VÕ TIẾN
Thảo luận kiến thức CNTT trường BK về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

Cấu Trúc Dữ Liệu Và Giải Thuật (DSA)

DSA0 - HK241

Lý Thuyết

Thảo luận kiến thức CNTT trường BK
về KHMT(CScience), KTMT(CEngineering)
https://www.facebook.com/groups/khmt.ktmt.cse.bku

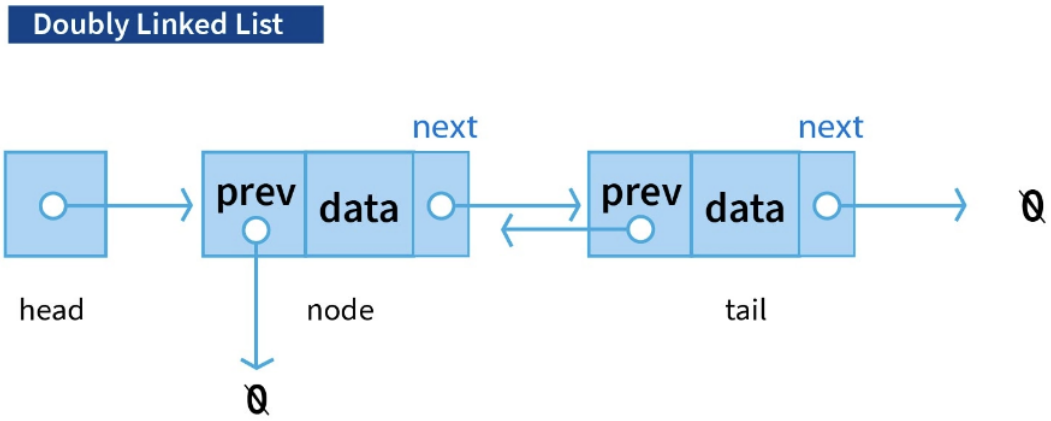
TP. HỒ CHÍ MINH, THÁNG 6/2024

Mục lục

1 Định nghĩa và so sánh với mảng

Linked List: Danh sách liên kết (Linked List)

Danh sách liên kết (Linklist) là một cấu trúc dữ liệu tuyến tính (Linear) trong đó các phần tử (Node) được liên kết với nhau bằng các liên kết. Mỗi phần tử có hai phần: dữ liệu (Data) và liên kết đến phần tử tiếp theo (Next) nhưng đối với một số danh sách liên kết đặt biệt có thể có thêm (Prev) để liên kết với phần tử trước đó. Phần tử đầu tiên của danh sách được gọi là node đầu (Head) và phần tử cuối được gọi là node cuối (Tail).



Hình 1: *Một đại diện Linked List (Singly Linked List)*

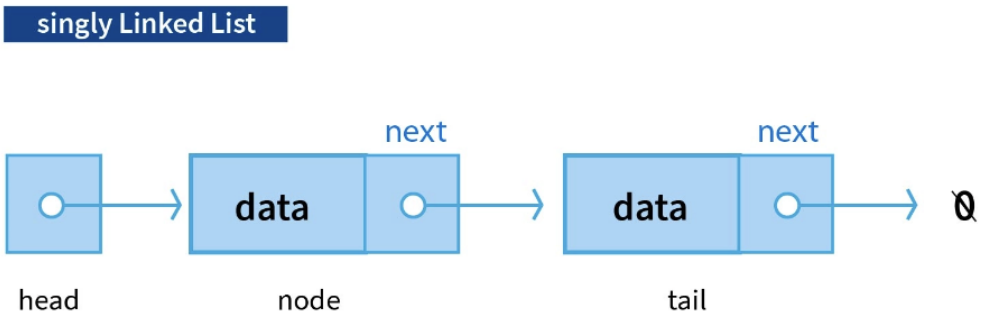
Danh sách liên kết có thể được sử dụng để lưu trữ các dữ liệu khác nhau, chẳng hạn như số, chuỗi và đối tượng, size. Chúng thường được sử dụng khi cần thêm hoặc xóa các phần tử trong danh sách một cách nhanh chóng và dễ dàng.

Tính năng	Danh sách liên kết	Mảng
Mức độ liên kết	Liên kết	Không liên kết
Kích thước	Độ dài có thể thay đổi	Độ dài cố định
Truy cập ngẫu nhiên	Không thể	Có thể
Thêm và xóa	Nhanh chóng và dễ dàng	Chậm
Bộ nhớ	Tiêu tốn nhiều bộ nhớ	Tiêu tốn ít bộ nhớ

Bảng 1: *So sánh Danh sách liên kết và Mảng*

2 Singly Linked List

Một danh sách liên kết đơn (Singly Linked List) là một cấu trúc dữ liệu tuyến tính trong đó mỗi phần tử chứa một giá trị và một liên kết đến phần tử tiếp theo.



Hình 2: *Singly Linked List*

Giải Thích: Danh sách liên kết đơn (Singly Linked List)

- *Head* lưu trữ vị trí đầu tiên của danh sách nếu trường hợp danh sách rỗng thì $Head = nullptr$
- *Tail* lưu trữ vị trí cuối cùng của node nếu trường hợp chỉ có một node thì $Tail = Head$
- *node* là nơi lưu dữ liệu và vị trí của *node* tiếp theo
- *data* là dữ liệu đang được lưu
- *next* là địa chỉ của *node* kế tiếp nếu là *node* cuối cùng thì $next = nullptr$

2.1 Implementation Singly Linked List in C++

```
class Singly_Linked_List
{
public:
    class Node;
protected:
    Node* head, *tail;  //! Node đầu tiên và Node cuối cùng
    int size;           //! kích thước của 1
public:
    //! khởi tạo
    Singly_Linked_List();
    Singly_Linked_List(const Singly_Linked_List& others);
    ~Singly_Linked_List();
    int Count();
    bool Empty();
    //! các hàm chỉnh sửa dữ liệu
    void Insertion(int data, int index);
    void Deletion(int index);

    //! cách hàm lấy dữ liệu
    int Get(int index);
    int IndexOf(int data);
    bool Contains ( int data );

```

```
24      //! các hàm sort tăng dần nha
25      void SelecetionSort();
26      //void MergerSort(); //~ hàm này qua lý thuyết sort sẽ có
27
28      //! các hàm hay dùng
29      Singly_Linked_List Concat(const Singly_Linked_List& others);
30      Singly_Linked_List SubSingly_Linked_List(int from, int to);
31      void Reverse();
32
33      //! hàm dislay cấu trúc [Size=2,DataHead=1,DataTail=2,DataNode : 1->2->nullptr]
34      string Dislay();
35
36      //! các hàm làm thêm bài tập
37  public:
38      class Node{
39      public:
40          int data;   //! dữ liệu
41          Node* next; //! trỏ tới node tiếp theo
42      public:
43          Node(int data = 0, Node* next = nullptr)
44          :data(data), next(next){}
45      };
46  };
```

Giải Thích: Danh sách liên kết đơn (Singly Linked List)

