**04. 리스트**

**04-1. 선형 리스트**

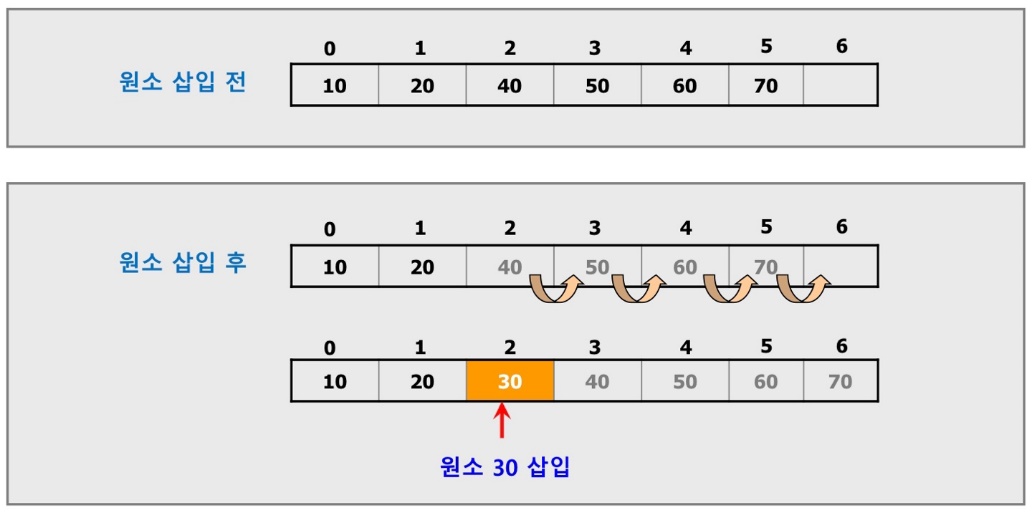
**# 리스트**

: 목록, 대부분의 목록은 도표(Table) 형태로 표시(추상 자료형 리스트는 이러한 목록 또는 도표를 추상화한 것)

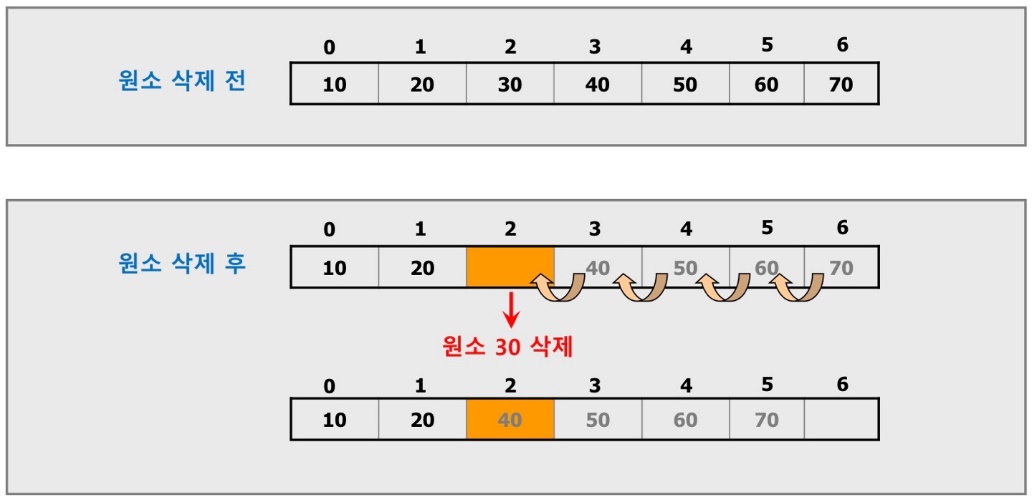
**# 선형 리스트(Linear List)**

**: 순서 리스트(Oredered List)**

* 리스트에서 나열한 원소들 간에 순서를 가지고 있는 리스트
* 원소들 간의 논리적인 순서와 물리적인 순서가 같은 구조(순차 자료구조)
* **원소 삽입**



* **원소 삭제**

****

**# 선형 리스트 구현**

* **1차원 배열의 순차 표현**
* int arr[4] = {70, 80, 90, 240};
* 2차원 배열의 숫자 표현 : 행과 열의 구조로 나타내는 배열(메모리에 저장될 때에는 1차원의 순서로 저장)
* int score[3][4]={

{70, 80, 90},

{50, 60, 70},

{60, 70, 80},

};

**04-2. 연결 리스트**

**# 순차 선형 리스트의 문제점**

* **리스트의 순서 유지를 위해 원소들의 삽입과 삭제가 어렵다.**
  + 삽입 또는 삭제 연산 후에 연속적인 물리 주소를 유지하기 위해서 원소들을 이동시키는 추가적인 작업과 시간이 소요된다.

- 원소들의 빈번한 이동 작업으로 인한 오버헤드가 발생

- 원소의 개수가 많고 삽입과 삭제 연산이 많이 발생하는 경우 더 많이 발생한다.

**- 메모리 사용의 비효율성 :** 최대한의 크기를 가진 배열을 처음부터 준비해 두어야 하기 때문에 기억장소의 낭비를 초래할 수 있다.

**# 연결 리스트**

* **순차 자료구조에서의 연산 시간에 대한 문제와 저장 공간에 대한 문제를 개선한 자료 표현 방법**
  + 연결 자료구조(Linked Data structure)
  + 비 순차 자료구조(Non sequential Data structure)
  + 데이터 아이템을 줄줄이 엮은(Link, Chain) 것
* **노드(Node) : <원소, 주소>단위로 저장**
  + 데이터 필드(Data Field) : 원소의 값을 저장
  + 링크 필드(Link Field) : 노드의 주소를 저장

**# 자기 참조 구조체**

**: 자신의 구조체 자료형을 가리키는 포인터 멤버를 가질 수 있다.**

struct \_score {

char name[12];

int kor, eng, math, tot;

float ave;

struct \_score\* link;

}; typedef struct \_score SCORE;

* link 멤버는 자신과 같은 구조의 구조체 주소를 저장하고 있다가 필요 시 저장된 주소의 구조체에 접근하는 것을 목표로 한다.

**# 자기 참조 구조체 : 구조체 노드의 생성**

// struct score \*head, \*new\_Node;

SCORE\* head, \* newNode;

head = NULL;

//SCORE 크기의 메모리 할당

newNode = (SCORE\*)malloc(sizeof(SCORE));

if (newNode == NULL) {

printf(“메모리 할당 실패!!!\n);

exti(100);

}



**04-3. 단순 연결 리스트(Singly Linked List)**

: 선형 연결 리스트\_단순 연결 선형 리스트(singly linked linear list)

// 노드: 데이터 링크

class SNode {

private:

int \_\_data;

SNode\* \_\_link.

friend class SLinkedList;

};

// 단순 연결 리스트

class SLinkedList {

private:

SNode\* \_\_head; // 첫 번째 노드

// SNode\* \_\_tail; // 맨 마지막 노드

// int \_\_count; // 노드의 총 개수

public:

SLinkedList(); // 생성자

~SLinkedList(); // 소멸자: 전체 노드 삭제

void addRear(const int& e); // 삽입: 맨 마지막 노드

void removeFront(); // 삭제: 첫 번째 노드

SNode\* frontSNode() const; // 탐색: 첫 번째 노드

SNode\* rearSNode() const; // 탐색: 맨 마지막 노드

bool isEmpty() const; // 빈 리스트 여부 판단

int countSNode() const; // 탐색: 노드의 총 개수(count)

