

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN TP.HCM



Báo cáo bài tập 04

- Phạm Quang Sang
- MSSV: 24120429

15/04/2025

➤ Singly Linklist

```
struct NODE
{
    int key;
    NODE *p_next;
};

struct List
{
    NODE *p_head;
    NODE *p_tail;
    List(NODE *head = nullptr, NODE *tail = nullptr) : p_head(head), p_tail(tail) {}
};
```

-
- Create Node

```
NODE *createNode(int data)
{
    NODE*new_node = new NODE;
    new_node->key = data;
    new_node->p_next = nullptr;
    return new_node;
}
```

-
- Cách hoạt động
- Cấp phát cho con trỏ new_node kiểu NODE* bộ nhớ
- Gán thành viên key giá trị data được truyền vào
- Gán thành viên con trỏ p_next tới con trỏ null
- Trả về con trỏ new_node

- Create List

```
List *createList(NODE *p_node)
{
    List*L=new List(p_node,p_node);
    L->p_tail->p_next = nullptr;
    return L;
}
```

-
- Cách hoạt động
- Cấp phát cho con trỏ L kiểu List* bộ nhớ với hàm khởi tạo nhận 2 đối số đều là p_node gán cho p_head và p_tail
- L sẽ được tạo với Head và Tail đều trỏ tới p_node
- Gán p_next của p_node (lúc này cũng được trỏ bởi L->p_tail & L->p_head) bằng null để đảm bảo danh sách kết thúc đúng
- Trả về con trỏ L

- Add Head

```
bool addHead(List *&L, int data)
{
    NODE*n = createNode(data);
    n->p_next = L->p_head;
    if(L->p_head==nullptr){
        L->p_head = L->p_tail = n;
    }
    else
        L->p_head = n;
    return true;
}
```

-
- Cách hoạt động

- Khởi tạo con trỏ n kiểu NODE* bằng hàm createNode()
- Gán con trỏ p_next của n bằng L->p_head
- Nếu
 - L chưa có node nào thì gán đồng thời n cho p_tail và p_head
 - Ngược lại, trỏ p_head vào n
- Trả về true

○ Add Tail

```
bool addTail(List *&L, int data)
{
    NODE *new_node = createNode(data);
    if (L->p_head == L->p_tail){
        if(L->p_head==nullptr){
            L->p_head = L->p_tail = new_node;
        }
        else{
            L->p_tail = new_node;
            L->p_head->p_next = L->p_tail;
        }
    }
    else{
        L->p_tail->p_next = new_node;
        L->p_tail = new_node;
    }
    return true;
}
```

- Cách hoạt động
- Tạo node mới từ data.
- Nếu danh sách rỗng => gán cả head và tail bằng node mới.
- Nếu danh sách có đúng 1 node => gán node mới làm tail, cập nhật head->p_next.
- Nếu danh sách có từ 2 node trở lên => nối tail->p_next với node mới và cập nhật tail
- Trả về true

○ Remove Head

```
bool removeHead(List *&L)
{
    if(L->p_head==nullptr) return false;
    NODE*temp = L->p_head;
    if(L->p_head==L->p_tail){
        L->p_head = L->p_tail = nullptr;
    }
    else{
        L->p_head = L->p_head->p_next;
    }
    delete temp;
    return true;
}
```

- Nếu danh sách rỗng => trả về false.
- Nếu danh sách chỉ có 1 node => xóa node đó và đặt head, tail về null.
- Nếu có nhiều node => cập nhật head trỏ sang node kế tiếp.
- Giải phóng bộ nhớ node cũ và trả về true.

○ Remove Tail

```
void removeTail(List *&L)
{
    if(L->p_head==nullptr) return;
    NODE*temp = L->p_tail;
    if(L->p_head==L->p_tail){
        L->p_head = L->p_tail = nullptr;
    }
    else{
        NODE*prev = L->p_head;
        while(prev->p_next->p_next!=nullptr){
            prev=prev->p_next;
        }
        prev->p_next = nullptr;
        L->p_tail = prev;
    }
    delete temp;
    return;
}
```

-
- Nếu danh sách rỗng => thoát.
- Tạo node tạm temp lưu tail hiện tại
- Nếu chỉ có 1 node => đặt head, tail về null.
- Nếu có nhiều node => duyệt tới node kế cuối (prev), cập nhật tail = prev và prev->p_next = null.
- Giải phóng bộ nhớ node tail cũ (temp).

○ Remove All

```
void removeAll(List *&L)
{
    NODE *current = L->p_head;
    while (current)
    {
        NODE *next = current->p_next;
        delete current;
        current = next;
    }
    L->p_head=L->p_tail=nullptr;
    return;
}
```

-
- Tạo con trỏ tạm current trỏ vào head
- Duyệt từ đầu đến hết danh sách.
- Mỗi lần lặp: lưu next, xóa current, rồi chuyển sang next.
- Sau khi xóa hết → đặt head và tail về null.