- *Singly_Linked_List* lớp dùng để quản lí danh sách liên kết dễ dàng hơn
- *Node* lớp này dùng để lưu dữ liệu (*data*) và vị trí của *Node* tiếp theo (*next*), Tùy vào bài toán mà có thể thêm thuộc tính (*Attribute*) và hàm (*Method*) khác để hỗ trợ cho quá trình xử lí.
- *Node* lớp nếu nằm bên trong (*Nested class*) hay bên ngoài đều được đối với lớp *Singly_Linked_List*, nếu bên trong thì chúng ta có thể kiểm sót được *class Node* có tính đóng gói (*Encapsulation*) bảo mật hơn so với bên ngoài và cũng mang tính trừ tượng (*Abstract*) hơn
- *head*, *tail* dùng để quản lí *node* đầu và cuối của danh sách liên kết đơn phần *tail* có thể không cần, *size* thì lưu trữ số lượng *Node* vì khi có *size* giảm độ phức tạp khi tìm kích thước của danh sách liên kết đi còn $O(1)$, có thể thêm 1 số thuộc tính (*Attribute*) ví dụ như các node lưu trữ tiền ta có thể thêm một thuộc tính *total* để lưu tổng số tiền vì tổng số tiền thường được dùng nên giảm đi độ phức tạp khi tìm tổng số tiền từ $O(n) -> O(1)$
- Hàm *Contructor* khởi tạo hoặc sao chép (*copy*) từ một danh sách khác độ phức tạp nếu *copy*, độ phức tạp $O(n)$. Hàm *Destructor* dùng giải phóng bộ nhớ còn lại, độ phức tạp $O(n)$
- Nhóm hàm chỉnh sửa gồm *insert* và *delete* thêm hoặc xóa tại một vị trí bất kì, độ phức tạp $O(n)$
- Nhóm hàm lấy giữ liệu gồm *count*(đếm), *empty*(rỗng), *get*(lấy), *indexof*(lấy chỉ số), *contains*(chứa), hàm *count*(đếm), *empty*(rỗng) độ phức tạp $O(1)$ vì có biến *size* còn lại độ phức tạp $O(n)$
- nhóm in ra gồm *dislay* độ phức tạp $O(n)$
- nhóm hàm khác *concat*(nối), *sub*(cắt), *reverse*(đảo ngược), *sort*(sắp xếp)

2.2 Contructor(Khởi tạo)

```cpp
//* tùy vào tham số truyền vào mà ta có thể có nhiều contructor
//* Thông thường thì chỉ cần rỗng và tham số truyền vào chính nó
//^ độ phức tạp O(1)
Singly_Linked_List::Singly_Linked_List():head(nullptr), tail(nullptr), size(0){

};

//* so chép tất cả các dữ liệu other sang this
//* luôn khởi tạo head tail và size trước khi xử lí
//^ độ phức tạp O(n)
Singly_Linked_List::Singly_Linked_List(const Singly_Linked_List& others)
:head(nullptr), tail(nullptr), size(0){
    Node* temp = others.head;
    for(int i = 0; i < others.size; i++)
    {
        this->Insertion(temp->data, i); //! chèn tại cuối
        temp = temp->next;  //! dịch ptr temp
    }
}
```

Hình 3: *Contructor Singly Linked List*

Giải Thích: Hàm Khởi tạo Danh sách liên kết đơn

- Hàm không tham số đầu vào thì khởi tạo *head*, *tail*, *size* các giá trị mặt định là *nullptr*, *nullptr*, 0, độ phức tạp $O(1)$
- Hàm có tham số đầu vào hoặc xử lý 1 gì đó bang đầu như khởi tạo *static* ta phải thực hiện, trong phần này truyền vào là 1 danh sách liên kết nên cần copy tất cả các node ra danh sách liên kết cần khởi tạo độ phức tạp sẽ là $O(n)$ vì hàm Insertion chèn ở cuối độ phức tạp $O(1)$
- Cách duyệt danh sách liên kết có thể dùng *for* với *size* hoặc là *while* với *head* tới giá trị *Nullptr* thường thì dùng *size* để quản lí và tránh các lỗi không mong muốn như *Dangling Pointers*, *Dereferencing NULL Pointer*
- $temp = temp->next$ dịch con trỏ đến *node* tiếp theo.

2.3 Destructor(Hàm hủy)

Hàm hủy(được gọi khi thu hồi) vì trong này vẫn còn nhiều *Node* đang sử dụng nên cần phải xóa để không bị leak memory xóa bằng cách duyệt qua tất cả các *Node* từ *head* đến *tail*, khi thu hồi xong cần cập nhật lại *head*, *tail*, *size*. Hàm này có thể giống hàm *clear*, độ phức tạp $O(N)$.

```
//* xóa tất cả các phần tử Node đang còn trong Singly_Linked_List
//* hàm này tương đương với clear nhưng chỉ gọi ghi kết thúc phạm vi
//* có thể dùng hàm remove để xử lý
//^ độ phức tạp O(n)
Singly_Linked_List::~Singly_Linked_List(){
    Node* temp; //~ node mẫu
    //! có thể dùng hàm Deletion như phần contructor phía trên
    while(size != 0){
        //! dịch ptr head trước khi xóa không sẽ bị lỗi Dangling pointer
        temp = head;
        head = head->next;
        delete temp;
        size --;
    }
    //! xóa hết rồi gán lại ban đầu thôi có thể có hay không thì cũng được
    //! nhưng hàm clear thì phải có
    head = tail = nullptr;
    size = 0;
}
```

Hình 4: *Destructor Singly Linked List*

Giải Thích: Hàm hủy tạo Danh sách liên kết đơn

- Dùng *while* để hoặc *for* để xóa đi số lượng *node* là *size* bằng lệnh *delete* lúc này nó sẽ gọi hàm hủy của *node* nếu có nên độ phức tạp là $O(n)$.
- Mỗi lần xóa sẽ xóa từ đầu đến cuối
- có thể gọi hàm *remove* để xóa

2.4 Insertion

```cpp
//* xóa tất cả các phần tử Node đang còn trong Singly_Linked_List
//* hàm này tương đương với clear nhưng chỉ gọi ghi kết thúc phạm vi
//* có thể dùng hàm remove để xử lý
//^ độ phức tạp O(n)
Singly_Linked_List::~Singly_Linked_List(){
    Node* temp; //~ node mẫu
    //! có thể dùng hàm Deletion như phần contructor phía trên
    while(size != 0){
        //! dịch ptr head trước khi xóa không sẽ bị lỗi Dangling pointer
        temp = head;
        head = head->next;
        delete temp;
        size --;
    }
    //! xóa hết rồi gán lại ban đầu thôi có thể có hay không thì cũng được
    //! nhưng hàm clear thì phải có
    head = tail = nullptr;
    size = 0;
}
```

Hình 5: *Insertion Singly Linked List*

Giải Thích: Hàm thêm tạo Danh sách liên kết đơn

- $index < 0 \quad \&\& \quad index > size$ trường hợp ngoại phạm vi nên nén ra lỗi, độ phức tạp $O(1)$.
- $size = 0$ Danh sách đang rỗng $(empty)$, độ phức tạp $O(1)$.
- $index = 0$ Chèn đầu Danh sách, độ phức tạp $O(1)$.
- $index = size$ Chèn cuối Danh sách, độ phức tạp $O(1)$ nếu có $tail$ nếu không có $tail$ thì sẽ là $O(N)$.
- $else$ trường hợp còn lại nằm giữa danh sách, độ phức tạp $O(N)$.

2.4.1 Index nằm ngoại phạm vi

Vì giá trị $index$ khi truyền vào hàm $insert$ có thể nằm ngoài giá trị cho phép của $index >= 0$ and $index <= size$ ra sẽ nén($thow$) ra lỗi.

```cpp
//! trường hợp index nằm ngoại phạm vi
if(index < 0 || index > this->size) out_of_range("Index of Linked List is invalid!");

Node* newNode = new Node(data, nullptr);
```

Hình 6: *Insertion với index nằm ngoại phạm vi*

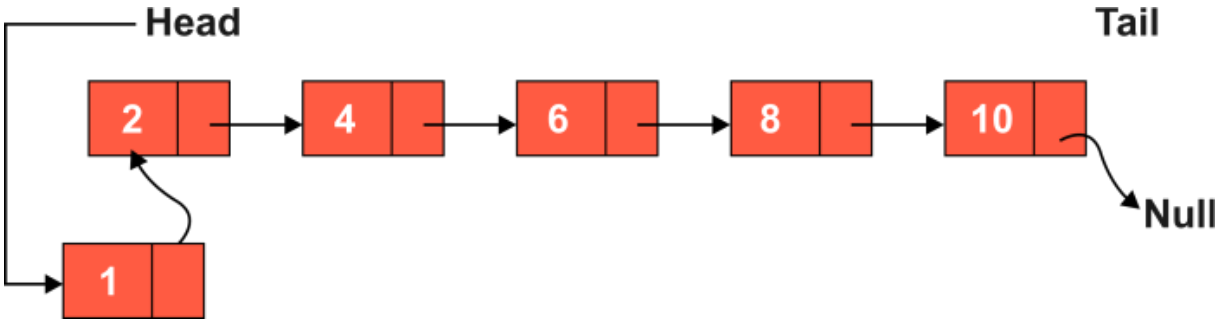
2.4.2 Danh sách đang rỗng(empty)