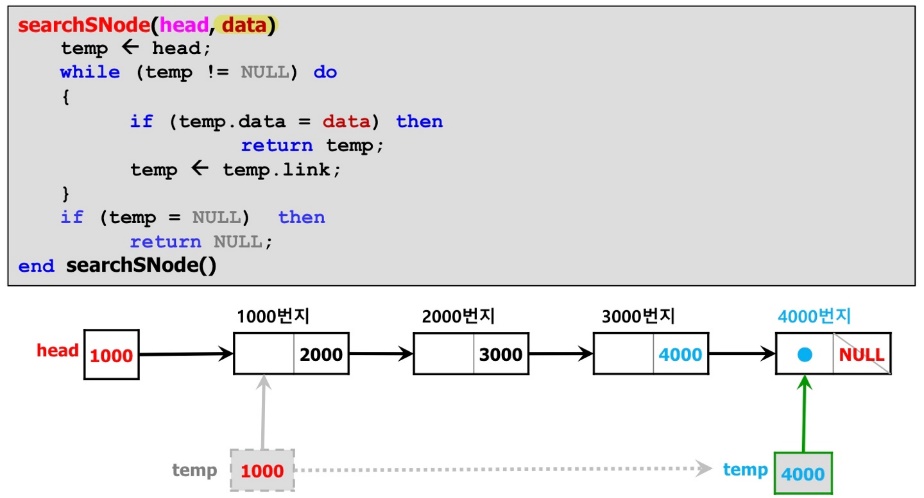
void printSLinkedList() const; // 리스트의 전체 원소(노드) 출력

};

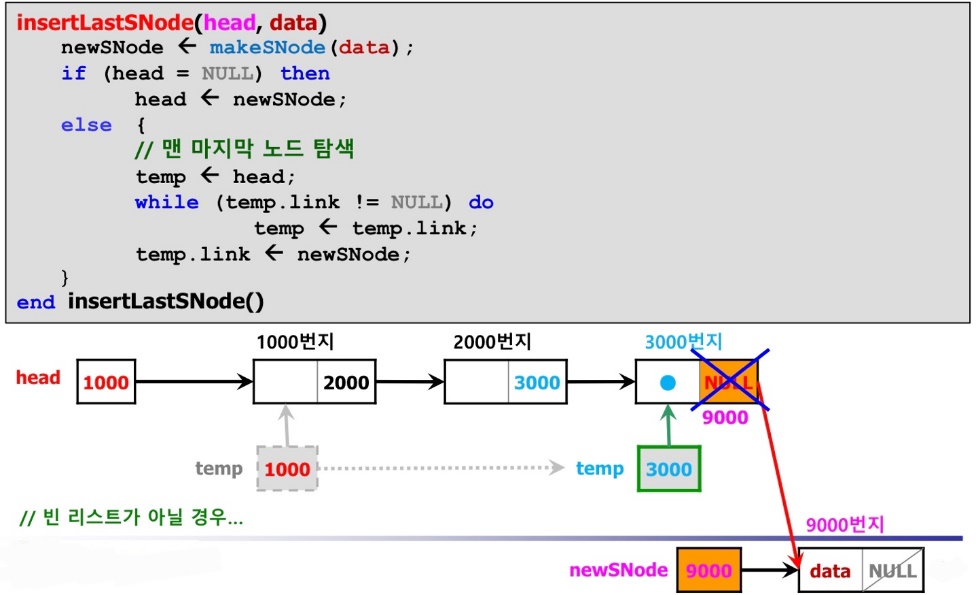
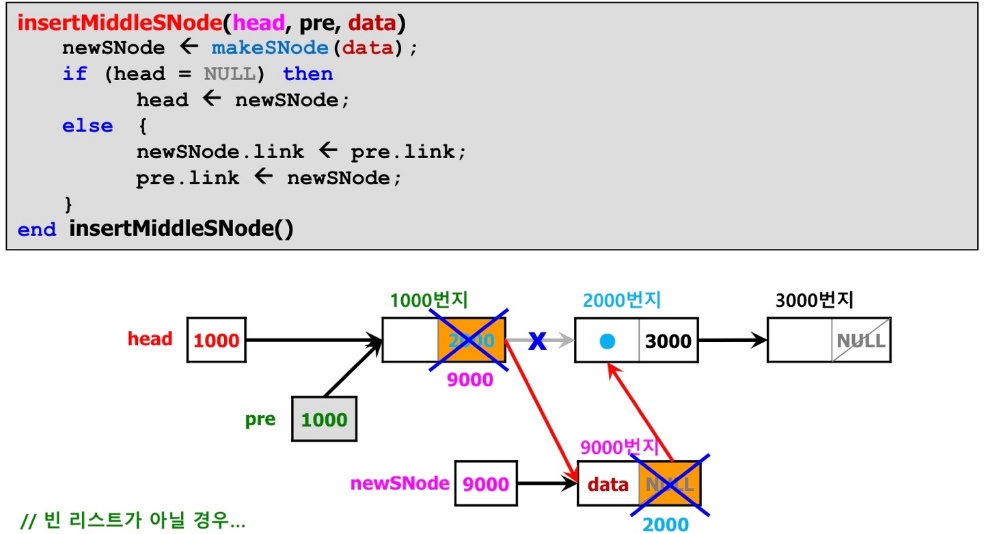
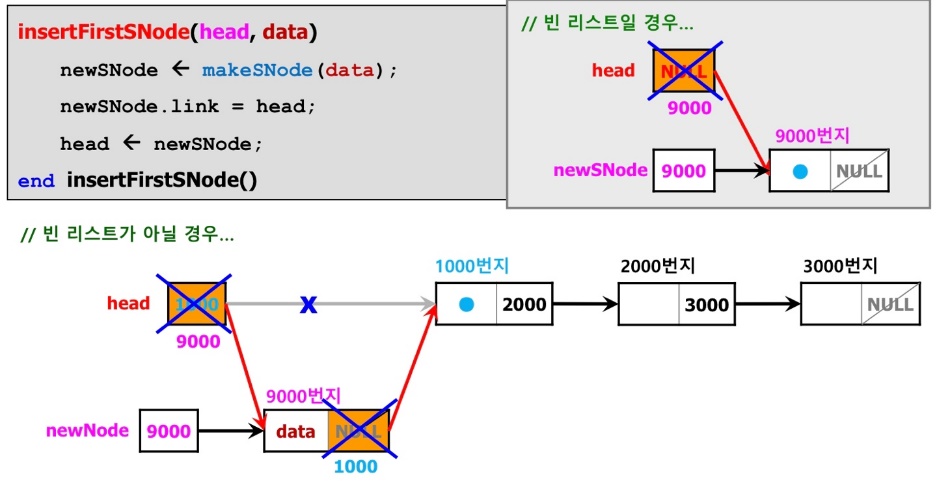
**# 알고리즘**