○ Remove Before

```
void removeBefore(List *&L, int val)
{
    if(L->p_head==L->p_tail) return;
    NODE*cur = L->p_head->p_next;
    NODE*prev_cur = L->p_head;
    if(cur!=nullptr)
    while(cur->p_next!=nullptr){
        if(cur->p_next->key==val){
            prev_cur->p_next = cur->p_next;
            delete cur;
            return;
        }
        prev_cur = cur;
        cur = cur->p_next;
    }
    return;
}
```

▪

- Cách hoạt động
- Trường hợp danh sách rỗng hoặc chỉ có 1 node:
 - Thoát hàm bằng return
- Khởi tạo con trỏ:
 - cur = head->next
 - prev_cur = head (node đầu tiên)
- Duyệt danh sách từ node thứ 2 trở đi:
 - Lặp khi cur->p_next != null (đảm bảo có node đứng sau để so sánh)
 - Khi tìm thấy cur->p_next->key == val:
 - Xóa cur (tức node đứng trước node có giá trị val)
 - Nối prev_cur->p_next sang cur->p_next (bỏ qua cur)
 - Giải phóng bộ nhớ cur, rồi return.
- Nếu không tìm thấy:
 - Không xóa gì cả, kết thúc hàm.

○ Remove After

```
void removeAfter(List *&L, int val)
{
    if(L->p_head==L->p_tail) return;
    NODE*cur = L->p_head;
    while(cur!=nullptr){
        if(cur->key==val){
            NODE*temp = cur->p_next;
            if(temp!=nullptr){
                cur->p_next = temp->p_next;
                delete temp;
            }
            return;
        }
        cur = cur->p_next;
    }
    return;
}
```

- Nếu danh sách rỗng hoặc chỉ có 1 node => thoát.
- Duyệt từng node từ head:
 - Nếu cur->key == val:
 - Nếu node sau (cur->p_next) tồn tại:
 - Nối node hiện tại với node temp->next .
 - Xóa node temp
 - Thoát ngay khi xóa/đến cuối danh sách
 - Cur=cur->next: dịch con trỏ tới node tiếp theo
- Nếu không tồn tại node có key == val
 - Thoát hàm

○ Add Pos

```

bool addPos(List *&L, int data, int pos)
{
    if(pos<0||pos>countElements(L)){
        return false;
    }
    if(pos==0){
        addHead(L,data);
        return true;
    }
    if(pos==countElements(L)){
        addTail(L,data);
        return true;
    }
    else{
        NODE*prev_cur;
        NODE*cur = L->p_head;
        while(pos>=1){
            prev_cur = cur;
            cur = cur->p_next;
            pos--;
        }
        NODE*n=createNode(data);
        n->p_next = cur;
        prev_cur->p_next = n;
        return true;
    }
    return false;
}

```

-
- Cách hoạt động
- Dựa trên testcase: pos = 0 thêm vào đầu, pos = sizeofList thêm vào cuối
- Nếu pos không hợp lệ (âm hoặc vượt quá số phần tử) => trả false.
- Nếu pos == 0 => chèn đầu bằng addHead().
- Nếu pos == countElements(L) => chèn cuối bằng addTail.
- Ngược lại:
 - Duyệt đến vị trí cần chèn.
 - Dùng 2 con trỏ
 - Prev(quia vòng lặp ở dưới sẽ đi sau cur)
 - Cur = head
 - Vòng lặp giảm pos liên tục, khi về 0 thì cur trở tới vị trí cần chèn, prev là node ngay trước vị trí cần chèn, tức node mới sẽ nằm giữa cur và prev
 - Tạo node mới n, gán p_next trở đến node cur.
 - Gán prev_cur->p_next trở đến node mới.
 - Trả về true

○ Remove Pos

```

void removePos(List *&L, int data, int pos)
{
    if(pos<0||pos>countElements(L)){
        return;
    }
    if(pos==0){
        removeHead(L);
        return;
    }
    if(pos==countElements(L)){
        removeTail(L);
        return;
    }
    else{
        NODE*prev_cur;
        NODE*cur = L->p_head;
        while(pos>=1){
            prev_cur = cur;
            cur = cur->p_next;
            pos--;
        }
        if(cur->key!=data) return;
        prev_cur->p_next = cur->p_next;
        delete cur;
        return ;
    }
    return ;
}

```

-
- Nếu pos không hợp lệ(âm hoặc vượt quá số phần tử) => thoát.
- Nếu pos == 0 => gọi removeHead().
- Nếu pos == countElements(L) => gọi removeTail().
- Ngược lại:
 - Duyệt đến node tại vị trí pos.
 - Dùng 2 con trỏ
 - Prev(quả vòng lặp ở dưới sẽ đi sau cur)
 - Cur = head
 - Vòng lặp giảm pos liên tục, khi về 0 thì cur trở tới vị trí cần xóa, prev là node ngay trước vị trí cần xóa, cur chính là node cần xóa
 - Nếu giá trị node đó khác data => không xóa=>thoát.
 - Nếu giống => “cô lập node cur” và giải phóng bộ nhớ.
 - Thoát hàm.