Khi danh sách đang rỗng lúc này *head*, *tail* đều mang giá trị *Nullptr* cần cập nhật giá trị cả 2 bằng *newNode* node cần chèn, tăng *size* lên 1.

```
//! Singly_Linked_List đang rỗng (empty)
//^ độ phức tạp O(1)
if(this->size == 0){
    head = tail = newNode;
}
```

Hình 7: *danh sách rỗng (empty)*

2.4.3 Chèn đầu Danh sách



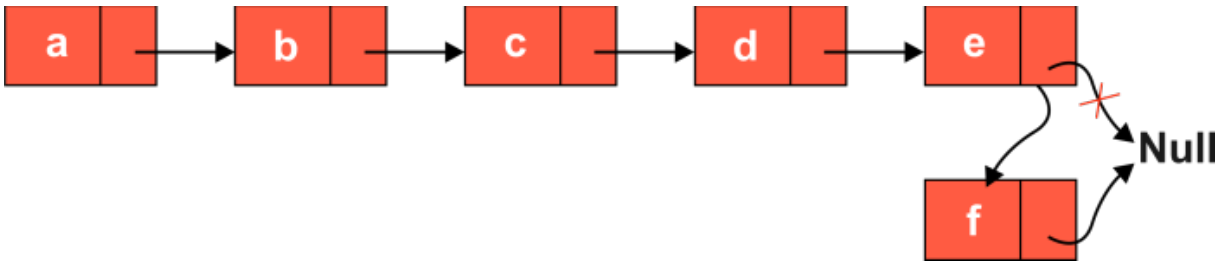
Hình 8: *Ý tưởng Chèn ở đầu danh sách*

Chèn đầu Danh sách ta phải cập nhật *head* bằng giá trị *newNode* và *newNode* trỏ tới *head* cũ

```
//! Insert ở đầu danh sách (head)
//^ độ phức tạp O(1)
else if(index == 0){
    newNode->next = head;
    head = newNode;
}
```

Hình 9: *Chèn ở đầu danh sách*

2.4.4 Chèn Cuối Danh sách



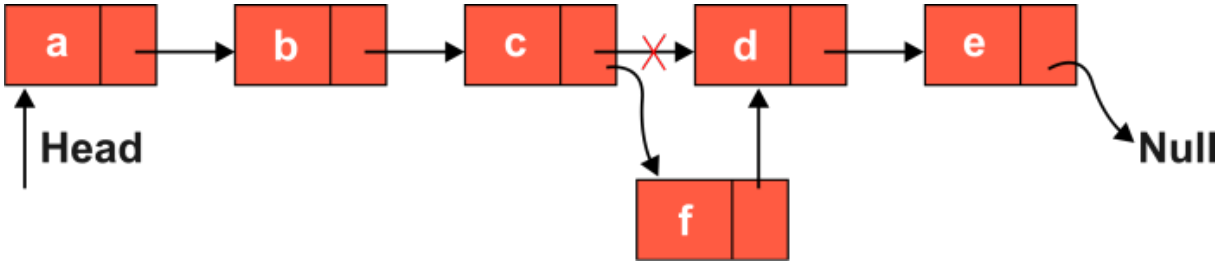
Hình 10: *Ý tưởng Chèn ở cuối danh sách*

Chèn cuối Danh sách ta phải cập nhật *tail* bằng giá trị *newNode* và *tail* cũ trỏ tới *newNode*

```
//! Insert ở cuối danh sách (tail)
//^ độ phức tạp O(1)
else if(index == this->size){
    tail->next = newNode;
    tail = newNode;
}
```

Hình 11: *Chèn ở cuối danh sách*

2.4.5 Chèn giữa Danh sách



Hình 12: *Ý tưởng Chèn ở giữa danh sách*

Chèn cuối Danh sách ta phải duyệt *temp* tới node phía trước vị trí cần chèn *nodeindex* – 1 cập nhật con trỏ *newNode* trỏ tới Node trước node *temp* sau đó node *temp* trỏ tới *newNode*

```
    //! Insert ở giữa danh sách (mid) - các trường hợp còn lại
    //^ độ phức tạp O(N)
    else{
        Node* temp = head;
        //! tìm kiếm node thứ index - 1 (node trước newNode cần insert)
        for(int i = 1 ; i < index; i++) temp = temp->next;
        //! newNode trỏ đến node index
        newNode->next = temp->next;
        //! node index - 1 trỏ đến NewNode
        temp->next = newNode;
    }
    this->size ++;  //! tăng số lượng node
}
```

Hình 13: *Chèn ở giữa danh sách*

2.5 Deletion

```
//* Deletion có 5 TH: index ngoài phạm vi, danh sách 1 phần tử, chèn đầu, chèn cuối, chèn mid
//* cuối and mid gộp chung
//^ độ phức tạp O(n)
void Singly_Linked_List::Deletion(int index){···
```

Hình 14: *Deletion Singly Linked List*

Giải Thích: Hàm xóa tạo Danh sách liên kết đơn

- $index < 0$ && $index > size$ trường hợp ngoại phạm vi nên nén ra lỗi, độ phức tạp $O(1)$.
- $size = 1$ Danh sách đang một phần tử, độ phức tạp $O(1)$.
- $index = 0$ xóa đầu Danh sách, độ phức tạp $O(1)$.
- $index = size - 1$ xóa cuối Danh sách, độ phức tạp $O(N)$.
- *else* trường hợp còn lại nằm giữa danh sách, độ phức tạp $O(N)$.

2.5.1 Index nằm ngoại phạm vi

Vì giá trị *index* khi truyền vào hàm *delete* có thể nằm ngoài giá trị cho phép của $index > 0$ and $index <= size$ ra sẽ nén(*thow*) ra lỗi.

```
//! trường hợp index nằm ngoại phạm vi
if(index < 0 || index >= this->size) out_of_range("Index of Linked List is invalid!");
```

