--- title: 算法专题：链表操作及相关常见问题 toc: true categories:

- 算法 tags:
- ACM date: 2017-03-23 10:10:03

链表是一种比较基础的数据结构，主要类型有单链表，双链表，循环链表等。 链表的实现可以使用数组；也可以使用结构体和指针实现。

链表的基础知识

链表的基本操作

C++11的标准中，结构体和类并没有本质的不同。所有可以用结构体和类来定义节点的数据结构。 单链表通常包含一个`val`和一个指向下一个节点的指针`*next` 双向链表通常包含一个`val`和两个指针`*pre`，`*next`,分别指向前面的节点和后面的节点。

链表节点结构的定义

单链表

```
//单链表节点的定义-使用结构体
struct Node{
    int value;
    Node *next;
    Node() : value(0),next(nullptr){}
    Node(int v) : value(v),next(nullptr){}
};
```

```
//使用类
class ListNode {
public:
     int val;
     ListNode \*next;
     ListNode(int val) {
             this->val = val;
         this->next = NULL;
     }
 }
```

双向链表

```
//双向链表
struct BiNode{
    int value;
    BiNode *pre;
    BiNode *next;
    BiNode() : value(0),pre(nullptr),next(nullptr){}
    BiNode(int v) : value(v),pre(nullptr),next(nullptr){}
};
```

链表的创建

单链表

首先判断是不是空，如果是空，返回空链表 如果不是空，创建第一个元素的节点，并用头指针指向它，日后返回这个头指针； 然后从第二个元素开始，创建新的节点，把这个节点赋值个前一个节点的next指针，然后更新tmp指向当前新创建的节点。 核心的语句：

```
Node *node = new Node(v[i]);
tmp->next = node;
tmp = tmp->next;
```

```
Node* createLinks(vector<int> v){
    if(v.empty()) return nullptr;
    Node *head = new Node(v[0]); //头节点，最后需要返回创建的链表的头结点，所以头结点需要保存下来
    Node *tmp = head;
    for(int i=1;i<v.size();++i){
```

```
        Node *node = new Node(v[i]);
        tmp->next = node;
        tmp = tmp->next;
    }
    return head;
}
```

双链表

双向链表的创建需要返回两个指针，一个指向头部，便于正向遍历链表；另外一个指向尾部，便于逆向遍历链表 首先判断是不是空，如果是空，返回空链表 用第一个元素创建节点，并用头节点指向它 创建临时变量指向第一个节点，此临时变量一直指向当前创建的节点 从数组的第二个元素开始创建新的节点，创建的节点的pre设置为tmp, 然后把tmp的next赋值成node,即让tmp所指向的节点连接到当前创建的节点上 更新tmp使它指向当前新创建的节点。 循环体结束后，tmp正好指向最会一个创建的节点，所以把tmp赋值给tail 返回head和tail; 关键语句：

```
  BiNode *node = new BiNode(v[i]);
  node->pre = tmp;
  tmp->next = node;
  tmp = tmp->next;
```

```
pair<BiNode*,BiNode*> createBiNode(vector<int> v){
    if(v.empty()) return {nullptr,nullptr};
    BiNode *head = new BiNode(v[0]);
    BiNode *tmp = head;
    for(int i=1;i<v.size();i++){
        BiNode *node = new BiNode(v[i]);
                node->pre = tmp;
         tmp->next = node;
         tmp = tmp->next;
    }
    BiNode *tail = tmp;
    return {head,tail};
}
```

循环链表

循环链表就是结尾的元素又指向了开头的元素。所以创建过程基本上是一样的，指示在结尾的处理稍有不同。 循环单链表只需要在结尾添加 `tmp->next = head;` 循环双向链表只需要在结尾添加 `tail->next = head; // 尾元素的后面是头元素 head->pre = tail; // 头元素的前面是尾元素` .

```
//创建循环单链表
Node* createLinksCycle(vector<int> v){
    if(v.empty()) return nullptr;
    Node *head = new Node(v[0]); //头节点，最后需要返回创建的链表的头结点，所以头结点需要保存下来
    Node *tmp = head;
    for(int i=1;i<v.size();++i){
        Node *node = new Node(v[i]);
        tmp->next = node;
        tmp = tmp->next;
    }
    tmp->next = head; // 让结尾指向开头的第一个元素
    return head;
}
```

```
//创建循环双链表
pair<BiNode*,BiNode*> createBiNodeCycle(vector<int> v){
    if(v.empty()) return {nullptr,nullptr};
    BiNode *head = new BiNode(v[0]);
    BiNode *tmp = head;
    for(int i=1;i<v.size();i++){
        BiNode *node = new BiNode(v[i]);
        tmp->next = node;
        node->pre = tmp;
        tmp = tmp->next;
    }
    BiNode *tail = tmp;
    tail->next = head;  // 尾元素的后面是头元素
    head->pre = tail;  // 头元素的前面是尾元素
    return {head,tail};
}
```