○ Add Before

```

bool addBefore(List *&L, int data, int val)
{
    if(L->p_head==nullptr){
        return false;
    }
    if(L->p_head->key==val){
        addHead(L,data);
        return true;
    }
    NODE* prev = L->p_head;
    while(prev->p_next!=nullptr){
        if(prev->p_next->key==val){
            NODE*new_node=createNode(data);
            new_node->p_next=prev->p_next;
            prev->p_next = new_node;
            return true;
        }
        prev = prev->p_next;
    }
    return false;
}

```

- Nếu danh sách rỗng => trả false.
 - Nếu node đầu tiên có giá trị val => chèn đầu bằng addHead().
 - Tạo con trỏ NODE *prev gán bằng head
 - Duyệt danh sách(đến khi prev = tail):
 - Nếu prev->p_next->key == val.
 - tạo node mới, chèn vào trước node có val.
 - Trả về true
 - Cập nhật prev = prev->next;
 - Duyệt xong => không tìm thấy val => trả false.
- Add After

```

bool addAfter(List *&L, int data, int val)
{
    if(L->p_head==nullptr) return false;
    NODE* cur=L->p_head;
    while(cur!=nullptr){
        if(cur->key==val){
            if(cur==L->p_tail){
                addTail(L,data);
            }
            else{
                NODE*new_node = createNode(data);
                new_node->p_next = cur->p_next;
                cur->p_next = new_node;
            }
            return true;
        }
        cur=cur->p_next;
    }
    return false;
}

```

- Nếu danh sách rỗng => trả về false
- Duyệt danh sách từ head.
 - Khi gặp node có key == val:
 - Nếu nó là tail => gọi addTail()
 - Ngược lại => tạo node mới, chèn ngay sau node hiện tại.
 - Cập nhật cur = cur->next;
- Duyệt xong, không tìm thấy node có val => trả false.

○ Print List

```
void printList(List *L)
{
    for(NODE*i=L->p_head;i!=nullptr;i=i->p_next){
        cout<<i->key<<" ";
    }
    cout<<endl;
    return;
}
```

- Vòng for duyệt từng node từ head đến hết danh sách.
 - Mỗi lần lặp: in key của node ra màn hình, cách nhau bằng dấu cách.
- Kết thúc vòng lặp, in lệnh endl.

○ Count Element

```
int countElements(List *L)
{
    int count = 0;
    NODE*cur = L->p_head;
    while(cur!=nullptr){
        count++;
        cur = cur->p_next;
    }
    return count;
}
```

- Tạo biến đếm int count = 0, NODE*cur = head
- Vòng while đến khi cur = null
 - Tăng biến đếm count
 - Cập nhật cur = cur->next
- Trả về giá trị count

○ Reverse List

```
List *reverseList(List *L)
{
    if(L->p_head==nullptr){
        return new List();
    }
    NODE*i=L->p_head;
    while(i!=L->p_tail){
        NODE*temp = i->p_next;
        i->p_next = L->p_tail->p_next;
        L->p_tail->p_next = i;
        i=temp;
    }
    List*n_L = new List(L->p_tail,L->p_head);
    L->p_head=L->p_tail =nullptr;
    return n_L;
}
```

- Nếu danh sách rỗng => trả về con trỏ List được khởi tạo với head = tail = null
- Ngược lại

- Tạo con trỏ NODE* l = head lặp đến khi trỏ đến tail
 - //Nối l với phần sau tail, để kết thúc thì tail đứng đầu dãy, head cuối dãy
 - Con trỏ tạm temp lưu i->next
 - Nối l với phần sau tail: tail->next
 - Đặt lại phần sau tail: tail->next = l;
 - Gán l = temp (i->next trước đó)
- Khi kết thúc vòng lặp
 - Tail->...->head->null
 - Tạo con trỏ list* nL gán bằng List được khởi tạo với head = l->tail, tail = l->head;
 - Trả về con trỏ nL;

○ Remove Duplicate

```
void removeDuplicate(List *&L)
{
    if(countElements(L) ≤ 1) return;
    for(NODE*i = L->p_head; i ≠ L->p_tail; i=i->p_next){
        for(NODE*j=i->p_next, *prev_j=i; j≠nullptr;){
            if(j->key=i->key){
                prev_j->p_next=j->p_next;
                if(L->p_tail=j){
                    L->p_tail = prev_j;
                }
                delete j;
                j = prev_j->p_next;
            }
            else{
                prev_j = j;
                j=j->p_next;
            }
        }
        if(i=L->p_tail) break;
    }
    return;
}
```

