结构。

示意图如下：



内存空间

由上图也可以看出，链表节点的构成：**数据域** 和 **指针域。**

- 数据域存储链表需要存储的数据
- 指针域存储后继节点在内存中的地址，即图中第一个节点指针域存储的 `0xAB87AS`，这个地址就是下一个节点的起始地址（由图中所示可以看出；其他的我省略没写）

由上图也可以看出，链表的节点在内存是散乱分布的，想要找到后继节点的办法就是查看当前节点的指针域存储的地址。尾节点的指针域存储NULL（NULL是内存中的一个特殊地址，相当于0），表示它后面没有节点了。

1.2 分类

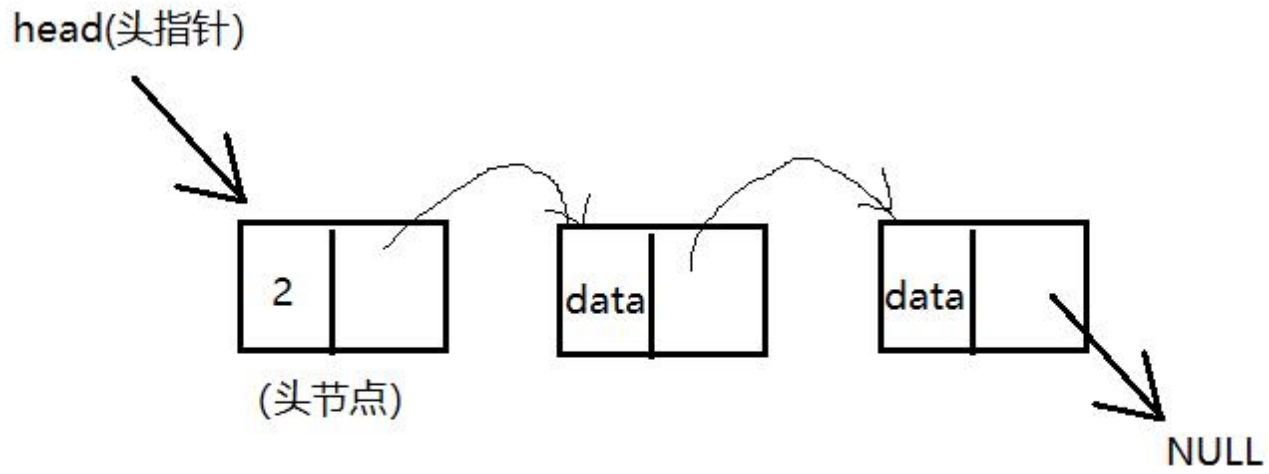
链表的分类既可以按照是否带头节点分为两类，也可以根据指针域的区别分为三类。

> 也可以混在一起分为六类~~

什么是头节点?

头节点和普通的节点唯一的区别就是**头节点的数据域不存放链表实际需要存放的数据**（一般可以存放当前链表的长度，也可以随便存些什么东西）

如下图：



带不带头节点有什么区别?