Hình 15: *index nằm ngoại phạm vi*

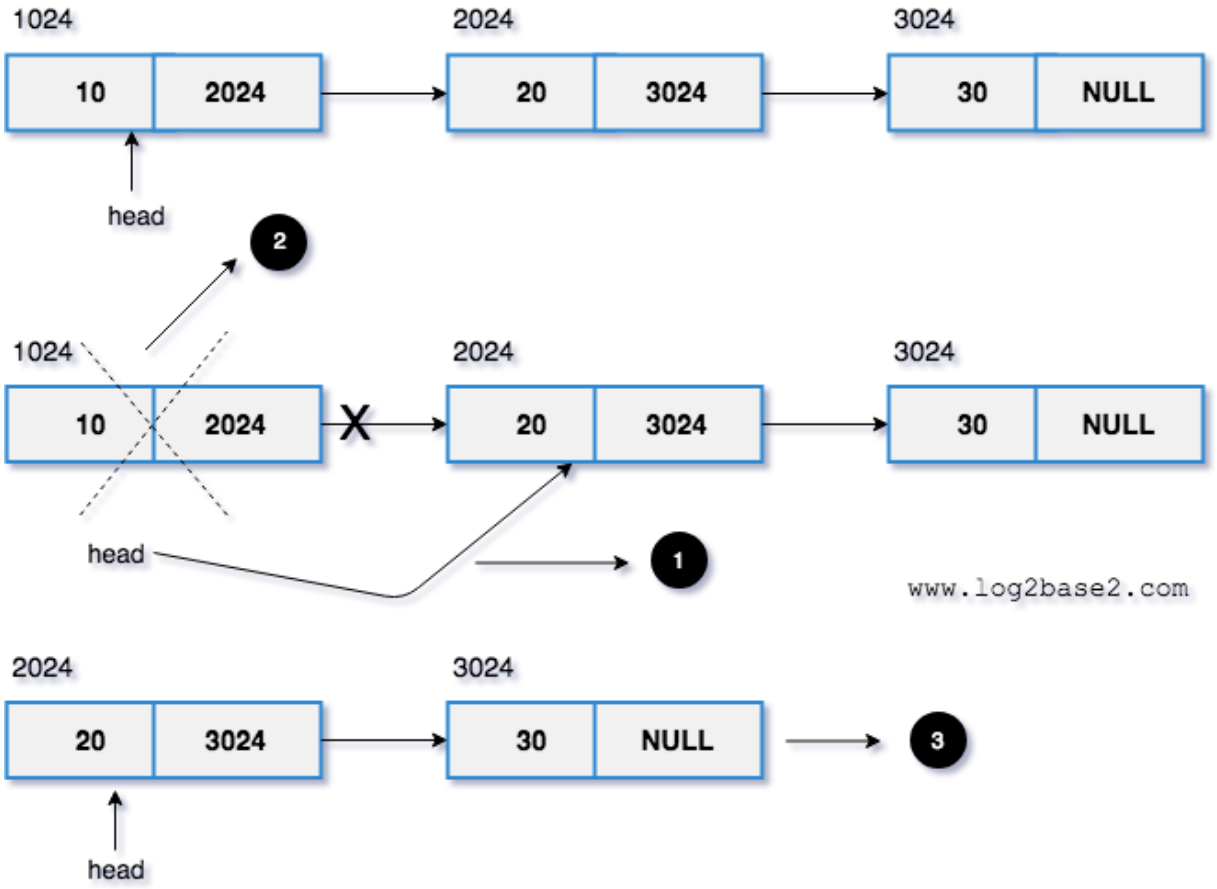
2.5.2 Danh sách có 1 phần tử

Khi xóa danh sách còn 1 phần tử ta phải cập nhật *head*, *tail* gán giá trị *nullptr*

```
//! Singly_Linked_List đang có 1 phần tử
//^ độ phức tạp O(1)
if(this->size == 1){
    deleteNode = head;
    head = tail = nullptr;
}
```

Hình 16: *Danh sách có 1 phần tử*

2.5.3 Delete ở đầu danh sách



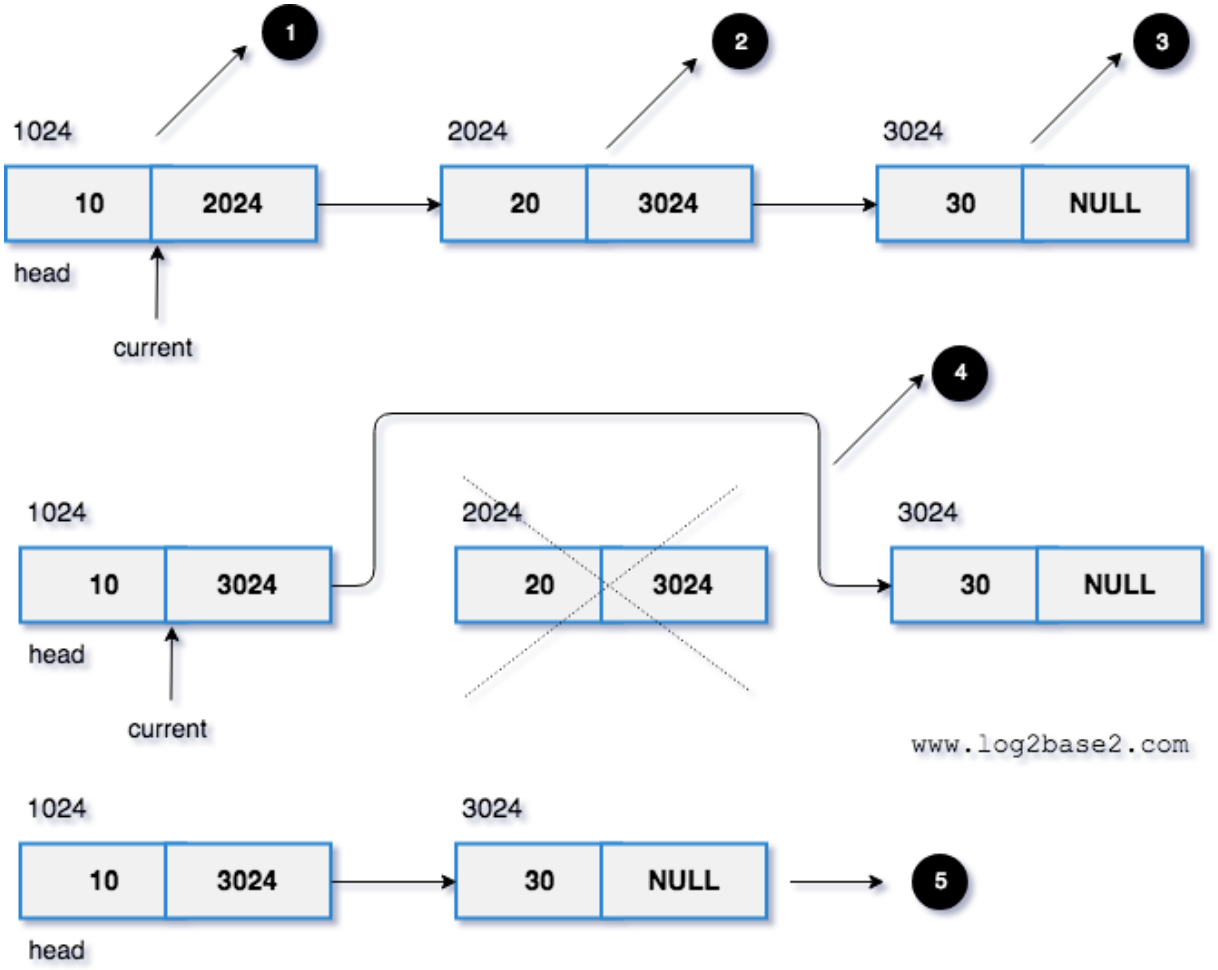
Hình 17: *Ý tưởng Delete ở đầu danh sách*

Khi xóa phần tử đầu ta cần cập nhật *head* bằng cách dịch *head* lên phần tử trước nó

```
//! Delete ở đầu danh sách (head)
//^ độ phức tạp O(1)
else if(index == 0){
    deleteNode = head;
    head = head->next;
}
```

Hình 18: *Delete ở đầu danh sách*

2.5.4 Delete ở cuối và giữa danh sách



Hình 19: *Ý tưởng Delete ở cuối và giữa danh sách*

Khi xóa cần tìm thấy $nodeIndex - 1$ trước phái trước $node$ cần xóa và lưu node vần xóa lại và cập nhật $nodeIndex - 1$ trỏ tới node phía sau node cần xóa $nodeIndex + 1$, nếu trường hợp node xóa là node cuối thì cần nhập nhật $tail$ lên node trước đó.

```
//! Delete ở cuối danh sách (tail) or Delete ở giữa danh sách (mid) - các trường hợp còn lại
//^ độ phức tạp O(n)
else{
    Node* temp = head;
    //! tìm kiếm node thứ index - 1 (node trước newNode cần insert)
    for(int i = 1 ; i < index; i++) temp = temp->next;
    //! deleteNode node cần xóa index
    deleteNode = temp->next;
    //! node index - 1 trỏ đến index + 1;
    temp->next = deleteNode->next;
    //! trường hợp node cuối cùng xóa đi tail -> cập nhật tail lên node trước node Index - 1
    if(index == this->size - 1) tail = temp;
}
delete deleteNode;
this->size --; //! giảm số lượng node
```

Hình 20: *Delete ở cuối và giữa danh sách*

2.6 Get

```cpp
//* duyệt tuyến tính(liner) đến phần tử cần lấy data
//^ độ phức tạp O(n)
int Singly_Linked_List::Get(int index){
    //! trường hợp index nằm ngoại phạm vi
    if(index < 0 || index >= this->size) out_of_range("Index of Linked List is invalid!");

    Node* temp = head;
    //! Get node thứ index (node cần tìm)
    for(int i = 0 ; i < index; i++) temp = temp->next;
    return temp->data;
}
```