* **탐색 알고리즘**

: 리스트에서 조건을 만족하는 데이터를 가진 노드 탐색 알고리즘

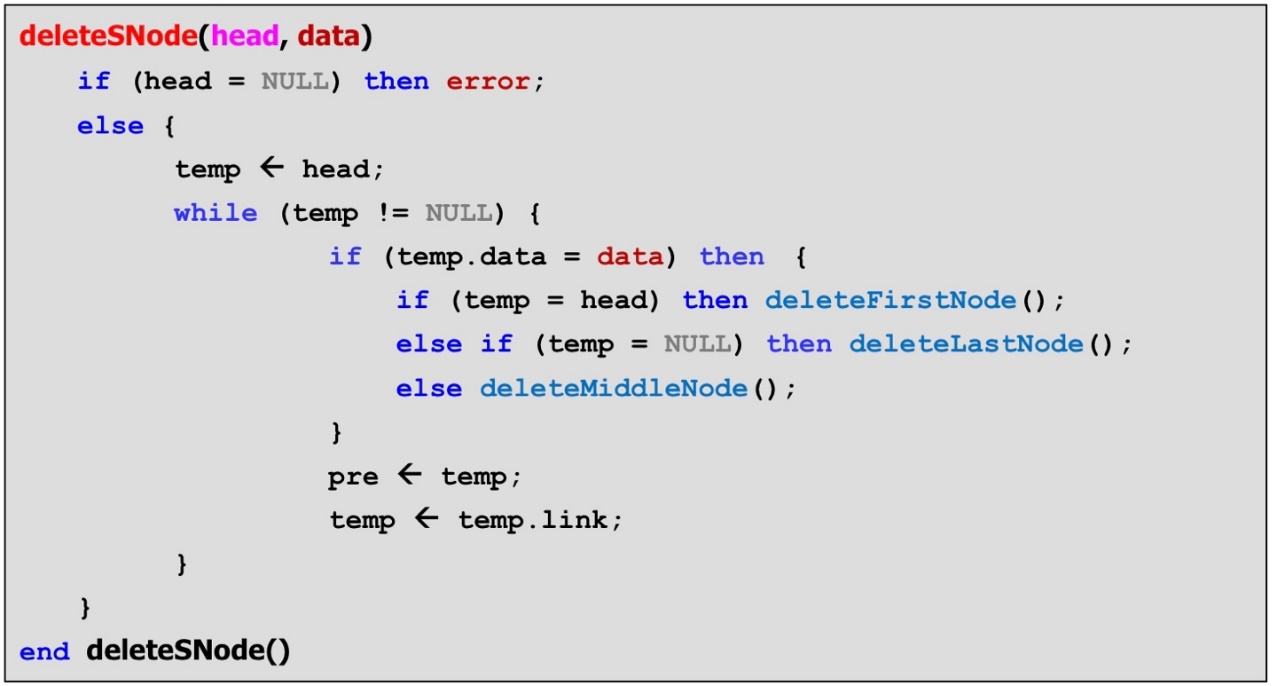


* **삽입 알고리즘(첫번째, 중간, 마지막 노드 삽입)**

****

* **삭제 알고리즘(첫번쨰, 중간, 마지막 노드 삭제)**

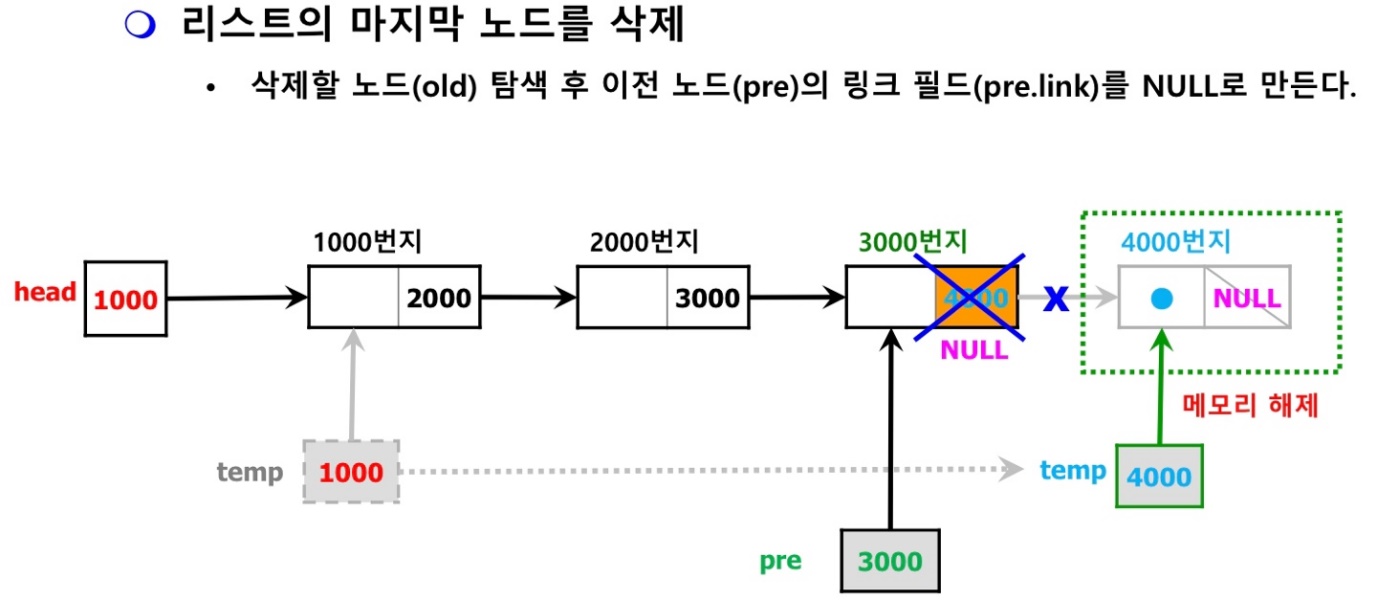
: 리스트에서 조건을 만족하는 노드 삭제 알고리즘



텍스트, 도표, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

텍스트, 도표, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

**# 코드**

/\*

단순 연결 리스트: 알고리즘 구현

파일명: SLinkedList(head).cpp

- 클래스: SNode

- 클래스: SLinkedList

생성자와 소멸자 : SLinkedList, ~SLinkedList

노드 삽입.삭제 : addRear, removeFront

노드 탐색 : frontSNode, rearSNode

노드 확인 : isEmpty, countSNode

전체 원소(노드) 출력 : printSLinkedList

\*/

#include <iostream>

using namespace std;

// 노드: 데이터 링크

class SNode {

private:

int \_\_data;

SNode\* \_\_link.

friend class SLinkedList;

public:

SNode(const int& data);

};

// SNode: 생성자와 메소드 정의

SNode::SNode(const int& data) :

\_\_data(data), \_\_link(nullptr) {}

// 단순 연결 리스트

class SLinkedList {

private:

SNode\* \_\_head; // 첫 번째 노드

// SNode\* \_\_tail; // 맨 마지막 노드

// int \_\_count; // 노드의 총 개수

public:

SLinkedList(); // 생성자

~SLinkedList(); // 소멸자: 전체 노드 삭제

void addRear(const int& e); // 삽입: 맨 마지막 노드

void removeFront(); // 삭제: 첫 번째 노드

SNode\* frontSNode() const; // 탐색: 첫 번째 노드

SNode\* rearSNode() const; // 탐색: 맨 마지막 노드

bool isEmpty() const; // 빈 리스트 여부 판단

int countSNode() const; // 탐색: 노드의 총 개수(count)

void printSLinkedList() const; // 리스트의 전체 원소(노드) 출력

};

// LinkedStack: 생성자(소멸자)와 메소드 정의

SLinkedList::SLinkedList()

: \_\_head(nullptr) { }

SLinkedList::~SLinkedList() {

// while (!isEmpty()) removeFront();

SNode\* tNode = \_\_head;

while (tNode) {

\_\_head = tNode->\_\_link;

delete tNode;

tNode = \_\_head;

}

}

// 삽입: 맨 마지막 노드

void SLinkedList::addRear(const int& e) {

SNode\* nNode = new SNode(e);

if (isEmpty()) \_\_head = nNode;

else {

SNode\* rNode = rearSNode();

rNode->\_\_link = nNode;

}

}

// 삭제: 첫 번째 노드

void SLinkedList::removeFront() {

if (isEmpty()) return;