网上的说法是：设置头结点是为了保证处理第一个节点和后面的节点的时候设计的算法相同

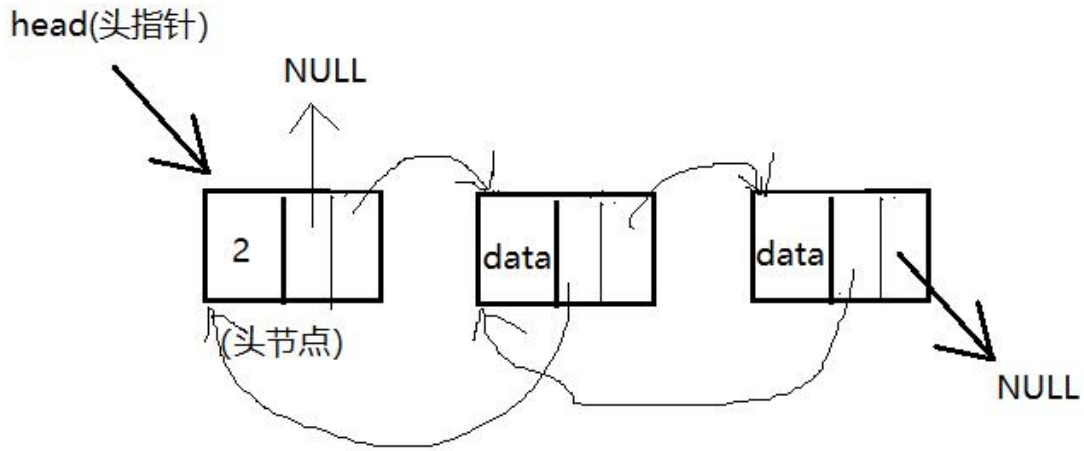
实际上带不带区别不大。

单链表

上面那个示意图就是单链表的逻辑模型，即只有一个指针域，存储后继节点的地址。

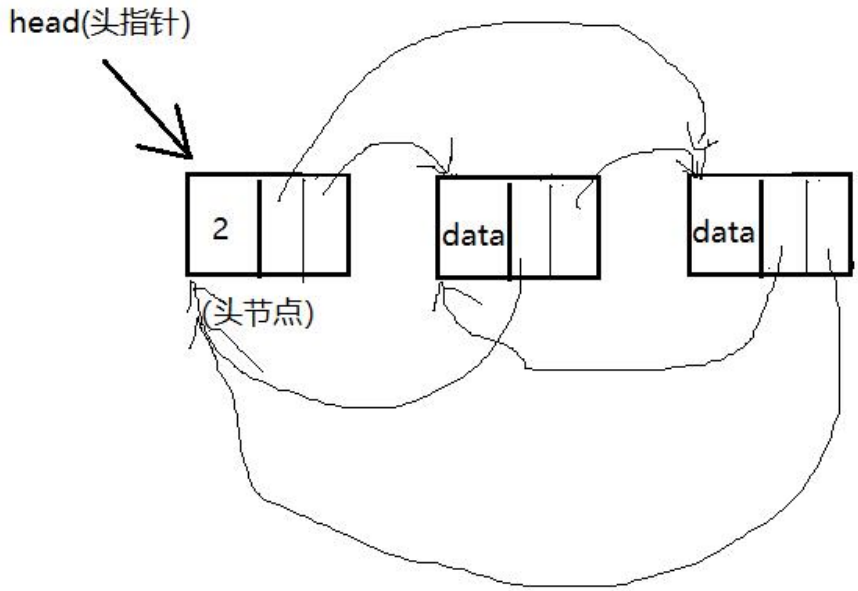
双链表

双链表的指针域有两个，一个存储前驱节点的地址，一个存储后继节点的地址。链表第一个节点的前驱指针域存储NULL。



双向循环链表

双向循环链表和双链表的唯一区别就是第一个节点的前驱指针域存储链表最后一个节点的地址，最后一个节点的后继指针域存储链表第一个节点的地址。



2.应用场所

链表在软件开发中使用到的频率很高，我简单的举几个实例：

- 其它的 **数据结构** 大部分都是基于链表实现的
- 计算机底层内存管理的 **内存池** 是基于链表实现的
- 太多了.....

使用场景：

- 频繁在数据结构的中间进行插入 / 删除元素操作的场景
- 整个数据结构存储容量需要不确定变化的场景

二、方法

定义节点

从链表的基本概念我们就可以得出节点类型包含两类数据：实际数据和地址。所以定义如下：

```
typedef int DataType;

typedef struct Node{
    DataType _elem;
    struct Node* _next;
}Node, *PNode;
```

解释一下：

众所周知，`typedef`是一个重命名的关键字，那么代码的含义就明显了，将`struct Node{...}`类型重命名为`Node`，将`struct Node {...}*`重命名为`PNode`。

为什么要这么做呢?