- Nếu danh sách có 0 hoặc 1 phần tử thì thoát hàm
- Dùng 2 vòng lặp for
- Vòng ngoài, NODE* l chạy từ head đến trước tail
 - Vòng trong, NODE* j chạy từ i->next đến hết danh sách, kèm theo prev_j = l là con trỏ NODE* đi ngay sau j
 - Nếu j->key = i->key
 - Nối prev_j với j->next để “cô lập j”
 - Nếu j là tail: cập nhật tail = prev_j
 - Xóa node j, gán node j = prev_j->next(j->next trước đó);
 - Ngược lại
 - Gán prev_j = j, j = j->next;
 - Kết thúc 1 vòng for trong, kiểm tra xem l đã trỏ

thành tail hay chưa

- Nếu là tail, dừng vòng for ngoài ngay
- Ngược lại tiếp tục vòng for ngoài
 - $I = i \rightarrow \text{next};$

- Kết thúc.

- Remove Element

```
bool removeElement(List *&L, int key)
{
    if(L->p_head==nullptr) return false;
    bool noChange = true;
    for(NODE*cur=L->p_head,*prev=cur;cur!=nullptr;){
        if(cur->key==key){
            if(cur==L->p_head){
                L->p_head = L->p_head->p_next;
                if(cur==L->p_tail){//head = tail
                    L->p_tail = L->p_head;
                }
            }
            else{
                prev->p_next = cur->p_next;
                if(cur==L->p_tail){
                    L->p_tail = prev;
                }
            }
            NODE*temp = cur;
            cur = cur->p_next;
            delete temp;
            noChange = false;
        }
        else
            prev=cur,cur=cur->p_next;
    }
    if(noChange) return false;
    return true;
}
```

- Nếu danh sách rỗng => trả false.
- Tạo biến check thay đổi danh sách: noChange = true
- Khởi tạo con trỏ cur (node hiện tại) và prev (node trước đó).
- Duyệt từng node:
 - Nếu $\text{cur} \rightarrow \text{key} == \text{key}$:
 - Nếu là head:
 - Cập nhật head sang node kế tiếp.
 - Nếu head == tail (danh sách chỉ có 1 node) => cập nhật luôn tail.
 - Nếu không phải head:
 - Cập nhật $\text{prev} \rightarrow \text{p_next}$ để bỏ qua cur.
 - Nếu cur là tail => cập nhật tail = prev.
 - Xóa node cur và chuyển sang node tiếp theo.
 - Đặt noChange = false.
 - Nếu không khớp key => cập nhật prev và cur sang node kế tiếp

- Sau khi duyệt:
 - Nếu không có node nào bị xóa (noChange == true) => trả false.
 - Ngược lại => trả true.

➤ Doubly Linkedlist

```
typedef struct d_NODE{
    int key;
    d_NODE*pNext;
    d_NODE*pPrev;
}d_NODE;
typedef struct d_List{
    d_NODE*pHead;
    d_NODE*pTail;
    d_List(d_NODE *head = nullptr, d_NODE *tail = nullptr) : pHead(head), pTail(tail) {}
}d_List;
```

○

○ Create Node

```
d_NODE *createNode(int data)
{
    d_NODE *new_dnode = new d_NODE;
    new_dnode->pNext = new_dnode->pPrev = nullptr;
    new_dnode->key = data;
    return new_dnode;
}
```

-
- Tạo một node mới (d_NODE).
- Khởi tạo key của node bằng giá trị data.
- Thiết lập pNext và pPrev của node mới là nullptr (node không có liên kết trước và sau).
- Trả về con trỏ đến node mới.

○ Create List

```
d_List *createList(d_NODE *p_node)
{
    d_List *dL=new d_List;
    dL->pHead=dL->pTail=p_node;
    return dL;
}
```

-
- Tạo một danh sách đôi mới (d_List).
- Gán pHead và pTail của danh sách bằng con trỏ p_node được truyền vào (node duy nhất trong danh sách).
- Trả về con trỏ đến danh sách mới tạo.

○ Add Head

```

bool addHead(d_List *&L, int data)
{
    d_NODE*new_head = createNode(data);
    new_head->pNext = L->pHead;
    if(L->pHead == nullptr){
        L->pHead = L->pTail = new_head;
    }
    else{
        L->pHead = new_head;
        L->pHead->pPrev = new_head;
    }
    return true;
}

```

-
- Tạo một node mới với giá trị data bằng cách gọi createNode(data).
- Gắn node mới làm phần tử đầu của danh sách:
 - Nếu danh sách rỗng (L->pHead == nullptr), gán cả pHead và pTail bằng node mới.
 - Nếu danh sách không rỗng, gán node mới vào đầu danh sách và cập nhật liên kết pPrev của node đầu tiên.
- Trả về true để thông báo việc thêm thành công.