SNode\* old = \_\_head;

\_\_head = old->\_\_link;

delete old;

}

// 탐색: 첫 번째 노드

SNode\* SLinkedList::frontSNode() const {

return \_\_head;

}

// 탐색: 맨 마지막 노드

SNode\* SLinkedList::rearSNode() const {

if (isEmpty()) return nullptr;

SNode\* rNode = \_\_head;

while (rNode->\_\_link)

rNode = rNode->\_\_link;

return rNode;

}

// 빈 리스트 여부 판단

bool SLinkedList::isEmpty() const {

return \_\_head == nullptr;

}

// 탐색: 노드의 총 개수(count)

int SLinkedList::countSNode() const {

if (isEmpty()) return 0;

int count = 0;

SNode\* rNode = \_\_head;

while (rNode->\_\_link) {

count++;

rNode = rNode->\_\_link;

}

return count;

}

// 리스트의 전체 원소(노드) 출력

void SLinkedList::printSLinkedList() const {

if (isEmpty()) {

cout << "입력된 데이터가 없습니다..." << endl;

return;

}

cout << "\n ### 입력된 데이터 ### \n" << endl;

SNode\* temp = \_\_head;

while (temp) {

cout.width(3);

cout << temp->\_\_data << " ->>";

temp = temp->\_\_link;

}

cout << " NULL" << endl;

}

int main(void)

{

int num;

SLinkedList sList = SLinkedList();

while (true) {

cout << "임의의 정수 입력(종료: 0): ";

cin >> num;

if (num == 0) break;

sList.addRear(num); // 맨 마지막 노드로 삽입

}

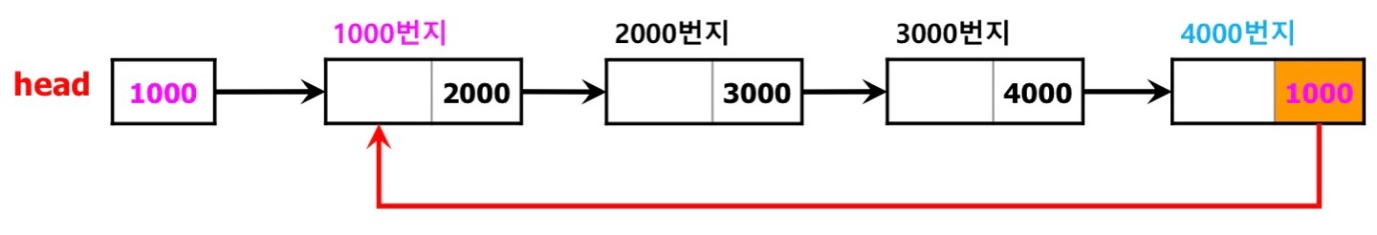
sList.printSLinkedList();

return 0; // sList.~SLinkedList();

}

**04-4. 원형 연결 리스트(Circular linked list)**

: 단순 연결 리스트에서 마지막 노드가 리스트의 첫번째 노드를 가리키게 하여 구조를 원형으로 만든 연결 리스트



**04-5. 이중 연결 리스트(Circular linked list)**

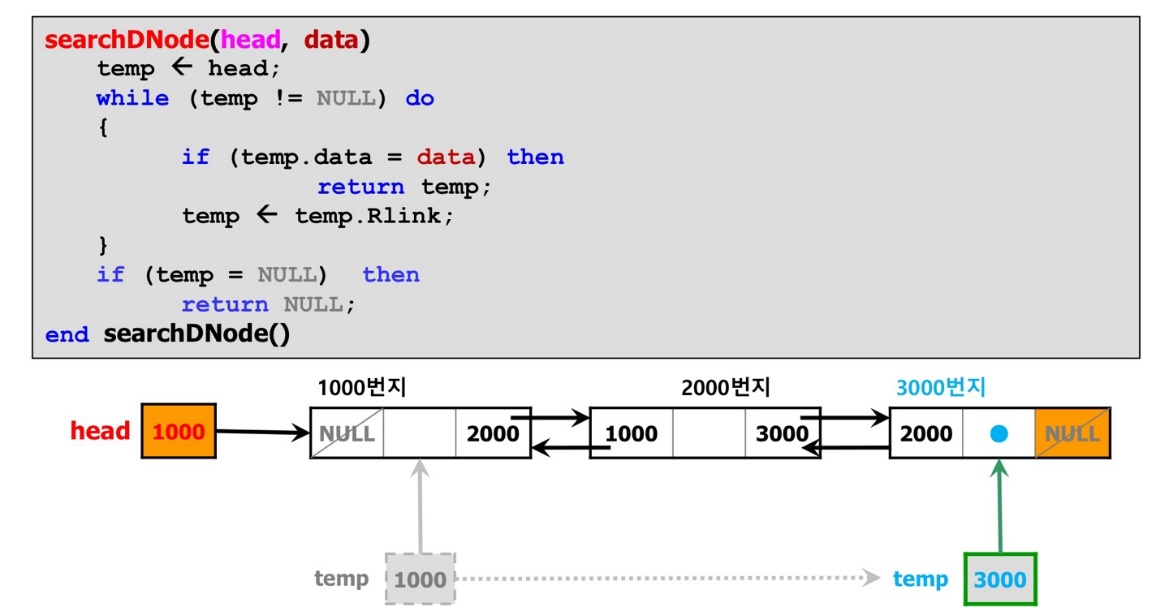
**# 단순. 원형 연결 리스트의 문제점**

: 현재 노드의 바로 이전 노드르 접근하려면 전체 리스트를 한 바퀴 순회해야 한다.