因为C语言使用结构体变量类型定义变量时必须要加上`struct`，（C++这点就比C做的好，因为可以不用加）为了后面书写方便，所以将节点类型重命名为一个简单易懂的名称。

> 注：这个方法在开发中非常常见，读者要仔细品味~

> 注：以下方法均以不带头节点的单链表为示例

初始化

链表的初始化就是将头指针置空。这一步看似鸡肋，但是如果没有这一步，后面可能会出现很多奇奇怪怪的bug。

原因也不难分析，定义指针的时候如果不初始化，就会出现 **野指针问题**。

```
void InitList(PNode *head){
    assert(head);
    *_head = NULL;
}
```

头插

头插就是将新节点插入到链表的头部。

头插的步骤：

- 检测参数正确性
- 如果头指针为NULL，分配一块节点大小的空间并赋值（可以理解为申请一个节点并将数据域赋为指定数据，将指针域指向NULL），然后让头指针指向这块空间，函数返回
- 如果头指针不为NULL，分配一块空间并赋值，定义一个指针指向这块空间
- 然后让这个新节点的指针域指向原链表第二个节点
- 然后让头指针的指针域指向新节点

```
void HeadInsertList(PNode *head, DataType elem){
    assert(head);
    if(NULL == *head){
        *head = (PNode)malloc(sizeof(Node));
        (*head)->_elem = elem;
        (*head)->_next = NULL;
    }else{
        PNode* New = (PNode)malloc(sizeof(Node));
        New->_elem = elem;
        New->_next = *head;
        *head = New;
    }
}
```

尾插

尾插就是将节点插入到链表的尾部。

尾插步骤：

- 检测参数的合法性
- 判断头指针是否为空，逻辑分流
- 如果头指针为空，分配一块节点大小的空间并赋值，然后让头指针指向这块空间，函数返回
- 如果头指针不为空， 定义一个指针遍历链表，找到链表的最后一个节点
- 分配一块空间并赋值，定义一个指针指向这块空间
- 让最后一个节点的指针域指向这块新空间

```
void TailInsertList(PNode *head, DataType elem){
    assert(head);
    if(NULL == *head){
        *head = (PNode)malloc(sizeof(Node));
        (*head)->_elem = elem;
        (*head)->_next = NULL;
    }else{
        PNode tmp = *head;
        while(tmp->_next != NULL){
            tmp = tmp->_next;
        }
        tmp->_next = (PNode)malloc(sizeof(Node));
        tmp->_next->_elem = elem;
        tmp->_next->_next = NULL;
    }
}
```

头删

头删就是删除链表第一个节点。

头删步骤：

- 检测参数的合法性
- 判断头指针是否为空，如果为空，则函数返回
- 如果不为空，定义一个指针指向链表第一个节点（即链表头指针指向的节点）
- 让头指针指向链表第二个节点
- 释放指向链表头节点的指针

```
void HeadDelList(PNode *head){
    assert(head);
    if(NULL == *head){
        return;
    }else{
        PNode tmp = *head;
        (*head) = (*head)->_next;
        free(tmp);
    }
}
```

尾删

尾删就是删除链表最后一个节点。

尾删步骤：

- 检测参数的合法性
- 判断头指针是否为空，为空则直接返回
- 如果头指针不为空，定义两个指针 `tmp` 和 `tmp_pre`，`tmp` 指向链表头节点
- 用这两个指针遍历链表，（`tmp_pre` 始终指向 `tmp` 的前驱结点）最终 `tmp` 指向链表最后一个节点，`tmp_pre` 指向链表倒数第二个节点
- 释放指针 `tmp` 指向的空间，将 `tmp_pre` 指向空间的指针域指向NULL

```
void TailDelList(PNode *head){
    assert(head);
    if(NULL == *head){
        return;
    }else{
        PNode tmp = *head;
        PNode tmp_pre;
        while(tmp->_next != NULL){
            tmp_pre = tmp;
            tmp = tmp->_next;
        }
        tmp->_pre->_next = NULL;
        free(tmp);
    }
}
```