链表的插入

单链表

要在单链表中插入一个节点，只需要知道插入位置的前一个节点就可以了。假设c指向了插入位置的前一个节点，n是新创建的节点 那么插入的方法是：

```
n->next = c->next ; //新节点n的后一个节点设置成c的后一个节点
c->next = n; //c节点指向新插入的节点
```

注意如果插入的位置是链表的头部，需要单独处理。

```
n->next = head; //新插入的节点的下一个节点连接到头节点
head = n; //头结点编程新插入的节点
```

双链表

双链表既可以向前访问，也可以向后访问。所以插入一个节点，只需要知道要插入的位置的前一个节点，或者要插入位置的后一个节点就可以了。 假设c代表要插入位置的前一个节点；n代表新插入的节点，b代表插入位置的后一个节点。 利用c的插入方法是：

```
n->next = c->next;
c->next->pre = n; // 这两句将新的节点和后面的节点连接起来
n->pre = c;
c->next = n; //这两句将c和n连接起来
```

如果要在头节点之前插入，会发现头结点之前没有节点了，所以不能使用上面的语句，需要另外处理：

```
n->next = head;
head->pre = n;
head = n;
```

利用b的插入方法是：

```
n->pre = b->pre;
b->pre->next = n; // 把新的节点和b前面的节点连接起来
n->next = b;
b->pre = n;  // 把新的节点和b连接起来
```

如果在最后一个节点后面插入节点，会发现没有b,这个时候需要另外处理：

```
n->pre = tail;
tail->next = n;
tail = n;
```

循环链表

循环链表的插入和删除没有了像上面不循环的链表那样需要另外处理头部和尾部的情况。所以插入操作更加简单。 如果插入的是头节点的话，注意head的更新情况即可。

链表的删除

单链表

删除单链表的某一个节点，只需要知道该节点的前一个节点。假设要删除的节点的前一个节点是c. 删除操作：`c->next = c->next->next;` 如果删除的是第一个节点，则直接`head = head->next;` 如果删除的是最后一个节点，则直接`c->next=nullptr`.

双链表

同插入类似，双链表的删除只需要知道要删除的节点的前一个节点或者后一个节点就可以了。假设c是前一个节点，b是后一个节点。 删除操作1：

```
c->next = c->next->next;
c->next->next->pre = c;
```

删除操作2：

```
b->pre = b->pre->pre;
```

```
b->pre->pre->next = b;
```

删除首节点和尾节点同样需要特殊考虑； 删除首节点：`b->pre = nullptr;head = b`；删除尾节点：`c->next=nullptr;tail=c;` 需要的注意的是，这样做只是从head和tail开始访问的话，再也访问不到被删除的节点，但是实际上，从删除的节点还有指针指向这个链表的某个节点。 如果想要删除干净，就是彻底断开连接，可以使用下面的语句： 删除首节点：`b->pre->next=nullptr; b->pre=nullptr;head=b;` 删除尾节点：`c->next->pre=nullptr;c->next=nullptr;tail=c;`

循环链表

循环链表的插入和删除同样不需要考虑特殊的情况，如果插入的是头节点的话，注意head的更新情况即可。

链表的替换

想要更新链表中的某个值，只需要查找到该节点c，然后执行`c->val = new_value;`即可。

链表的反向

单链表

基本思路是设置三个指针pre,current,next分别代表相邻的三个节点。 每次循环执行下面的操作：

```
next = current->next; // 首先记录current的下一个节点
current->next = pre; // curent的下一个节点设置为前一个节点
pre = current; // pre向前推进一个元素
current = next; //current像前推进一个元素
```

```
Node* notLinks(Node* head){
    if(head == nullptr || head->next == nullptr) return head;
    Node *current = head;
    Node *pre=nullptr;
    Node *next= nullptr;
    while(current){
        next = current->next; // 首先记录current的下一个节点
        current->next = pre; // curent的下一个节点设置为前一个节点
        pre = current; // pre向前推进一个元素
        current = next; //current像前推进一个元素
    }
    return pre; //返回指向新链表的头节点
}
```

