链表

本页面将简要介绍链表。

引入

链表是一种用于存储数据的数据结构，通过如链条一般的指针来连接元素。它的特点是插入与删除数据十分方便，但寻找与读取数据的表现欠佳。

与数组的区别

链表和数组都可用于存储数据。与链表不同，数组将所有元素按次序依次存储。不同的存储结构令它们有了不同的优势：

链表因其链状的结构，能方便地删除、插入数据，操作次数是 $O(1)$。但也因为这样，寻找、读取数据的效率不如数组高，在随机访问数据中的操作次数是 $O(n)$。

数组可以方便地寻找并读取数据，在随机访问中操作次数是 $O(1)$。但删除、插入的操作次数是 $O(n)$ 次。

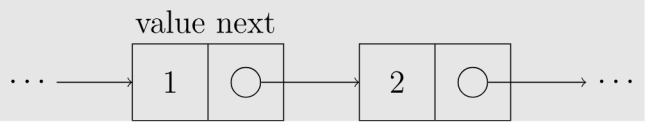
构建链表

Tip

构建链表时，使用指针的部分比较抽象，光靠文字描述和代码可能难以理解，建议配合作图来理解。

单向链表

单向链表中包含数据域和指针域，其中数据域用于存放数据，指针域用来连接当前结点和下一节点。



实现

C++

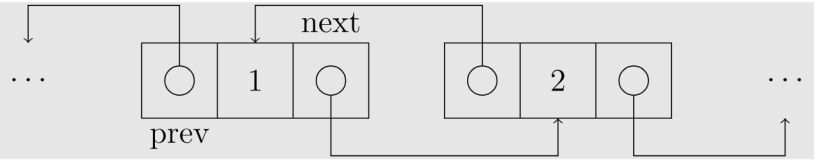
```cpp
struct Node {
  int value;
  Node *next;
};
```

Python

```python
class Node:
    def __init__(self, value=None, next=None):
        self.value = value
        self.next = next
```

双向链表

双向链表中同样有数据域和指针域。不同之处在于，指针域有左右（或上一个、下一个）之分，用来连接上一个结点、当前结点、下一个结点。



实现

C++

```cpp
struct Node {
  int value;
  Node *left;
  Node *right;
};
```

Python

```python
class Node:
    def __init__(self, value=None, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
```

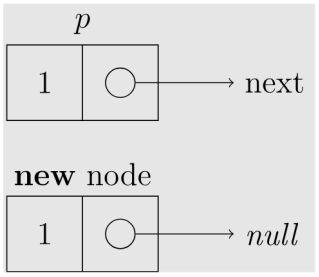
向链表中插入（写入）数据

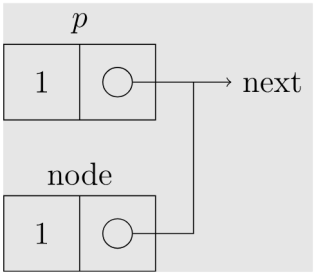
单向链表

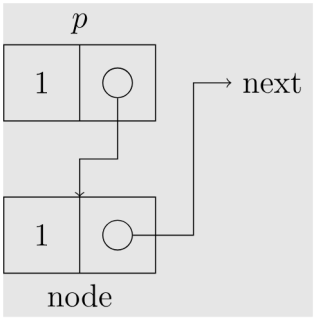
流程大致如下：

1. 初始化待插入的数据 `node`；
2. 将 `node` 的 `next` 指针指向 `p` 的下一个结点；
3. 将 `p` 的 `next` 指针指向 `node`。

具体过程可参考下图：

1. 

2. 

3. 

代码实现如下：

实现

C++

```cpp
void insertNode(int i, Node *p) {
  Node *node = new Node;
  node->value = i;
  node->next = p->next;
  p->next = node;
}
```

Python

```python
def insertNode(i, p):
    node = Node()
    node.value = i
    node.next = p.next
    p.next = node
```

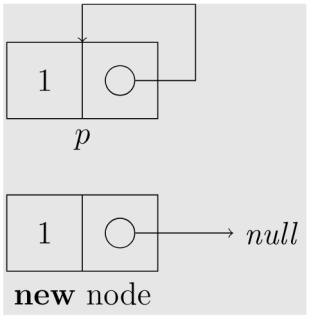
单向循环链表

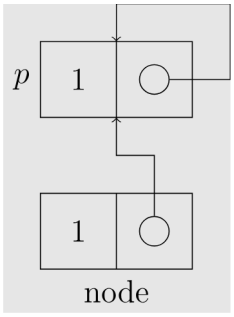
将链表的头尾连接起来，链表就变成了循环链表。由于链表首尾相连，在插入数据时需要判断原链表是否为空：为空则自身循环，不为空则正常插入数据。

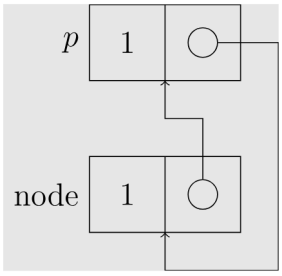
大致流程如下：

1. 初始化待插入的数据 `node`；
2. 判断给定链表 `p` 是否为空；
3. 若为空，则将 `node` 的 `next` 指针和 `p` 都指向自己；
4. 否则，将 `node` 的 `next` 指针指向 `p` 的下一个结点；
5. 将 `p` 的 `next` 指针指向 `node`。

具体过程可参考下图：

1. 

2. 

3. 