指定插入

将数据插入到链表中指定的位置。

步骤：

- 检测参数的合法性
- 判断头指针是否为NULL，如果是，则直接返回
- 如果头指针不是NULL，则和删除的步骤类似，定义两个指针遍历链表
- 如果 `tmp` 没有找到参数中的指定节点，则函数返回
- 否则，定义一个指针 `new_node`，让 `new_node` 指向一块新分配的空间并赋值
- 让 `new_node` 的指针域指向 `tmp`，让 `tmp_pre` 的指针域指向这块新空间

```
void InsertList(PNode *head, PNode pos, DataType elem){
    assert(head);
    assert(pos);
    if(NULL == *head){
        return;
    }else{
        new_node = (PNode)malloc(sizeof(Node));
        new_node->_elem = elem;
```

```
        new_node->_next = NULL;
        PNode tmp_pre = NULL;
        PNode tmp = *head;
        while (tmp != NULL && tmp != pos){
            tmp_pre = tmp;
            tmp = tmp->_Next;
        }
        if(NULL == tmp){
            return;
        }
        new_node->_next= tmp;
        tmp_pre->_next = new_node;
    }
}
```

删除指定

删除链表中指定的数据。

步骤：

- 检测参数的合法性
- 判断头指针是否为NULL，如果是，则函数返回
- 如果头指针不是NULL，则和添加指定的步骤类似
- 如果没找到指定数据，则函数返回
- 否则， 将`tmp_pre`的指针域指向`tmp`的指针域，即让包含指定数据节点的上一个节点的指针域指向包含指定数据节点的下一个节点（相当于把指定节点从链表中摘了出来）
- 释放包含指定数据节点的空间

```
void DeleteList(PNode *head, DataType elem){
    assert(head);
    if(NULL == *head){
        return
    }else{
        PNode tmp = *head;
        PNode tmp_pre = NULL;
        while (tmp != NULL && tmp->_elem != elem){
            tmp_pre = tmp;
            tmp = tmp->_next;
        }
        if(NULL == tmp){
            return;
        }
        tmp_pre->_next = tmp->_Next;
        free(tmp);
    }
}
```

销毁

销毁链表的步骤：

- 检测函数参数的正确性
- 依次释放链表中每个节点的空间
- 让头指针指向NULL

```
void DetmpoyList(PNode* head){
    assert(head);
    if(NULL == *head){
        return;
    }else{
        PNode tmp = NULL;
        PNode tmp_pre = *head;
        while (tmp_pre){
            tmp = tmp_pre;
            tmp_pre = tmp_pre->_Next;
            free(tmp);
        }
        *head = NULL;
    }
}
```

顺序表和链表对比

看完了链表和顺序表，想必读者已经清楚顺序表物理上是存储在一大块空间，而链表是由分散存储的小块空间组成的。所以可以得出以下的结论：

- 顺序表支持随机存取，即用下标就可以访问指定的数据。而链表则需要遍历寻址。相比之下，顺序表的性能就要高于链表。

但是，顺序表如果要支持扩容，那么就要先申请一块更大的空间，然后将旧的数据拷贝到新的空间，最后销毁掉原来的空间，这一个流程下来系统开销是比较大的。而链表扩容，则只要申请一块空间，然后填充数据域和指针域即可。所以从这个角度看：**链表的性能要优于顺序表。**

根据上面的讨论，我们可以得出最终结论：

- 如果不确定线性结构的容量，那么用链表比较合适一些。
- 如果需要获得随机存取的时间效率，那么用顺序表合适一些。

有些人会想，那为什么不一开始就把顺序表的容量申请大一些呢?

这样显然有两个问题：

1. 如果你申请了4K的空间，但是实际上只用到了几百个字节，那无疑是对内存资源的极大浪费。
2. 需求是变化的，谁也不能保证当时的足够大以后就会够用，历史上无数个例子证明了这一点：
    - 早些年的PC机内存只有8M，现在看来实在是太小了
    - IPv4地址有43亿多，但是现在依然面临不够用的问题
    - ……