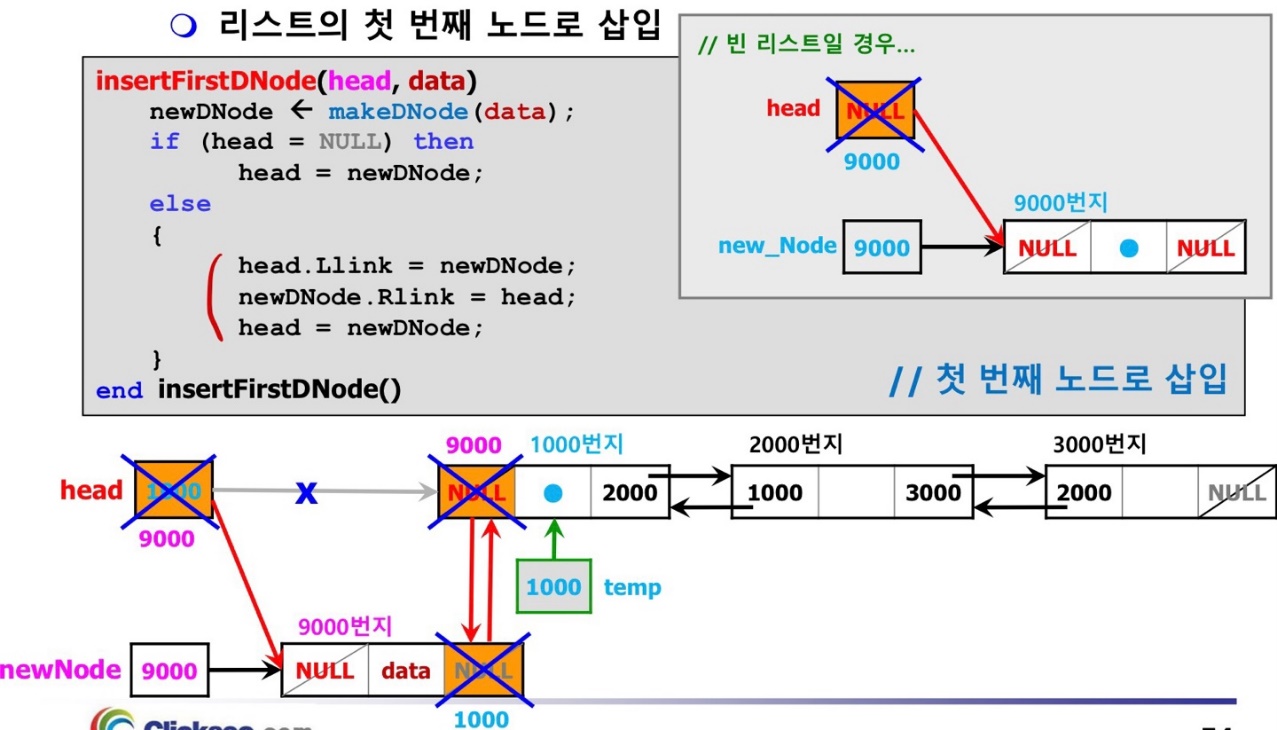
**# 알고리즘**

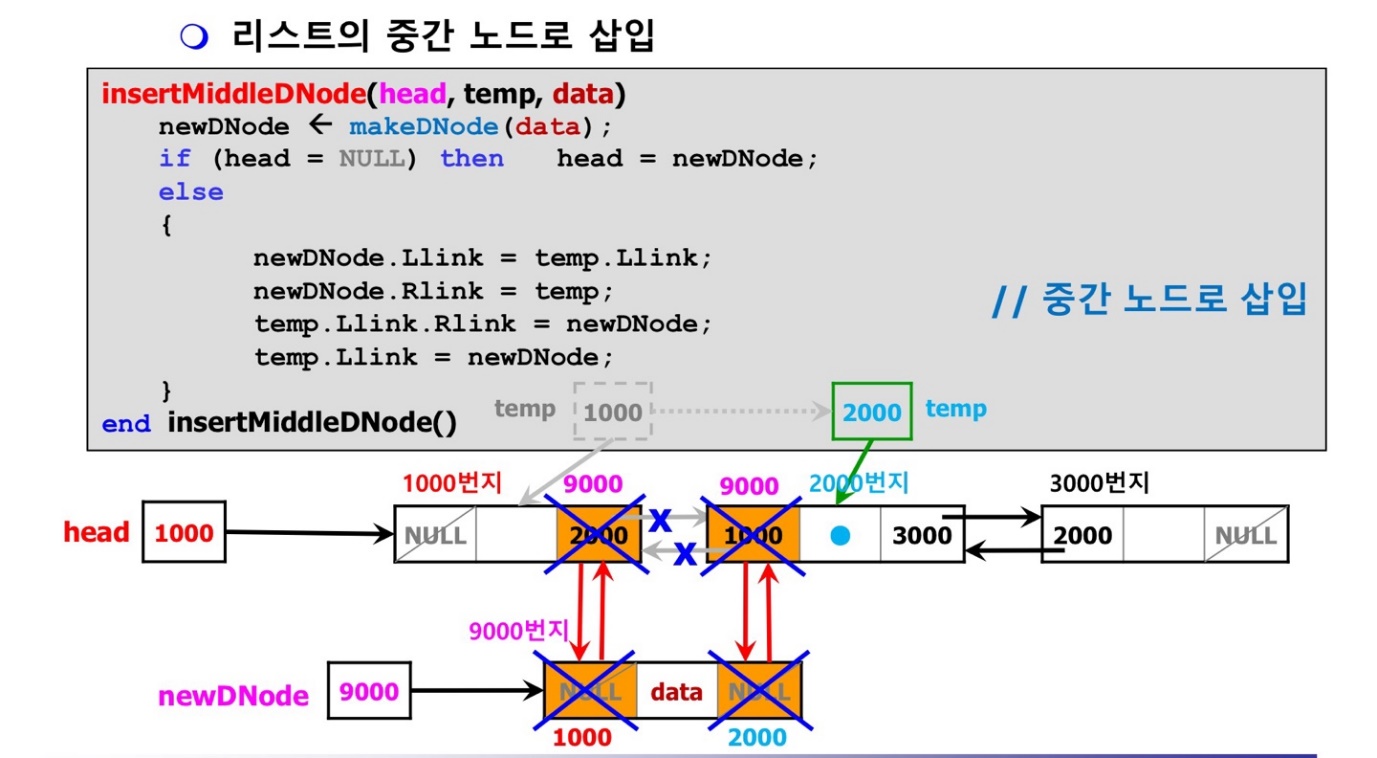
* **탐색 알고리즘**

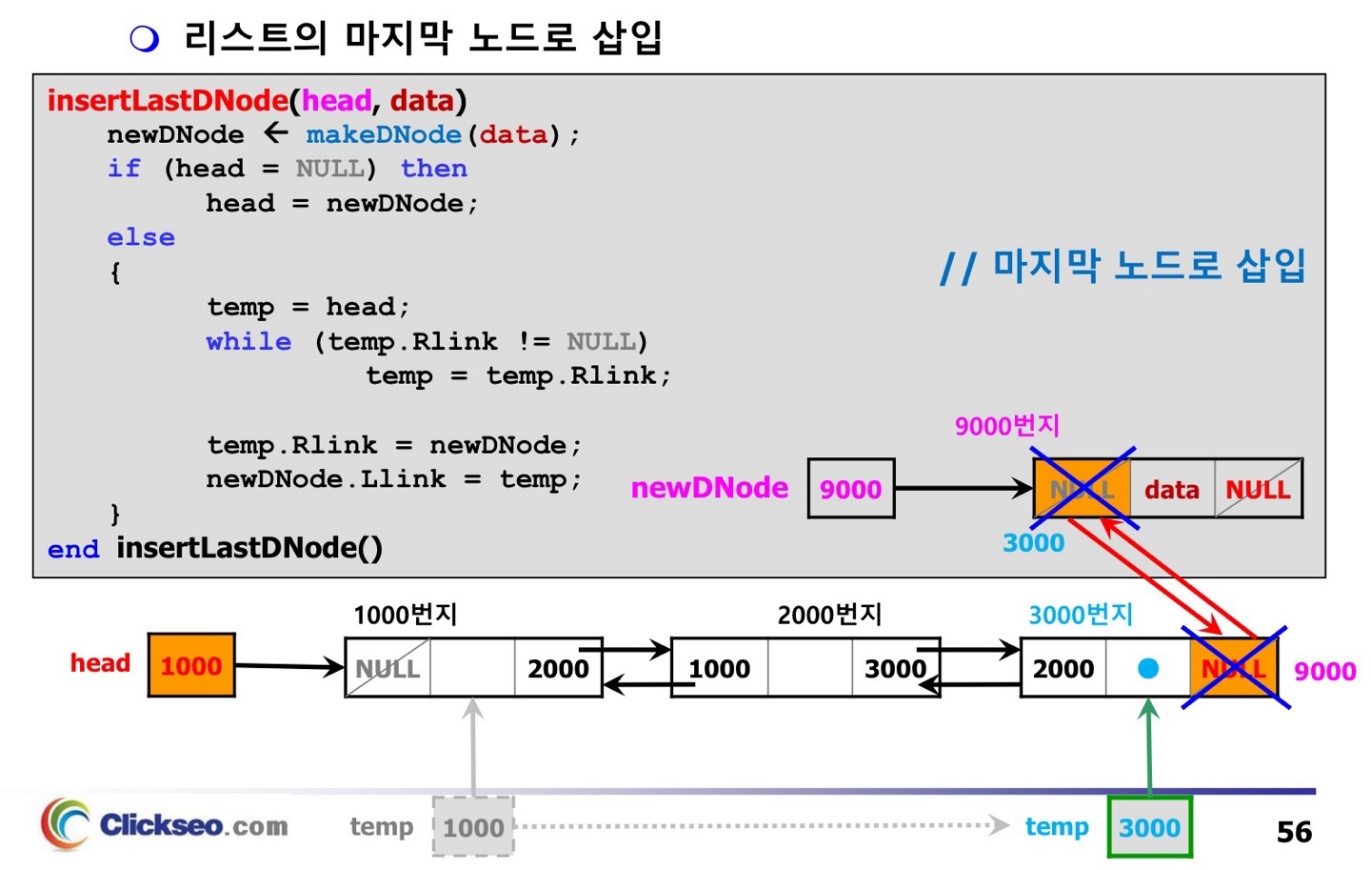
: 리스트에서 조건을 만족하는 데이터를 가진 노드 탐색 알고리즘



* **삽입 알고리즘(첫번째, 중간, 마지막 노드)**

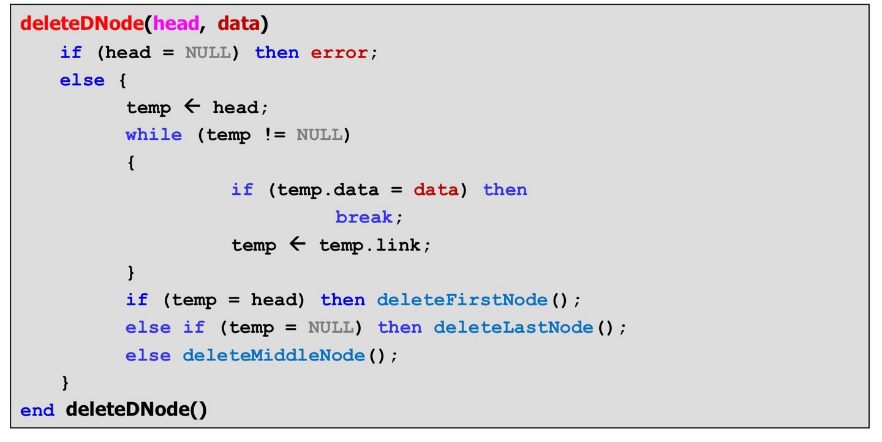
****

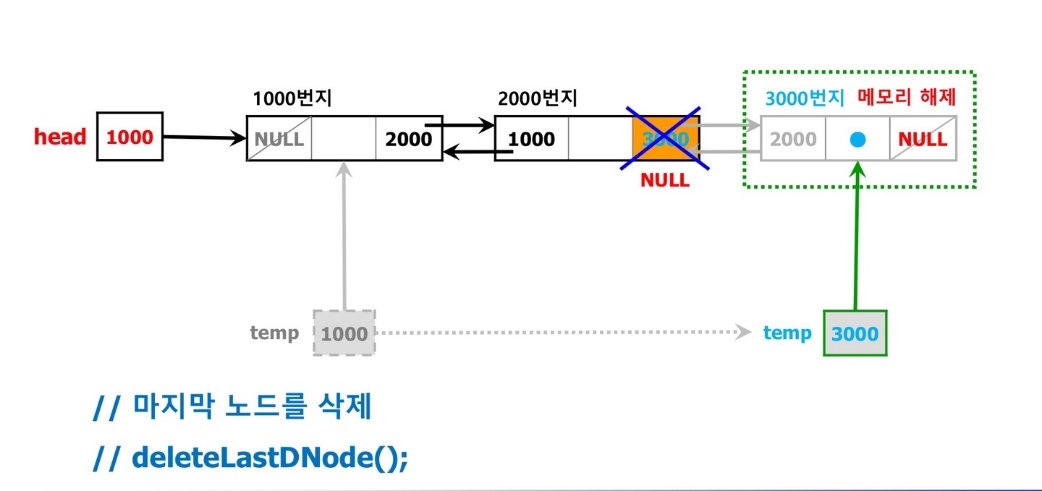
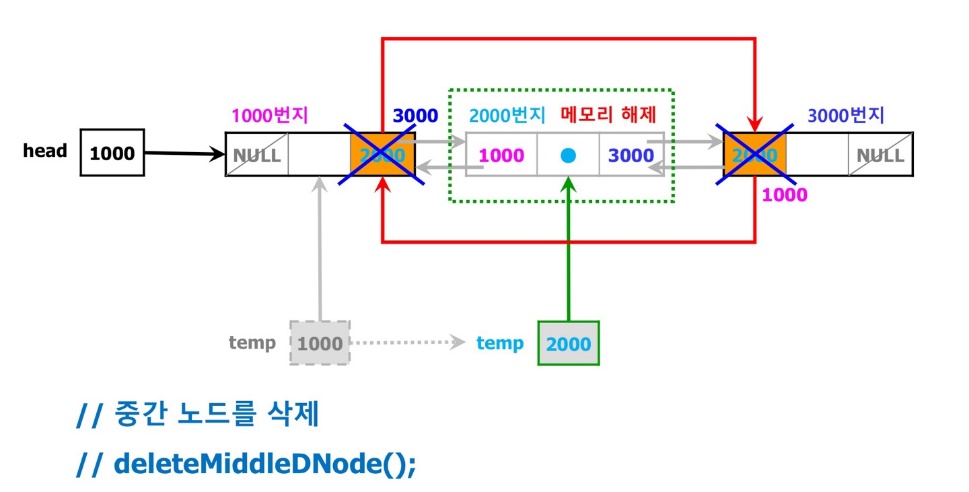
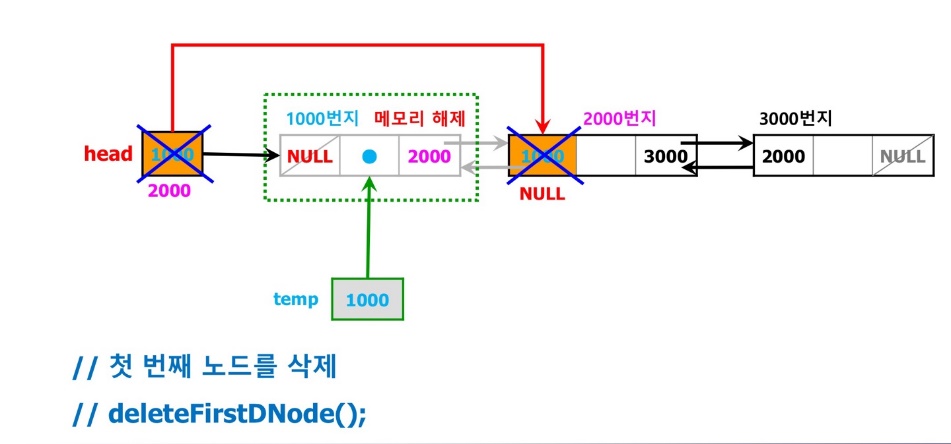
****

****

* **삭제 알고리즘(첫번째, 중간, 마지막 노드)**

: 리스트에서 조건을 만족하는 노드 삭제 알고리즘





**# 코드**

/\*

이중 연결 리스트: 알고리즘 구현

파일명: DLinkedList(head).cpp

- 클래스: DNode

- 클래스: DLinkedList

생성자와 소멸자 : DLinkedList, ~DLinkedList

노드 삽입.삭제 : addRear, removeFront

노드 탐색 : frontSNode, rearSNode

노드 확인 : isEmpty, countSNode

전체 원소(노드) 출력 : printSLinkedList

\*/

#include <iostream>

using namespace std;