Hình 21: *Ý tưởng*

2.7 IndexOf

```cpp
//* duyệt tuyến tính(liner) hết phần tử khi nào gặp phần tử data giống thôi
//^ độ phức tạp O(n)
int Singly_Linked_List::IndexOf(int data){
    Node* temp = head;
    //! IndexOf node đầu tiên có data = với data truyền vào
    for(int i = 0 ; i < this->size; i++){
        if(temp->data == data){
            return i;
        }
        temp = temp->next;
    }
    //! không tìm thấy vị trí
    return -1;
}
```

Hình 22: *Ý tưởng*

2.8 Contains

```cpp
//* duyệt tuyến tính(liner) hết phần tử khi nào gặp phần tử data giống thôi
//^ độ phức tạp O(n)
bool Singly_Linked_List::Contains ( int data ){
    Node* temp = head;
    //! Contains tìm kiếm thử data có nằm trong link list hay không
    for(int i = 0 ; i < this->size; i++){
        if(temp->data == data){
            return true;
        }
        temp = temp->next;
    }
    //! không tìm thấy vị trí
    return false;
}
```

Hình 23: *Ý tưởng*

2.9 SelecetionSort

```cpp
//* hàm này có 2 cách là swap data và swap Node
//* phần này dùng swap data còn swap node tự làm nha khá là khoai
//^ độ phức tạp O(n^2)
void Singly_Linked_List::SelecetionSort(){
    Node* node_I = head;
    while(node_I != nullptr){
        Node* node_J = node_I->next;
        while(node_J != nullptr){
            if(node_I->data > node_J->data) swap(node_I->data, node_J->data);
            node_J = node_J->next;
        }
        node_I = node_I->next;
    }
}
```

Hình 24: *Ý tưởng*

2.10 Concat

```cpp
//* hàm này sẽ tạo ra 2 Singly_Linked_List mới từ this và other và hợp chúng lại
Singly_Linked_List Singly_Linked_List::Concat(const Singly_Linked_List& others){
    Singly_Linked_List h1(*this);
    Singly_Linked_List h2(others);

    //! nếu size this khác không
    if(h1.size != 0) h1.tail->next = h2.head;
    else h1 = h2;
    h1.size += h2.size; //~ cập nhật size
    return h1;
}
```

Hình 25: *Ý tưởng*

2.11 SubSingly_Linked_List

```cpp
//* Deletion có 3 TH: lỗi range, lỗi logic, cắt
//^ độ phức tạp O(n)
Singly_Linked_List Singly_Linked_List::SubSingly_Linked_List(int from, int to){
    //!lỗi range
    if(from < 0 || from >= size || to <= 0 || to > size) out_of_range("Index of Link List is invalid!");
    //! lỗi logic
    else if(from >= to) logic_error("Invalid range");

    Singly_Linked_List result;
    Node* temp = head;
    //! tìm đến vị trí cần cắt
    for(int i = 0 ; i < from; i++){
        temp = temp->next;
    }

    for(int i = from; i < to; i++){
        result.Insertion(temp->data, i - from); // insert tail
        temp = temp->next;
    }
    return Singly_Linked_List(result);
}
```

Hình 26: *Ý tưởng*

2.12 Reverse

```cpp
//^ độ phức tạp O(n)
void Singly_Linked_List::Reverse(){
    if(this->size <= 1) return;
    //! curr là con trỏ hiện tại, pre là con trỏ phía trước
    Node* curr = this->head;
    Node* pre = this->head->next;
    for(int i = 0; i < this->size - 1; i++){
        //! lưu lại node i + 2
        Node* temp = pre->next;
        //! trỏ node i + 1 tới node i
        pre->next = curr;
        //! gán curr thành node i + 1 và pre thành node i + 2 (tăng lên)
        curr = pre;
        pre = temp;
    }

    //! cập nhật lại node đầu thành node cuối
    this->head->next = nullptr;
    this->tail = this->head;
    //! cập nhật node cuối thành node đầu
    this->head = curr; //! node cuối đang là curr
}
```

Hình 27: *Ý tưởng*

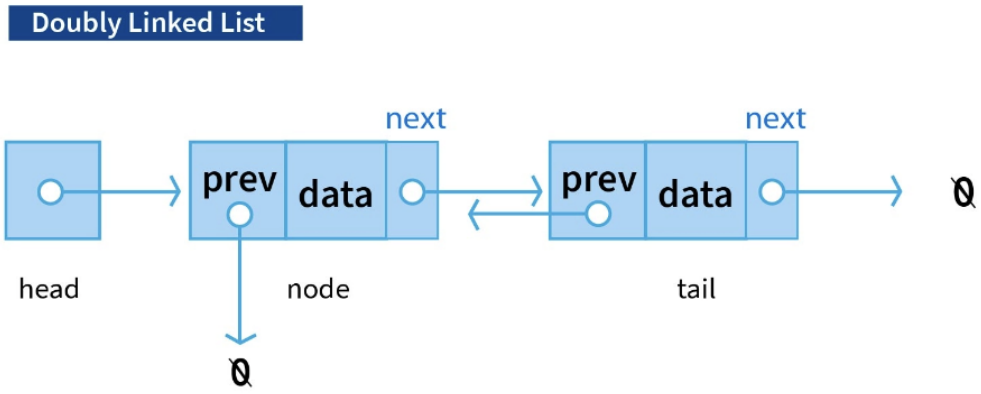
2.13 Dislay

```cpp
//^ độ phức tạp O(n)
string Singly_Linked_List::Dislay(){
    string result = "[Size=" + to_string(size)
                    + ",DataHead=" + (head ? to_string(head->data) : "nullptr")
                    + ",DataTail=" + (tail ? to_string(tail->data) : "nullptr")
                    + ",DataNode: ";
    if(this->size == 0) return result + "nullptr]";
    Node* temp = head;
    for(int i = 0 ; i < this->size; i++){
        result += to_string(temp->data) + "->";
        temp = temp->next;
        if(i == this->size - 1) result += "nullptr]";
    }
    return result;
}
```

Hình 28: *Ý tưởng*

3 Double Linked List

Một danh sách liên kết đôi (Double Linked List) là một cấu trúc dữ liệu tuyến tính trong đó mỗi phần tử chứa một giá trị và một liên kết đến phần tử tiếp theo và một liên kết đến phần tử trước đó.



Hình 29: *Singly Linked List*

3.1 Implementation Double Linked List in C++

Ta sử dụng class trong class (Nested class) nghĩa là class Node trong class LinkList, có nhiều cách có thể làm class Node ở bên ngoài đập lập với class LinkList.
Class Node nằm trong danh sách liên kết và lưu trữ data có kiểu dữ liệu int(có thể float, OOP, string... tùy vào bài).

```
public:
    class Node{
    public:
        int data;   //! dữ liệu
        Node* next; //! trỏ tới node tiếp theo (phía sau)
        Node* prev; //! con trỏ phái trước
    public:
        Node(int data = 0, Node* next = nullptr, Node* prev = nullptr)
        :data(data), next(next), prev(prev){}
    };
};
```

Hình 30: *class Node Double Linked List*

```
class Double_Linked_List
{
public:
    class Node;
protected:
    Node* head, *tail;  //! Node đầu tiên và Node cuối cùng
    int size;           //! kích thước của 1
public:
```

Hình 31: *thuộc tính Double Linked List*

Danh sách các nhóm hàm được thực hiện.