○ Add Tail

```

bool addTail(d_List *&L, int data) // this fu
{
    d_NODE *new_tail = createNode(data);
    new_tail->pPrev = L->pTail;
    if(L->pHead==nullptr){
        L->pTail = L->pHead = new_tail;
    }
    else{
        L->pTail->pNext = new_tail;
        L->pTail = new_tail;
    }
    return true;
}

```

-
- Tạo một node mới với giá trị data bằng cách gọi createNode(data).
- Nối prev node mới bằng tail list.
 - Nếu danh sách rỗng (L->pHead == nullptr), gán cả pHead và pTail bằng node mới.
 - Nếu danh sách không rỗng
 - Nối next tail list với node mới
 - Trở tail vào node mới
- Trả về true để thông báo việc thêm thành công.

○ Remove Head

```

bool removeHead(d_List *L)
{
    d_NODE*temp = L->pHead;
    if(L->pHead==L->pTail){
        if(L->pHead==nullptr){
            return false;
        }else{
            delete temp;
            L->pHead = L->pTail = nullptr;
        }
    }else{
        L->pHead = L->pHead->pNext;
        L->pHead->pPrev = nullptr;
        delete temp;
    }
    return true;
}

```

- Kiểm tra xem danh sách có một phần tử duy nhất (L->pHead == L->pTail):
 - Nếu danh sách rỗng (L->pHead == nullptr), trả về false.
 - Nếu danh sách chỉ có một phần tử, xóa phần tử đó và cập nhật cả pHead và pTail thành nullptr.
- Nếu danh sách có nhiều phần tử
 - gán pHead là phần tử tiếp theo của node đầu và cập nhật liên kết pPrev của phần tử mới đầu thành nullptr.
 - Xóa node đầu trước đó (temp).
- Trả về true để thông báo việc xóa thành công.

○ Remove Tail

```

void removeTail(d_List *L)
{
    d_NODE*temp = L->pTail;
    if(L->pHead == L->pTail){
        if(L->pHead == nullptr){
            return;
        }
        else{
            L->pHead = L->pTail = nullptr;
            delete temp;
        }
    }
    else{
        L->pTail = L->pTail->pPrev;
        L->pTail->pNext = nullptr;
        delete temp;
    }
    return;
}

```

- Cách hoạt động
- Kiểm tra xem danh sách có một phần tử duy nhất (L->pHead == L->pTail):
 - Nếu danh sách rỗng (L->pHead == nullptr), không làm gì và trả về.
 - Nếu danh sách chỉ có một phần tử, xóa phần tử đó và cập nhật cả pHead và pTail thành nullptr.
- Nếu danh sách có nhiều phần tử

- gán pTail là phần tử trước đó của node cuối và cập nhật liên kết pNext của phần tử mới cuối thành nullptr.
- Xóa node cuối (temp).
- Thoát hàm
- Remove All

```
void removeAll(d_List *&L)
{
    d_NODE *current = L->pHead;
    while (current)
    {
        d_NODE *next = current->pNext;
        delete current;
        current = next;
    }
    L->pHead = L->pTail = nullptr;
    return;
}
```

- Cách hoạt động
- Khởi tạo con trỏ current là pHead của danh sách.
- Duyệt qua tất cả các node trong danh sách:
 - Lưu trữ con trỏ tiếp theo (next) của node hiện tại.
 - Xóa node hiện tại.
 - Cập nhật con trỏ current thành node tiếp theo.
- Sau khi xóa tất cả các node, cập nhật cả pHead và pTail của danh sách thành nullptr.
- Remove Before

```
void removeBefore(d_List *&L, int val)
{
    if(L->pHead == L->pTail) return;
    d_NODE*cur = L->pHead;
    while(cur!=nullptr){
        if(cur->key==val&&cur!=L->pHead){
            break;
        }
        cur = cur->pNext;
    }
    if(cur==nullptr) return;
    if(cur->pPrev==L->pHead){
        removeHead(L);
        return;
    }
    d_NODE*temp = cur->pPrev;
    cur->pPrev->pNext = cur;
    cur->pPrev = temp->pPrev;
    delete temp;
    return;
}
```

- Cách hoạt động
- Kiểm tra xem danh sách có chỉ một phần tử hay không (L->pHead == L->pTail), nếu có thì không thực hiện gì và kết thúc.
- Khởi tạo con trỏ cur trỏ đến pHead của danh sách.
- Duyệt qua danh sách để tìm phần tử có giá trị key == val, đồng thời đảm bảo phần tử tìm được không phải là pHead.