// DNode class

class DNode {

private:

int \_\_data;

DNode\* \_\_Llink;

DNode\* \_\_Rlink;

friend class DLinkedList;

public:

DNode(const int& data);

};

// DNode: 생성자와 메소드 정의

DNode::DNode(const int& data) :

\_\_data(data), \_\_Llink(nullptr), \_\_Rlink(nullptr) {}

// DLinkedList class

class DLinkedList {

private:

DNode\* \_\_head; // 첫 번째 노드

// DNode\* \_\_tail; // 맨 마지막 노드

// int \_\_count; // 노드의 총 개수

public:

DLinkedList(); // 생성자

~DLinkedList(); // 소멸자: 전체 노드 삭제

void addRear(const int& e); // 삽입: 맨 마지막 노드

void removeFront(); // 삭제: 첫 번째 노드

DNode\* frontDNode() const; // 탐색: 첫 번째 노드

DNode\* rearDNode() const; // 탐색: 맨 마지막 노드

bool isEmpty() const; // 빈 리스트 여부 판단

int countDNode() const; // 탐색: 노드의 총 개수(count)

void printSLinkedList() const; // 리스트의 전체 원소(노드) 출력: 순방향

void revPrintSLinkedList() const; // 리스트의 전체 원소(노드) 출력: 역방향

};

// DLinkedList: 생성자(소멸자)와 메소드 정의

DLinkedList::DLinkedList()

: \_\_head(nullptr) { }

DLinkedList::~DLinkedList() {

// while (!isEmpty()) removeFront();

DNode\* tNode = \_\_head;

while (tNode) {

\_\_head = tNode->\_\_Rlink;

delete tNode;

tNode = \_\_head;

}

}

// 노드 삽입: 맨 마지막 노드로...

void DLinkedList::addRear(const int& e) {

DNode\* nNode = new DNode(e);

if (isEmpty()) \_\_head = nNode;

else {

DNode\* rNode = rearDNode();

rNode->\_\_Rlink = nNode;

nNode->\_\_Llink = rNode;

}

}

// 노드 삭제: 첫 번째 노드를...

void DLinkedList::removeFront() {

DNode\* old;

while (\_\_head) {

old = \_\_head;

\_\_head = old->\_\_Rlink;

if (\_\_head != nullptr)

\_\_head->\_\_Llink = nullptr;

delete old;

}

}

// 탐색: 첫 번째 노드

DNode\* DLinkedList::frontDNode() const {

return \_\_head;

}

// 탐색: 맨 마지막 노드(tail)

DNode\* DLinkedList::rearDNode() const {

if (isEmpty()) return nullptr;

DNode\* rNode = \_\_head;

while (rNode->\_\_Rlink)

rNode = rNode->\_\_Rlink;

return rNode;

}

// 빈 리스트 여부 판단

bool DLinkedList::isEmpty() const {

return \_\_head == nullptr;

}

// 탐색: 노드의 총 개수(count)

int DLinkedList::countDNode() const {

if (isEmpty()) return 0;

int count = 0;

DNode\* rNode = \_\_head;

while (rNode->\_\_Rlink)

rNode = rNode->\_\_Rlink;

return count;

}

// 리스트의 전체 원소(노드) 출력: 순방향

void DLinkedList::printSLinkedList() const {

DNode\* temp = \_\_head;

while (temp) {

cout.width(3);

cout << temp->\_\_data << " ->>";

temp = temp->\_\_Rlink;

}

cout << " NULL" << endl;

}

// 리스트의 전체 원소(노드) 출력: 역방향

void DLinkedList::revPrintSLinkedList() const {

DNode\* temp = rearDNode();

while (temp) {

cout.width(3);

cout << temp->\_\_data << " ->>";

temp = temp->\_\_Llink;

}

cout << " NULL" << endl;

}

int main(void)

{

int num;

DLinkedList dList = DLinkedList();

while (true) {

cout << "임의의 정수 입력(종료: 0): ";

cin >> num;

if (num == 0) break;

// 맨 마지막 노드로 삽입

dList.addRear(num);

}

// 전체 원소 출력

if (dList.isEmpty()) {

cout << "입력된 데이터가 없습니다..." << endl;

return 0; // dList.~DLinkedList();

}

printf("\n ### 입력된 데이터 ### \n\n");

dList.printSLinkedList();

dList.revPrintSLinkedList();

return 0; // dList.~DLinkedList();

}

**04-6. 실습보고서2\_연습문제 #06\_간단한 데이터 처리 : 단순 연결리스트 구현**

#include <iostream>

using namespace std;

// SNode class

class SNode {

private:

int \_\_data;

SNode\* \_\_link;

friend class SLinkedList;

public:

SNode(const int& data);

};

// SNode: 생성자와 메소드 정의

SNode::SNode(const int& data) :

\_\_data(data), \_\_link(nullptr) {}

// SLinkedList class

class SLinkedList {

private:

SNode\* \_\_head; // 첫 번째 노드

SNode\* \_\_tail; // 맨 마지막 노드

int \_\_count; // 노드의 총 개수

public:

SLinkedList(); // 생성자

~SLinkedList(); // 소멸자: 전체 노드 삭제

void addRear(const int& e); // 삽입: 맨 마지막 노드

void removeFront(); // 삭제: 첫 번째 노드

SNode\* frontSNode() const; // 탐색: 첫 번째 노드

SNode\* rearSNode() const; // 탐색: 맨 마지막 노드

bool isEmpty() const; // 빈 리스트 여부 판단

int countSNode() const; // 탐색: 노드의 총 개수(count)

void printSLinkedList() const; // 리스트의 전체 원소(노드) 출력

};

// LinkedStack: 생성자(소멸자)와 메소드 정의

SLinkedList::SLinkedList()

: \_\_head(nullptr), \_\_tail(nullptr), \_\_count(0){ }

SLinkedList::~SLinkedList() {

while (!isEmpty())

removeFront();

}

// 삽입: 맨 마지막 노드

void SLinkedList::addRear(const int& e) {

SNode\* nNode = new SNode(e);

if (isEmpty()) {

\_\_head = nNode;

\_\_tail = nNode;

}

else {

\_\_tail->\_\_link = nNode;

\_\_tail = nNode;

}

\_\_count++;

}

// 삭제: 첫 번째 노드

void SLinkedList::removeFront() {

if (isEmpty())

return;

SNode\* old = \_\_head;

\_\_head = old->\_\_link;

delete old;

if (\_\_head == nullptr) {

\_\_tail = nullptr;

}

\_\_count--;

}

// 탐색: 첫 번째 노드

SNode\* SLinkedList::frontSNode() const {

return \_\_head;

}

// 탐색: 맨 마지막 노드

SNode\* SLinkedList::rearSNode() const {

return \_\_tail;

}

// 빈 리스트 여부 판단

bool SLinkedList::isEmpty() const {

return \_\_head == nullptr;

}

// 탐색: 노드의 총 개수(count)

int SLinkedList::countSNode() const {

return \_\_count;

}

// 리스트의 전체 원소(노드) 출력

void SLinkedList::printSLinkedList() const {

if (isEmpty()) {

cout << "입력된 데이터가 없습니다..." << endl;

return;

}

cout << "\n ### 입력된 데이터 ### \n" << endl;

SNode\* temp = \_\_head;

while (temp) {

cout.width(3);

cout << temp->\_\_data << " ->>";

temp = temp->\_\_link;

}

cout << " NULL" << endl;

}

int main(void)