- khởi tạo gồm Contructor và Destructor
- chỉnh sửa gồm insert và delete
- lấy giữ liệu gồm count(đếm), empty(rỗng), get(lấy), indexof(lấy chỉ số), contains(chứa)
- in ra gồm dislay
- sắp xếp dữ liệu Sort
- một số hàm khác concat(nối), sub(cắt), reverse(đảo ngược)
- các hàm count(đếm), empty(rỗng), indexof(lấy chỉ số), contains(chứa),Sort, concat(nối), sub(cắt) **giống singly linked list**

3.2 Contructor(Khởi tạo)

Hàm khởi tạo(được gọi khi khai báo) giá trị cho *head*, *tail* và *size* nhưng thường thì chỉ khởi tạo *nullptr* cho *head* và *tail*, $size = 0$ trường hợp không có tham số đầu vào độ phức tạp sẽ là $O(1)$, nếu có tham số đầu vào hoặc xử lý 1 gì đó bang đầu như khởi tạo *static* ta phải thực hiện, trong phần này truyền vào là 1 danh sách liên kết nên cần copy tất cả các node ra danh sách liên kết cần khởi tạo độ phức tạp sẽ là $O(n)$ vì hàm Insertion chèn ở cuối độ phức tạp $O(1)$

```cpp
//* tùy vào tham số truyền vào mà ta có thể có nhiều contructor
//* Thông thường thì chỉ cần rỗng và tham số truyền vào chính nó
//^ độ phức tạp O(1)
Double_Linked_List::Double_Linked_List()
:head(nullptr), tail(nullptr), size(0){

};

//* so chép tất cả các dữ liệu other sang this
//* luôn khởi tạo head tail và size trước khi xử lí
//^ độ phức tạp O(n)
Double_Linked_List::Double_Linked_List(const Double_Linked_List& others)
:head(nullptr), tail(nullptr), size(0){
    Node* temp = others.head;
    for(int i = 0; i < others.size; i++)
    {
        this->Insertion(temp->data, i); //! chèn tại cuối
        temp = temp->next;  //! dịch ptr temp
    }
}
```

Hình 32: *Contructor Double Linked List*

3.3 Destructor(Hàm hủy)

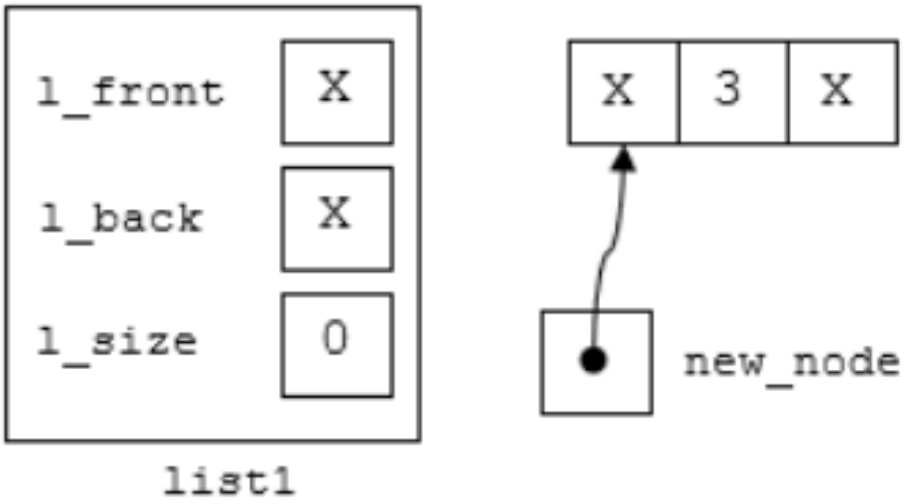
Hàm hủy(được gọi khi thu hồi) vì trong này vẫn còn nhiều *Node* đang sử dụng nên cần phải xóa để không bị leak memory xóa bằng cách duyệt qua tất cả các *Node* từ *head* đến *tail*, khi thu hồi xong cần cập nhật lại *head*, *tail*, *size*. Hàm này có thể giống hàm *clear*, độ phức tạp $O(N)$.

```cpp
//* xóa tất cả các phần tử Node đang còn trong Double_Linked_List
//* hàm này tương đương với clear nhưng chỉ gọi ghi kết thúc phạm vi
//* có thể dùng hàm remove để xử lý
//^ độ phức tạp O(n)
Double_Linked_List::~Double_Linked_List(){
    Node* temp; //~ node mẫu
    //! có thể dùng hàm Deletion như phần contructor phía trên
    while(size != 0){
        //! dịch ptr head trước khi xóa không sẽ bị lỗi Dangling pointer
        temp = head;
        head = head->next;
        delete temp;
        size --;
    }
    //! xóa hết rồi gán lại ban đầu thôi có thể có hay không thì cũng được
    //! nhưng hàm clear thì phải có
    head = tail = nullptr;
    size = 0;
}
```

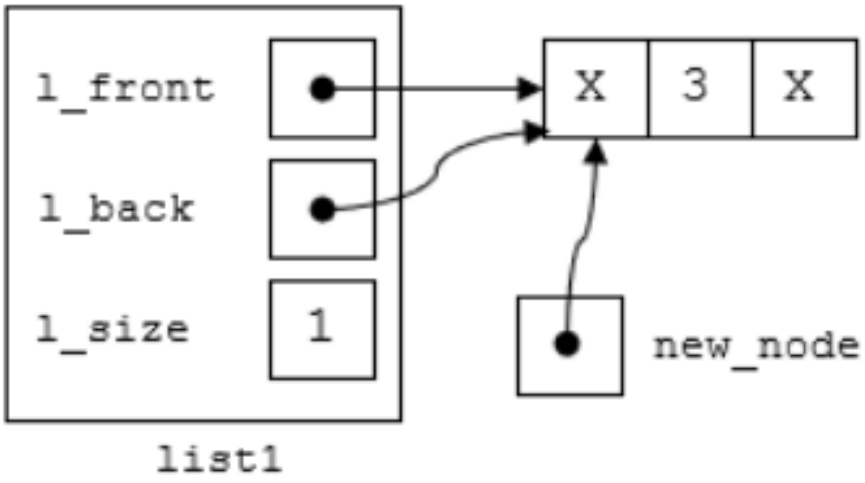
Hình 33: *Destructor Double Linked List*

3.4 Insertion

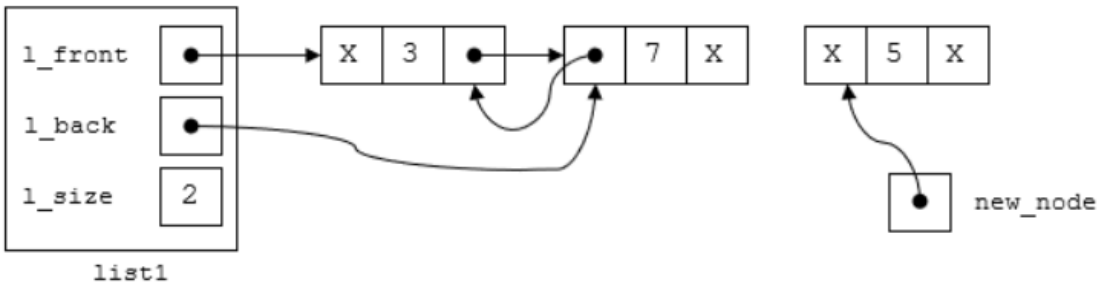
Xử lí 2 trường hợp chính là List is not empty và List is empty ý tưởng trong 4 hình sau.



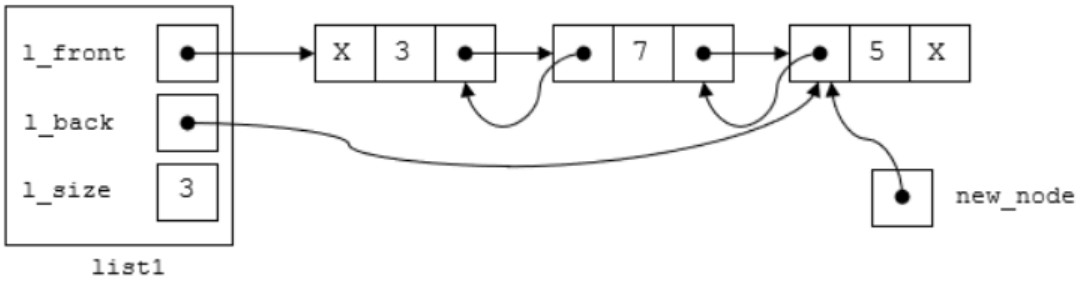
Hình 34: *List is empty trước*



Hình 35: *List is empty sau*



Hình 36: *List is not empty trước*



Hình 37: *List is not empty sau*

```
//* insert có 5 TH: index ngoài phạm vi, danh sách rỗng, chèn đầu, chèn cuối, chèn mid
//* cuối and mid có thể chung nhau
//* đầu và rỗng có thể chung nhau
//* data : dữ liệu cần chèn, index là vị trí (vị trí đầu tiên 0, cuối cùng là size)
//^ độ phức tạp O(n)
void Double_Linked_List::Insertion(int data,int index){···
```