双链表

双向链表的反转与单向链表的思路一致，设置三个指针pre,current,next分别指向相邻的三个节点； 每次循环这样操作：

```
next = current->next; // 首先记录current的下一个节点
current->next = pre; // curent的下一个节点设置为前一个节点
current->pre = next;
pre = current; // pre向前推进一个元素
current = next; //current像前推进一个元素
```

```
BiNode* notBiLinks(BiNode* head){
    if(head == nullptr || head->next == nullptr) return head;
    BiNode *current = head;
    BiNode *pre=nullptr;
    BiNode *next= nullptr;
    while(current){
        next = current->next; // 首先记录current的下一个节点
        current->next = pre; // curent的下一个节点设置为前一个节点
        current->pre = next;
        pre = current; // pre向前推进一个元素
        current = next; //current像前推进一个元素
    }
    return pre; //返回指向新链表的头节点
}
```

当然，在逻辑上双向链表是没有正反的，以上所说的反向是在相同的输出函数下，输出的顺序正好相反。 也可以利用这一点反转双向链表，只需要把链表节点的指针pre,next交换一下即可。具体代码如下：

```
BiNode *tmp = current->pre;
```

```
current->pre = current->next;
current->next = tmp;  // 以上三行交换两个变量的值
newHead = current;   //存储下来当前节点，日后返回该节点
current = current->pre; //处理下一个节点
```

```
BiNode* notBiLinks2(BiNode* head){
    if(head == nullptr || head->next == nullptr) return head;
    BiNode *current = head;
    BiNode *newHead;
    while(current){
        BiNode *tmp = current->pre;
        current->pre = current->next;
        current->next = tmp;
        newHead = current;
        current = current->pre;
    }
    return newHead; //返回指向新链表的头节点
}
```

循环链表

循环链表的反转没有什么实际的意义，只是用来练习链表的处理。具体的步骤和上面的基本相同。

链表的可视化

打印单链表

```
void printLinks(Node *head) {
    if (head == nullptr) {
        cout << "empty" << endl;
        return;
    }
    Node *tmp = head;
    while (tmp->next != nullptr) {
        cout << tmp->value << "->";
        tmp = tmp->next;
    }
    cout << tmp->value << endl;
}
```

打印出来的样式

```
1->2->3->4->5->6
```

打印双向链表

```
void printBiLinks(BiNode *head){
    if(head == nullptr){cout<<"empty"<<endl;return;}
    BiNode *tmp = head;
    while(tmp->next != nullptr){
        cout<<"["<<tmp->value<<"]"<<"<->";
        tmp = tmp->next;
    }
    cout<<"["<<tmp->value<<"]"<<endl;
    cout<<endl;
}
```

打印出来的样式

```
[1]<->[2]<->[3]<->[4]<->[5]<->[6]
```

单链表类的定义

```
class Links{
public:
    Node *head; // 头节点
    Links() : head(nullptr){}
```

```
    //创建链表
    Links(vector<int> v) {
        if(v.empty()){
            head = nullptr;
        }else {
            Node *first = new Node(v[0]);
            head = first;
            Node *tmp = first;
            for (int i = 1; i < v.size(); ++i) {
                Node *node = new Node(v[i]);
                tmp->next = node;
                tmp = node;
            }
        }
    }
    //打印链表
    void printLinks() {
        if (head == nullptr) {
            cout << "empty" << endl;
            return;
        }
        Node *tmp = head;
        while (tmp->next != nullptr) {
            cout << tmp->value << "->";
            tmp = tmp->next;
        }
        cout << tmp->value << endl;
    }
    //反转这个链表
    void inverseLinks(){
        Node *pre = nullptr;
        Node *next = nullptr;
        while(head!= nullptr){
            next = head->next;
            head->next = pre;
            pre = head;
            head = next;
        }
        head = pre;
    }
};
```