- Nếu không tìm thấy phần tử, thoát hàm.
- Nếu phần tử cần xóa nằm ngay trước pHead, gọi hàm removeHead để xóa phần tử đầu tiên.
- Nếu không phải pHead, tìm phần tử trước phần tử hiện tại (cur->pPrev), cập nhật liên kết của phần tử trước đó (cur->pPrev->pNext) để bỏ qua node cần xóa và liên kết lại.
- Xóa node temp (phần tử cần xóa).

○ Remove After

```
void removeAfter(d_List *L, int val)
{
    if(L->pHead==L->pTail) return;
    d_NODE*cur=L->pHead;
    while(cur!=L->pTail){
        if(cur->key==val){
            break;
        }
        cur=cur->pNext;
    }
    if(cur==L->pTail) return;
    d_NODE* temp = cur->pNext;
    if(temp==L->pTail){
        removeTail(L);
        return;
    }
    cur->pNext = temp->pNext;
    temp->pNext->pPrev = cur;
    delete temp;
    return;
}
```

- Cách hoạt động
- Kiểm tra xem danh sách có chỉ một phần tử hay không (L->pHead == L->pTail), nếu có thì không thực hiện gì và kết thúc.
- Khởi tạo con trỏ cur trỏ đến pHead của danh sách.
- Duyệt qua danh sách để tìm phần tử có giá trị key == val, và đảm bảo phần tử hiện tại không phải là pTail.
- Nếu không tìm thấy phần tử hoặc phần tử là pTail, thoát hàm.
- Tạo con trỏ temp trỏ đến phần tử sau cur (phần tử cần xóa).
- Nếu phần tử cần xóa là pTail, gọi hàm removeTail để xóa phần tử cuối.
- Cập nhật liên kết để bỏ qua phần tử temp trong danh sách: cur->pNext trỏ đến phần tử sau temp, và cập nhật lại liên kết pPrev cho phần tử sau temp.
- Xóa node temp (phần tử cần xóa).

○ Add Pos


```

bool addPos(d_List *L, int data, int pos)
{
    int sizeOfList = countElements(L);
    if(pos<0||pos>sizeOfList){
        return false;
    }
    if(pos==0){
        addHead(L,data);
        return true;
    }
    if(pos==sizeOfList){
        addTail(L,data);
        return true;
    }
    d_NODE *new_node = createNode(data);
    new_node->pNext = L->pHead;
    while(pos){
        new_node->pPrev = new_node->pNext;
        new_node->pNext = new_node->pNext->pNext;
        pos--;
    }
    new_node->pPrev->pNext = new_node;
    new_node->pNext->pPrev = new_node;
    return true;
}

```

-
- Cách hoạt động
- Kiểm tra tính hợp lệ của pos: Nếu pos không hợp lệ (nhỏ hơn 0 hoặc lớn hơn kích thước danh sách), trả về false.
- Chèn vào đầu danh sách (pos == 0): Gọi addHead(L, data) để thêm phần tử vào đầu.
- Chèn vào cuối danh sách (pos == sizeOfList): Gọi addTail(L, data) để thêm phần tử vào cuối.
- Chèn vào giữa danh sách:
- Tạo node mới với giá trị data.
- Duyệt qua danh sách từ đầu đến vị trí pos.
- Cập nhật liên kết giữa các node để chèn node mới vào đúng vị trí:
- Cập nhật con trỏ pPrev và pNext của node trước và node sau.
- Trả về true, thông báo thành công

○ Remove Pos

```

void removePos(d_List *L, int data, int pos)
{
    int sizeOfList = countElements(L);
    if(pos<0||pos>sizeOfList){
        return;
    }
    if(pos==0&&L->pHead->key==data){
        removeHead(L);
        return;
    }
    if(pos==sizeOfList&&L->pTail->key==data){
        removeTail(L);
        return;
    }
    d_NODE*cur = L->pHead;
    while(pos){
        cur = cur->pNext;
        pos--;
    }
    if(cur->key==data) return;
    cur->pPrev->pNext = cur->pNext;
    delete cur;
    return;
}

```

- Cách hoạt động
- Kiểm tra tính hợp lệ của pos: Nếu pos không hợp lệ (nhỏ hơn 0 hoặc lớn hơn kích thước danh sách), trả về.
- Xử lý khi xóa phần tử đầu danh sách (pos == 0): Nếu phần tử đầu có giá trị data, gọi removeHead(L).
- Xử lý khi xóa phần tử cuối danh sách (pos == sizeofList): Nếu phần tử cuối có giá trị data, gọi removeTail(L).
- Duyệt đến vị trí pos trong danh sách.
- Kiểm tra giá trị data tại node hiện tại:
- Nếu phần tử tại vị trí đó có giá trị data, cập nhật liên kết:
- Cập nhật liên kết giữa node trước và node sau để loại bỏ node tại pos.
- Xóa node.