Hình 38: *Ý tưởng*

3.4.1 Index nằm ngoại phạm vi

Vì giá trị *index* khi truyền vào hàm *insert* có thể nằm ngoài giá trị cho phép của *index* >= 0 and *index* <= *size* ra sẽ nén(*thow*) ra lỗi.

```
//! trường hợp index nằm ngoại phạm vi
if(index < 0 || index > this->size) out_of_range("Index of Linked List is invalid!");
```

Hình 39: *index nằm ngoại phạm vi*

3.4.2 Danh sách đang rỗng(empty)

Khi danh sách đang rỗng lúc này *head*, *tail* đều mang giá trị *Nullptr* cần cập nhật giá trị cả 2 bằng *newNode* node cần chèn, tăng *size* lên 1.

```
Node* newNode = new Node(data);
//! Double_Linked_List đang rỗng (empty)
//^ độ phức tạp O(1)
if(this->size == 0){
    head = tail = newNode;
}
```

Hình 40: *danh sách rỗng (empty)*

3.4.3 Chèn đầu Danh sách

Chèn đầu Danh sách ta phải cập nhật *head* bằng giá trị *newNode* và *newNode* trỏ tới *head* cũ

```
//! Insert ở đầu danh sách (head)
//^ độ phức tạp O(1)
else if(index == 0){
    newNode->next = head;
    head->prev = newNode;
    head = newNode;
}
```

Hình 41: *Chèn ở đầu danh sách*

3.4.4 Chèn Cuối Danh sách

Chèn cuối Danh sách ta phải cập nhật *tail* bằng giá trị *newNode* và *tail* cũ trỏ tới *newNode*

```
//! Insert ở cuối danh sách (tail)
//^ độ phức tạp O(1)
else if(index == this->size){
    tail->next = newNode;
    newNode->prev = tail;
    tail = newNode;
}
```

Hình 42: *Chèn ở cuối danh sách*

3.4.5 Chèn giữa Danh sách

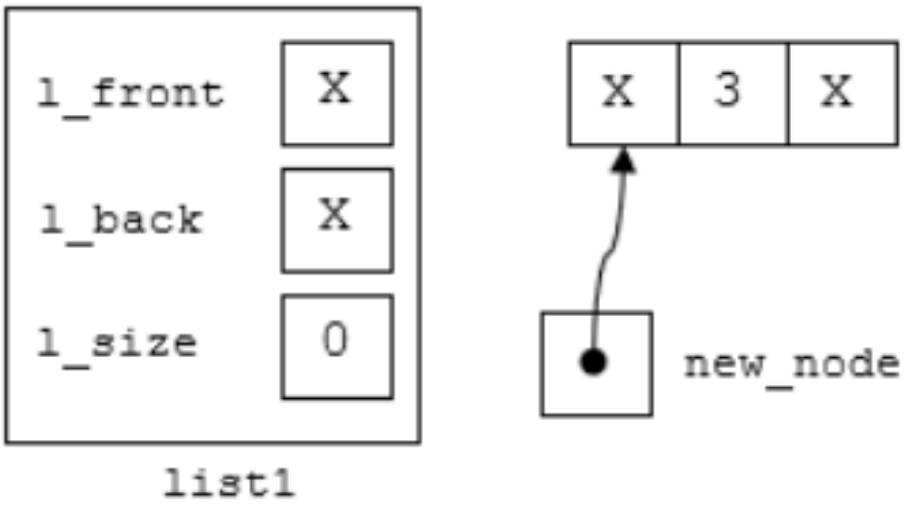
Chèn cuối Danh sách ta phải duyệt *temp* tới node phía trước vị trí cần chèn $nodeindex - 1$ cập nhật con trỏ $newNode$ trỏ tới Node trước node *temp* sau đó node *temp* trỏ tới $newNode$

```
//! Insert ở giữa danh sách (mid) - các trường hợp còn lại
//^ độ phức tạp O(N)
else{
    Node* temp = head;
    //! tìm kiếm node thứ index - 1 (node trước newNode cần insert)
    for(int i = 1 ; i < index; i++) temp = temp->next;
    //! newNode trỏ đến node index
    newNode->next = temp->next;
    //! node index - 1 trỏ đến NewNode
    temp->next = newNode;
    //! NewNode prev trỏ node index - 1;
    newNode->prev = temp;
    //! index prev trỏ NewNode
    newNode->next->prev = newNode;
}
this->size ++;  //! tăng số lượng node
```

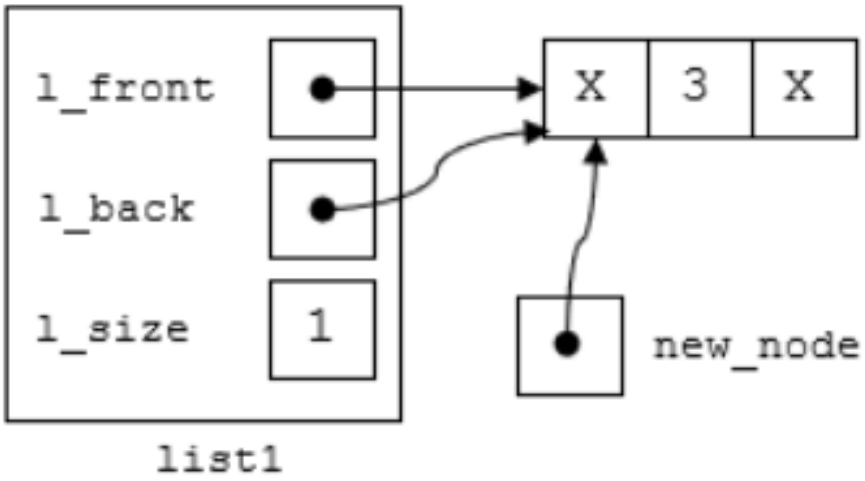
Hình 43: *Chèn ở giữa danh sách*

3.5 Deletion

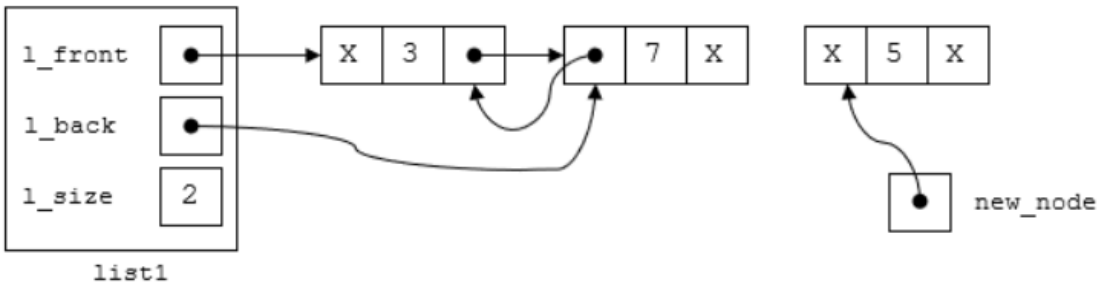
Xử lí 2 trường hợp chính là List contains more than one node và List contains one node ý tưởng trong 4 hình sau.