双向链表类的定义

```
//双向链表的实现
class BiLinks{
public:
    BiNode *head;
    BiNode *tail;
    BiLinks() : head(nullptr),tail(nullptr){}
    BiLinks(vector<int> v){
        BiNode *first = new BiNode(v[0]);
        head = first;
        BiNode *tmp = head;
        for(int i=1;i<v.size();i++){
            BiNode *node = new BiNode(v[i]);
            tmp->next = node;
            node->pre = tmp;
            tmp = tmp->next;
        }
        tail = tmp;
    }
    void printBiLinks(){
        if(head == nullptr || tail == nullptr){cout<<"empty!"<<endl;return;}
        BiNode *tmp = head;
        while(tmp->next != nullptr){
            cout<<"["<<tmp->value<<"]"<<"->";
            tmp = tmp->next;
        }
        cout<<"["<<tmp->value<<"]"<<endl;
        tmp = tail;
        while(tmp->pre!= nullptr){
            tmp = tmp->pre;
        }
        while(tmp->next!= nullptr){
```

```
            cout<<tmp->value<<"<->";
            tmp = tmp->next;
        }
        cout<< tmp->value <<endl;
        cout<<endl;
    }
    void inverseBiLinks(){
        BiNode *pre = nullptr;
        BiNode *next = nullptr;
        while(head!= nullptr){
            next = head->next;
            head->next= pre;
            head->pre = next;
            pre = head;
            head = next;
        }
        head = pre;
    }
};
```

链表操作的完整示例

```
#include <iostream>
#include <vector>
using namespace std;

struct Node{
    int value;
    Node *next;
    Node() : value(0),next(nullptr){}
    Node(int v) : value(v),next(nullptr){}
};
//双向链表
struct BiNode{
    int value;
    BiNode *pre;
    BiNode *next;
    BiNode() : value(0),pre(nullptr),next(nullptr){}
    BiNode(int v) : value(v),pre(nullptr),next(nullptr){}
};
Node* createLinks(vector<int> v){
    if(v.empty()) return nullptr;
    Node *head = new Node(v[0]); //头节点，最后需要返回创建的链表的头结点，所以头结点需要保存下来
    Node *tmp = head;
    for(int i=1;i<v.size();++i){
        Node *node = new Node(v[i]);
        tmp->next = node;
        tmp = tmp->next;
    }
    return head;
}
//创建循环单链表
Node* createLinksCycle(vector<int> v){
    if(v.empty()) return nullptr;
    Node *head = new Node(v[0]); //头节点，最后需要返回创建的链表的头结点，所以头结点需要保存下来
    Node *tmp = head;
    for(int i=1;i<v.size();++i){
        Node *node = new Node(v[i]);
        tmp->next = node;
        tmp = tmp->next;
    }
    tmp->next = head; // 让结尾指向开头的第一个元素
    return head;
}
pair<BiNode*,BiNode*> createBiNode(vector<int> v){
    if(v.empty()) return {nullptr,nullptr};
    BiNode *head = new BiNode(v[0]);
    BiNode *tmp = head;
    for(int i=1;i<v.size();i++){
        BiNode *node = new BiNode(v[i]);
        tmp->next = node;
        node->pre = tmp;
        tmp = tmp->next;
    }
    BiNode *tail = tmp;
```

```
    return {head,tail};
}
//创建循环双链表
pair<BiNode*,BiNode*> createBiNodeCycle(vector<int> v){
    if(v.empty()) return {nullptr,nullptr};
    BiNode *head = new BiNode(v[0]);
    BiNode *tmp = head;
    for(int i=1;i<v.size();i++){
        BiNode *node = new BiNode(v[i]);
        tmp->next = node;
        node->pre = tmp;
        tmp = tmp->next;
    }
    BiNode *tail = tmp;
    tail->next = head;  // 尾元素的后面是头元素
    head->pre = tail;  // 头元素的前面是尾元素
    return {head,tail};
}
//反转单向链表
Node* notLinks(Node* head){
    if(head == nullptr || head->next == nullptr) return head;
    Node *current = head;
    Node *pre=nullptr;
    Node *next= nullptr;
    while(current){
        next = current->next; // 首先记录current的下一个节点
        current->next = pre; // curent的下一个节点设置为前一个节点
        pre = current; // pre向前推进一个元素
        current = next; //current像前推进一个元素
    }
    return pre; //返回指向新链表的头节点
}
BiNode* notBiLinks(BiNode* head){
    if(head == nullptr || head->next == nullptr) return head;
    BiNode *current = head;
    BiNode *pre=nullptr;
    BiNode *next= nullptr;
    while(current){
        next = current->next; // 首先记录current的下一个节点
        current->next = pre; // curent的下一个节点设置为前一个节点
        current->pre = next;
        pre = current; // pre向前推进一个元素
        current = next; //current像前推进一个元素
    }
    return pre; //返回指向新链表的头节点
}
BiNode* notBiLinks2(BiNode* head){
    if(head == nullptr || head->next == nullptr) return head;
    BiNode *current = head;
    BiNode *newHead;
    while(current){
        BiNode *tmp = current->pre;
        current->pre = current->next;
        current->next = tmp;
        newHead = current;
        current = current->pre;
    }
    return newHead; //返回指向新链表的头节点
}
void printLinks(Node *head) {
    if (head == nullptr) {
        cout << "empty" << endl;
        return;
    }
    Node *tmp = head;
    while (tmp->next != nullptr) {
        cout << tmp->value << "->";
        tmp = tmp->next;
    }
    cout << tmp->value << endl;
}
void printBiLinks(BiNode *head){
    if(head == nullptr){cout<<"empty"<<endl;return;}
    BiNode *tmp = head;
    while(tmp->next != nullptr){
        cout<<"["<<tmp->value<<"]"<<"<->";
        tmp = tmp->next;
```

```
    }
    cout<<"["<<tmp->value<<"]"<<endl;
    cout<<endl;
}
class Links{
public:
    Node *head; // 头节点
    Links() : head(nullptr){}
    //创建链表
    Links(vector<int> v) {
        if(v.empty()){
            head = nullptr;
        }else {
            Node *first = new Node(v[0]);
            head = first;
            Node *tmp = first;
            for (int i = 1; i < v.size(); ++i) {
                Node *node = new Node(v[i]);
                tmp->next = node;
                tmp = node;
            }
        }
    }
    //打印链表
    void printLinks() {
        if (head == nullptr) {
            cout << "empty" << endl;
            return;
        }
        Node *tmp = head;
        while (tmp->next != nullptr) {
            cout << tmp->value << "->";
            tmp = tmp->next;
        }
        cout << tmp->value << endl;
    }
    //反转这个链表
    void inverseLinks(){
        Node *pre = nullptr;
        Node *next = nullptr;
        while(head!= nullptr){
            next = head->next;
            head->next = pre;
            pre = head;
            head = next;
        }
        head = pre;
    }
};

int main() {
    vector<int> v={1,2,3,4,5,6};
    Node *head = createLinks(v);
//    Node *CHead = createLinksCycle(v);
//    Node *IHead = notLinks(head);
    auto Binode = createBiNode(v);
//    auto *IBiHead = notBiLinks2(Binode.first);
//    auto BinodeC = createBiNodeCycle(v);
    printLinks(head);
    printBiLinks(Binode.first);
    return 0;
}
```