○ Add Before

```
bool addBefore(d_List *L, int data, int val)
{
    d_NODE*cur=L->pHead;
    while(cur!=nullptr&&cur->key!=val){
        cur=cur->pNext;
    }
    if(cur==nullptr) return false;
    if(cur==L->pHead){
        addHead(L,data);
        return true;
    }
    else{
        d_NODE*new_node = createNode(data);
        cur->pPrev->pNext = new_node;
        new_node->pNext = cur;
        return true;
    }
    return false;
}
```

- Cách hoạt động
- Duyệt qua danh sách: Tìm phần tử có giá trị val.
- Kiểm tra: Nếu không tìm thấy phần tử có giá trị val, trả về false.
- Thêm vào đầu danh sách (cur == L->pHead):
 - Gọi addHead(L, data) để thêm phần tử vào đầu danh sách.
 - Trả về true
- Thêm vào trước phần tử khác:
 - Tạo node mới với giá trị data.
 - Cập nhật liên kết của node trước node cur và node cur để chèn node mới vào đúng vị trí.
 - Trả về true
- Trả về false

○ Add After

```

bool addAfter(d_List *L, int data, int val)
{
    d_NODE*cur = L->pHead;
    while(cur!=nullptr&&cur->key!=val){
        cur=cur->pNext;
    }
    if(cur==nullptr) return false;
    if(cur==L->pTail){
        addTail(L,data);
        return true;
    }
    else{
        d_NODE*new_node = createNode(data);
        new_node->pPrev = cur;
        new_node->pNext = cur->pNext;
        cur->pNext = new_node;
        return true;
    }
    return false;
}

```

-
- Cách hoạt động
- Duyệt qua danh sách: Tìm phần tử có giá trị val.
- Kiểm tra: Nếu không tìm thấy phần tử có giá trị val, trả về false.
- Thêm vào cuối danh sách (cur == L->pTail)
 - Gọi addTail(L, data) để thêm phần tử vào cuối danh sách.
 - Trả về true
- Thêm vào sau phần tử khác:
 - Tạo node mới với giá trị data.
 - Cập nhật liên kết của node cur và node sau cur để chèn node mới vào đúng vị trí.
 - Trả về true
- Trả về false

○ Print List

```

void printList(d_List *L)
{
    d_NODE*cur = L->pHead;
    while(cur!=nullptr){
        std::cout<<cur->key<<' ';
        cur=cur->pNext;
    }
    std::cout<<'\n';
    return;
}

```

-
- Cách hoạt động
- Duyệt qua danh sách: Bắt đầu từ pHead và in ra giá trị của mỗi phần tử.
- Kết thúc khi hết danh sách.
- Cuối cùng In ra \n để xuống dòng

○ Count element

```

int countElements(d_List *L)
{
    d_NODE*cur = L->pHead;
    int count = 0;
    while(cur!=nullptr){
        count++;
        cur=cur->pNext;
    }
    return count;
}

```

-
- Cách hoạt động
- Duyệt qua danh sách: Bắt đầu từ pHead,
 - Đếm số phần tử trong danh sách.
 - Dịch con trỏ cur
- Trả về số lượng phần tử trong danh sách.

○ Reverse List

```

d_List *reverseList(d_List *L)
{
    d_NODE*i = L->pHead;
    while(i!=L->pTail){
        d_NODE*temp = i->pNext;
        i->pNext = L->pTail->pNext;
        if(L->pTail->pNext!=nullptr)
            L->pTail->pNext->pPrev = i;
        L->pTail->pNext = i;
        i = temp;
    }
    std::swap(L->pHead,L->pTail);
    d_List* rL = new d_List(L->pHead,L->pTail);
    L->pHead = L->pTail = nullptr;
    return rL;
}

```

-
- Cách hoạt động
- Khởi tạo con trỏ i: Gán i là pHead của danh sách.
- Duyệt qua danh sách: Lặp qua từng phần tử trong danh sách cho đến khi gặp pTail.
 - *Đảo ngược liên kết:
 - Tạm lưu pNext của node hiện tại vào temp.
 - Chỉnh sửa liên kết pNext của node hiện tại để trỏ đến pTail->pNext.
 - Nếu pTail->pNext không phải nullptr, cập nhật lại liên kết pPrev của node tiếp theo.
 - Cập nhật lại pTail->pNext trỏ đến node hiện tại.
- Hoán đổi pHead và pTail: Sau khi đảo ngược tất cả các liên kết.
- Tạo danh sách mới rL: Khởi tạo danh sách mới với pHead là L->pHead và pTail là L->pTail(hàm khởi tạo).
- Gán lại pHead và pTail của L là nullptr.
- Trả về danh sách đảo ngược (rL).