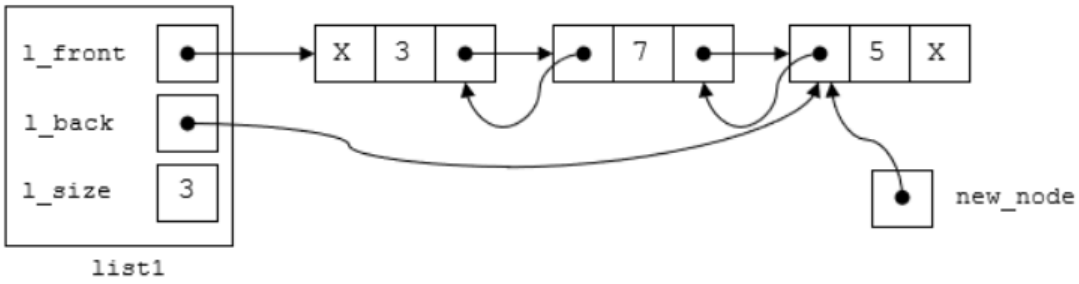
Hình 44: *List contains one node trước*



Hình 45: *List contains one node sau*



Hình 46: *List contains more than one node trước*



Hình 47: *List contains more than one node sau*

```
//* Deletion có 5 TH: index ngoài phạm vi, danh sách còn 1 node, chèn đầu, chèn cuối, chèn mid
//* cuối and mid gộp chung
//^ độ phức tạp O(n)
void Double_Linked_List::Deletion(int index){···
```

Hình 48: *Ý tưởng*

3.5.1 Index nằm ngoại phạm vi

Vì giá trị *index* khi truyền vào hàm *delete* có thể nằm ngoài giá trị cho phép của *index* > 0 and *index* <= *size* ra sẽ nén(*thow*) ra lỗi.

```
//! trường hợp index nằm ngoại phạm vi
if(index < 0 || index >= this->size) out_of_range("Index of Linked List is invalid!");
```

Hình 49: *index nằm ngoại phạm vi*

3.5.2 Danh sách có 1 phần tử

Khi xóa danh sách còn 1 phần tử ta phải cập nhật *head*, *tail* gán giá trị *nullptr*

```
//! Double_Linked_List đang có 1 phần tử
//^ độ phức tạp O(1)
if(this->size == 1){
    deleteNode = head;
    head = tail = nullptr;
}
```

Hình 50: *Danh sách có 1 phần tử*

3.5.3 Delete ở đầu danh sách

Khi xóa phần tử đầu ta cần cập nhật *head* bằng cách dịch *head* lên phần tử trước nó

```
//! Delete ở đầu danh sách (head)
//^ độ phức tạp O(1)
else if(index == 0){
    deleteNode = head;
    head = head->next;
    head->prev = nullptr;
}
```

Hình 51: *Delete ở đầu danh sách*

3.5.4 Delete ở cuối và giữa danh sách

Khi xóa cần tìm thấy $nodeIndex - 1$ trước phái trước $node$ cần xóa và lưu node vần xóa lại và cập nhật $nodeIndex - 1$ trỏ tới node phía sau node cần xóa $nodeIndex + 1$, nếu trường hợp node xóa là node cuối thì cần nhập nhật $tail$ lên node trước đó.

```
}
//! Delete ở cuối danh sách (tail) or Delete ở giữa danh sách (mid) - các trường hợp còn lại
//^ độ phức tạp O(n)
else{
    Node* temp = head;
    //! tìm kiếm node thứ index - 1 (node trước newNode cần insert)
    for(int i = 1 ; i < index; i++) temp = temp->next;
    //! deleteNode node cần xóa index
    deleteNode = temp->next;
    //! node index - 1 trỏ đến index + 1;
    temp->next = deleteNode->next;
    //! trường hợp node cuối cùng xóa đi tail -> cập nhật tail lên node trước node Index - 1
    if(index == this->size - 1) tail = temp;
    //! xét thử node index có nullptr không rồi xét node index + 1 prev trỏ node index - 1
    else temp->next->prev = temp;
}
delete deleteNode;
this->size --; //! giảm số lượng node
```

Hình 52: *Delete ở cuối và giữa danh sách*

3.6 Get

```
//^ độ phức tạp O(n)
int Double_Linked_List::Get(int index){
    //! trường hợp index nằm ngoại phạm vi
    if(index < 0 || index >= this->size) out_of_range("Index of Linked List is invalid!");

    Node* temp;
    //! duyệt từ node head -> node size / 2 theo chiều xuôi
    if(index <= size / 2){
        temp = head;
        //! Get node thứ index (node cần tìm)
        for(int i = 0 ; i < index; i++) temp = temp->next;
    }
    //! duyệt từ node tail -> node size / 2 theo chiều ngược
     else{
        //! Get node thứ index (node cần tìm)
        temp = tail;
        for(int i = 0 ; i < size - index; i++) temp = temp->prev;
    }
    return temp->data;
}
```

Hình 53: *Ý tưởng*

3.7 Reverse

```cpp
//^ độ phức tạp O(n)
void Double_Linked_List::Reverse(){
    if(this->size <= 1) return;
    //! curr là con trỏ hiện tại, pre là con trỏ phía trước
    Node* curr = this->head;
    Node* pre = this->head->next;
    for(int i = 0; i < this->size - 1; i++){

        //! lưu lại node i + 2
        Node* temp = pre->next;
        //! next trỏ node i + 1 tới node i
        //! prev node i tới node i + 1
        pre->next = curr;
        curr->prev = pre;
        //! gán curr thành node i + 1 và pre thành node i + 2 (tăng lên)
        curr = pre;
        pre = temp;

    }
    //! cập nhật lại node đầu thành node cuối
    this->head->next = nullptr;
    this->tail = this->head;
    //! cập nhật node cuối thành node đầu
    curr->prev = nullptr;
    this->head = curr; //! node cuối đang là curr
}
```

Hình 54: *Ý tưởng*

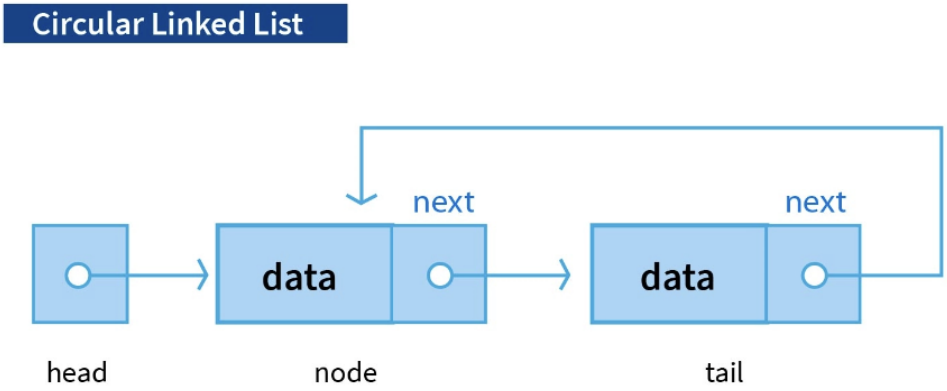
3.8 Dislay

```cpp
//^ độ phức tạp O(n)
string Double_Linked_List::Dislay(){
    string result = "[Size=" + to_string(size)
                    + ",DataHead=" + (head ? to_string(head->data) : "nullptr")
                    + ",DataTail=" + (tail ? to_string(tail->data) : "nullptr")
                    + ",DataNode: ";
    if(this->size == 0) return result + "nullptr]";
    Node* temp = head;
    result += "\n           next: ";
    for(int i = 0 ; i < this->size; i++){
        result += to_string(temp->data) + "->";
        temp = temp->next;
        if(i == this->size - 1) result += "nullptr";
    }
    result += "\n           prev: ";
    temp = tail;
    for(int i = 0 ; i < this->size; i++){
        result += to_string(temp->data) + "->";
        temp = temp->prev;
        if(i == this->size - 1) result += "nullptr]";
    }
    return result;
}
```

Hình 55: *Ý tưởng*

4 Circular Linked List(ít sài nên tự đọc code)

Một danh sách liên kết vòng (Circular Linked List) là một cấu trúc dữ liệu tuyến tính trong đó mỗi phần tử chứa một giá trị và một liên kết đến phần tử tiếp theo và node cuối trỏ tới node đầu tiên nên không một node nào trỏ tới *nullptr*



Hình 56: *Singly Linked List*

4.1 Implementation Double Linked List in C++

Cũng giống như Single Linked List và vì node *tail* trỏ đến node *head* nên ta cần giữ node *tail* và xử lí nó tương tự Single Linked List