链表的相关应用

打印两个有序单链表的公共部分（值相同的部分）

例如：a : `1-->2-->3-->4-->5` b : `10->3-->4-->20-->30` 打印 3→4 思路： 因为是有序的链表，所以问题很简单。开始的时候l1 = head1;l2 = head2; 然后比较l1和l2指向的值的大小，谁小就向前移动谁，相等的话记录下来，最后一起打印出来。 如果是打印最长的公共部分，定义一个max变量，每次比较一下，最后打印那个最长的公共部分就行了。

删除单向链表中倒数第K个节点

例如：`head-->1-->2-->3-->4-->5-->6` K=2 返回：`head-->1-->2-->3-->4-->6`

思路一： 先遍历一遍链表，统计链表有多少个元素。然后就可以知道倒数第K个元素就是 正数第n-k+1个元素，从head开始移动指针，移动n-k次就是要删除的节点。 删除一个节点，只需要把原来指向这个元素的指针指向它后面的元素就可以了。所以我们要做的 操作就是从head开始，移动n-k-1次，找到要删除的节点前面的节点a,然后执行a->next = a->next->next

思路二： 首先遍历一边链表，每移动一次指针，就让K减1，如果遍历到链表的结尾K还是大于0的，说明没有这个节点 如果K为负，再从头遍历一遍链表，每次K+1，K为0的时候就找到的要删除的节点的前一个节点。

```

```

删除双向链表的倒数第K个节点

双向链表删除某个节点，需要知道该节点前的节点或者该节点后面的节点 设a为要删除的节点的前一个节点，则删除后面的节点的方式是：

```
a->next = a->next->next;
a->next->next->pre = a->next->pre;
```

设b为要删除的节点的后一个节点，则删除前面的节点的方式是：

```
b->pre = b->pre->pre;
b->pre->pre->next = b;
```

如果双向链表是有首尾头节点的，直接从尾头结点开始计数即可。 如果没有尾头结点，处理方式可以参考单链表删除倒数第K个节点的方法。

```

```

删除链表的中间节点

删除链表的a/b处的节点