○ Remove duplicate

```
void removeDuplicate(d_List *&L)
{
    if(countElements(L) <= 1) return;
    for(d_NODE*i=L->pHead; i!=L->pTail; i=i->pNext){
        for(d_NODE*j=i->pNext; j!=nullptr;){
            if(j->key==i->key){
                d_NODE*temp = j->pNext;
                j->pPrev->pNext = j->pNext;
                if(j->pNext!=nullptr)
                    j->pNext->pPrev = j->pPrev;
                if(j==L->pTail){
                    L->pTail=j->pPrev;
                }
                delete j;
                j = temp;
            }
            else{
                j = j->pNext;
            }
        }
        if(i==L->pTail) break;
    }
    return;
}
```

-
- Cách hoạt động
- Nếu danh sách có 0 hoặc 1 phần tử ($\text{countElements}(L) \leq 1$), thoát hàm ngay.
- Dùng 2 vòng lặp for:
 - Vòng ngoài: $d_NODE^* i$ chạy từ $pHead$ đến trước $pTail$.
 - Vòng trong: $d_NODE^* j$ chạy từ $i \rightarrow pNext$ đến hết danh sách.
 - Nếu $j \rightarrow key == i \rightarrow key$:
 - Cập nhật liên kết: Nối $i \rightarrow pPrev$ với $j \rightarrow pNext$, nếu $j \rightarrow pNext$ không phải $nullptr$, cập nhật $j \rightarrow pNext \rightarrow pPrev$.
 - Cập nhật $pTail$: Nếu j là $pTail$, cập nhật $pTail$ bằng $j \rightarrow pPrev$.
 - Xóa phần tử j và gán lại $j = j \rightarrow pNext$ (phần tử sau j).
 - Tiếp tục vòng lặp ngoài:
 - Sau mỗi vòng lặp trong, kiểm tra xem i có phải là $pTail$ chưa.
 - Nếu là $pTail$, thoát vòng lặp ngoài.
 - Nếu không, tiếp tục vòng lặp ngoài ($i = i \rightarrow pNext$).
 - Kết thúc hàm: hoàn thành yêu cầu!!!!

○ Remove Element

```


bool removeElement(d_List *&L, int key)
{
    if(L->pHead==nullptr) return false;
    bool noChange = true;
    for(d_NODE*i=L->pHead;i!=nullptr;){
        if(i->key==key){
            if(i==L->pHead){
                L->pHead = i->pNext;
                if(L->pHead!=nullptr)
                    L->pHead->pPrev=i->pPrev;
                if(i ==L->pTail){
                    L->pTail = L->pHead;
                }
            }
            else{
                i->pPrev->pNext = i->pNext;
                if(i==L->pTail){
                    L->pTail = i->pPrev;
                }
                else{
                    i->pNext->pPrev = i->pPrev;
                }
            }
            d_NODE*temp = i;
            i = i->pNext;
            delete temp;
            noChange = false;
        }
        else
            i = i->pNext;
    }
    if(noChange)
        return false;
    return true;
}



```




-
- Cách hoạt động
- Kiểm tra danh sách rỗng: Nếu $L \rightarrow pHead == nullptr$, thoát hàm và trả về false.
- Dùng vòng lặp for, $d_NODE^* i = L \rightarrow pHead$ để duyệt qua tất cả các phần tử trong danh sách.
- Nếu $i \rightarrow key == key$, thực hiện các bước sau:
 - Nếu i là $pHead$:
 - Cập nhật $L \rightarrow pHead$ sang $i \rightarrow pNext$.
 - Nếu $L \rightarrow pHead \neq nullptr$, cập nhật $L \rightarrow pHead \rightarrow pPrev = nullptr$.
 - Nếu $i == L \rightarrow pTail$, cập nhật $L \rightarrow pTail = L \rightarrow pHead$.
 - Nếu i không phải $pHead$:
 - Nối $i \rightarrow pPrev \rightarrow pNext$ với $i \rightarrow pNext$.
 - Nếu $i == L \rightarrow pTail$, cập nhật $L \rightarrow pTail = i \rightarrow pPrev$.
 - Nếu không, cập nhật $i \rightarrow pNext \rightarrow pPrev = i \rightarrow pPrev$.
- Xóa phần tử i :
- Giữ lại i vào $temp$, sau đó cập nhật $i = i \rightarrow pNext$ và xóa $temp$.
- Đánh dấu $noChange = false$ để cho biết có thay đổi trong danh sách.
- Nếu i không phải key , dịch i sang phải
- Kết thúc vòng lặp:
- Kiểm tra thay đổi:
 - Nếu không có thay đổi ($noChange == true$), trả về false.

- Nếu có thay đổi, trả về true.

➤ GIT

TH_DSA_24120429 / week4 / 

 **coangsang** complete a65e62d · 3 minutes ago  **History**

Name	Last commit message	Last commit date
 ..		
 DoublyLinkedList.cpp	complete	3 minutes ago
 LinkedList.cpp	complete	3 minutes ago

○