代码实现如下：

实现

C++

```
void insertNode(int i, Node *p) {
  Node *node = new Node;
  node->value = i;
  node->next = NULL;
  if (p == NULL) {
    p = node;
    node->next = node;
  } else {
    node->next = p->next;
    p->next = node;
  }
}
```

Python

```
def insertNode(i, p):
    node = Node()
    node.value = i
    node.next = None
    if p == None:
        p = node
        node.next = node
    else:
        node.next = p.next
        p.next = node
```

双向循环链表

在向双向循环链表插入数据时，除了要判断给定链表是否为空外，还要同时修改左、右两个指针。

大致流程如下：

1. 初始化待插入的数据 `node`；
2. 判断给定链表 `p` 是否为空；
3. 若为空，则将 `node` 的 `left` 和 `right` 指针，以及 `p` 都指向自己；
4. 否则，将 `node` 的 `left` 指针指向 `p`；
5. 将 `node` 的 `right` 指针指向 `p` 的右结点；
6. 将 `p` 右结点的 `left` 指针指向 `node`；
7. 将 `p` 的 `right` 指针指向 `node`。

代码实现如下：

实现

C++

```cpp
void insertNode(int i, Node *p) {
  Node *node = new Node;
  node->value = i;
  if (p == NULL) {
    p = node;
    node->left = node;
    node->right = node;
  } else {
    node->left = p;
    node->right = p->right;
    p->right->left = node;
    p->right = node;
  }
}
```

Python

```python
def insertNode(i, p):
    node = Node()
    node.value = i
    if p == None:
        p = node
        node.left = node
        node.right = node
    else:
        node.left = p
        node.right = p.right
        p.right.left = node
        p.right = node
```

从链表中删除数据

单向（循环）链表

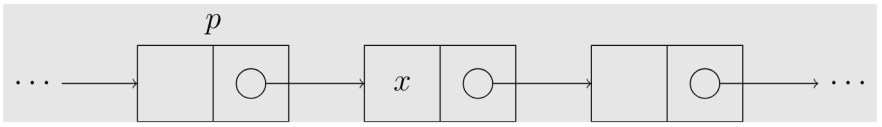
设待删除结点为 `p`，从链表中删除它时，将 `p` 的下一个结点 `p->next` 的值覆盖给 `p` 即可，与此同时更新 `p` 的下下个结点。

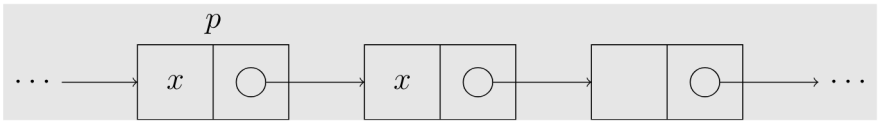
流程大致如下：

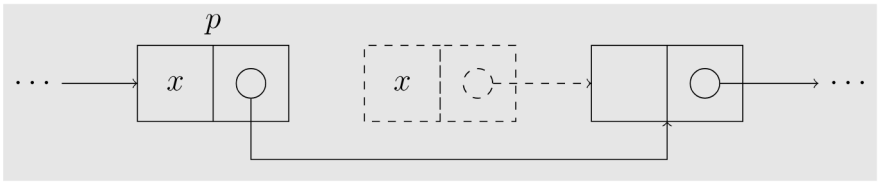
1. 将 `p` 下一个结点的值赋给 `p`，以抹掉 `p->value`；
2. 新建一个临时结点 `t` 存放 `p->next` 的地址；
3. 将 `p` 的 `next` 指针指向 `p` 的下下个结点，以抹掉 `p->next`；

4. 删除 `t`。此时虽然原结点 `p` 的地址还在使用，删除的是原结点 `p->next` 的地址，但 `p` 的数据被 `p->next` 覆盖， `p` 名存实亡。

具体过程可参考下图：

1. 

2. 

3. 

代码实现如下：

实现

C++

```cpp
void deleteNode(Node *p) {
  p->value = p->next->value;
  Node *t = p->next;
  p->next = p->next->next;
  delete t;
}
```

Python

```python
def deleteNode(p):
    p.value = p.next.value
    p.next = p.next.next
```

双向循环链表

流程大致如下：

1. 将 `p` 左结点的右指针指向 `p` 的右节点；
2. 将 `p` 右结点的左指针指向 `p` 的左节点；
3. 新建一个临时结点 `t` 存放 `p` 的地址；
4. 将 `p` 的右节点地址赋给 `p`，以避免 `p` 变成悬垂指针；

5. 删除 `t`。

代码实现如下：

实现

C++

```cpp
void deleteNode(Node *&p) {
  p->left->right = p->right;
  p->right->left = p->left;
  Node *t = p;
  p = p->right;
  delete t;
}
```

Python

```python
def deleteNode(p):
    p.left.right = p.right
    p.right.left = p.left
    p = p.right
```

技巧

异或链表

异或链表（XOR Linked List）本质上还是 **双向链表**，但它利用按位异或的值，仅使用一个指针的内存大小便可以实现双向链表的功能。

我们在结构 `Node` 中定义 `lr = left ^ right`，即前后两个元素地址的 **按位异或值**。正向遍历时用前一个元素的地址异 或当前节点的 `lr` 可得到后一个元素的地址，反向遍历时用后一个元素的地址异或当前节点的 `lr` 又可得到前一个的元素地址。 这样一来，便可以用一半的内存实现双向链表同样的功能。