{

int num;

cout << "20215229 이채훈" << endl;

cout << endl;

SLinkedList sList = SLinkedList();

while (true) {

cout << "임의의 정수 입력(종료: 0): ";

cin >> num;

if (num == 0) break;

sList.addRear(num); // 맨 마지막 노드로 삽입

}

sList.printSLinkedList();

sList.~SLinkedList();

return 0;

}

**# 연구조사**

**함수 뒤에 const 의 의미 : 읽기 전용 함수이다.**

**리스트의 구성 함수는 생성, 조사, 출력, 삽입, 삭제로 나누어 이해할 수 있다**

생성 : makeDNode

조사 : isEmpty, countDNode, searchSNode

출력 : PrintSLinkedList

삽입 : insertRear

삭제 : deleteFront, deleteMid, deleteRear

**# 삽입 알고리즘 과정 요약**

1. 새로운 노드를 생성한다

2. head나 tail, pre\_node 에서 물려받을 값 먼저 처리

3. 이후 head, tail, pre\_node 갱신

**#삭제 알고리즘 과정 요약**

1. 살릴 정보 작업

2. 노드 삭제

\*\* 뒤에서 삭제하는 경우, 노드가 한 개일 때에 대한 예외처리가 필요하다. 그 이유는 n-1번째의 노드에 대한 정보를 가져와야하는데, 노드가 한 개인 경우엔 n-1번째 노드가 존재하지 않기 때문이다.

**#chatgpt**

SNode 클래스: 이 클래스는 연결 리스트의 노드를 나타냅니다. 각 노드는 데이터와 다음 노드를 가리키는 링크로 이루어져 있습니다.

SLinkedList 클래스: 이 클래스는 연결 리스트 자체를 나타냅니다. 이 클래스에는 연결 리스트의 첫 번째 노드와 마지막 노드를 가리키는 포인터 및 노드의 총 개수를 추적하는 변수가 포함되어 있습니다.

생성자와 소멸자: SLinkedList 클래스의 생성자는 초기에 연결 리스트를 비어 있는 상태로 설정하고, 소멸자는 모든 노드를 삭제하여 메모리를 해제합니다.

addRear 메서드: 이 메서드는 연결 리스트의 맨 뒤에 새로운 노드를 추가합니다. 빈 리스트인 경우, \_\_head와 \_\_tail이 새로운 노드를 가리키게 되며, 그렇지 않으면 \_\_tail의 \_\_link를 새로운 노드로 갱신하고 \_\_tail을 업데이트합니다.

removeFront 메서드: 이 메서드는 연결 리스트의 첫 번째 노드를 삭제합니다. 먼저 리스트가 비어 있는지 확인하고, 첫 번째 노드를 삭제한 후 \_\_head와 \_\_tail을 갱신합니다.

기타 메서드: frontSNode, rearSNode, isEmpty, countSNode 등은 연결 리스트의 상태를 확인하고 조작하기 위한 메서드들입니다.

printSLinkedList 메서드: 이 메서드는 현재 연결 리스트의 모든 원소를 출력합니다.

main 함수: 메인 함수에서는 사용자로부터 정수를 입력받아 연결 리스트의 끝에 추가하고, 리스트를 출력합니다. 마지막으로 sList.~SLinkedList()를 호출하여 메모리를 해제하고 프로그램을 종료합니다.

**04-7. 실습보고서2\_연습문제 #07\_ 간단한 데이터 처리 : 이중 연결리스트 구현**

#include <iostream>

using namespace std;

// DNode class

class DNode {

private:

int \_\_data;

DNode\* \_\_Llink;

DNode\* \_\_Rlink;

friend class DLinkedList;

public:

DNode(const int& data);

};

// DNode: 생성자와 메소드 정의

DNode::DNode(const int& data) :

\_\_data(data), \_\_Llink(nullptr), \_\_Rlink(nullptr) {}

// DLinkedList class

class DLinkedList {

private:

DNode\* \_\_head; // 첫 번째 노드

DNode\* \_\_tail; // 맨 마지막 노드

int \_\_count; // 노드의 총 개수

public:

DLinkedList(); // 생성자

~DLinkedList(); // 소멸자: 전체 노드 삭제

void addRear(const int& e); // 삽입: 맨 마지막 노드

void removeFront(); // 삭제: 첫 번째 노드

DNode\* frontDNode() const; // 탐색: 첫 번째 노드

DNode\* rearDNode() const; // 탐색: 맨 마지막 노드

bool isEmpty() const; // 빈 리스트 여부 판단

int countDNode() const; // 탐색: 노드의 총 개수(count)

void printSLinkedList() const; // 리스트의 전체 원소(노드) 출력: 순방향

void revPrintSLinkedList() const; // 리스트의 전체 원소(노드) 출력: 역방향

};

// DLinkedList: 생성자(소멸자)와 메소드 정의

DLinkedList::DLinkedList()

: \_\_head(nullptr), \_\_tail(nullptr), \_\_count(0) { }

DLinkedList::~DLinkedList() {

while (!isEmpty()) removeFront();

}

// 노드 삽입: 맨 마지막 노드로...

void DLinkedList::addRear(const int& e) {

DNode\* nNode = new DNode(e);

if (isEmpty()) {

\_\_head = nNode;

\_\_tail = nNode;

}

else {

\_\_tail->\_\_Rlink = nNode;

nNode->\_\_Llink = \_\_tail;

\_\_tail = nNode;

}

\_\_count++;

}

// 노드 삭제: 첫 번째 노드를...

void DLinkedList::removeFront() {

if (!isEmpty()) {

DNode\* old = \_\_head;

\_\_head = old->\_\_Rlink;

if (\_\_head != nullptr)

\_\_head->\_\_Llink = nullptr;

delete old;

\_\_count--.

if (isEmpty()) {

\_\_tail = nullptr;

}

}

}

**교수님 코드**

// 노드 삭제: 첫 번째 노드를...

void DLinkedList::removeFront() {

if (!isEmpty()) return;

DNode\* old = \_\_head;

\_\_head = old->\_\_Rlink;

if (\_\_head != nullptr)

old->\_\_Llink = nullptr;

else

\_\_tail = nullptr;

delete old;

\_\_count--;

}

// 탐색: 첫 번째 노드

DNode\* DLinkedList::frontDNode() const {

return \_\_head;

}

// 탐색: 맨 마지막 노드(tail)

DNode\* DLinkedList::rearDNode() const {

return \_\_tail;

}

// 빈 리스트 여부 판단

bool DLinkedList::isEmpty() const {

return \_\_head == nullptr;

}

// 탐색: 노드의 총 개수(count)

int DLinkedList::countDNode() const {

return \_\_count;

}

// 리스트의 전체 원소(노드) 출력: 순방향

void DLinkedList::printSLinkedList() const {

DNode\* temp = \_\_head;

while (temp) {

cout.width(3);

cout << temp->\_\_data << " ->>";

temp = temp->\_\_Rlink;

}

cout << " NULL" << endl;

}

// 리스트의 전체 원소(노드) 출력: 역방향

void DLinkedList::revPrintSLinkedList() const {

DNode\* temp = \_\_tail;

while (temp) {

cout.width(3);

cout << temp->\_\_data << " ->>";

temp = temp->\_\_Llink;

}

cout << " NULL" << endl;

}

int main(void) {

int num;

cout << "20215229 이채훈" << endl;

cout << endl;

DLinkedList dList = DLinkedList();

while (true) {

cout << "임의의 정수 입력(종료: 0): ";

cin >> num;

if (num == 0) break;

// 맨 마지막 노드로 삽입

dList.addRear(num);

}

// 전체 원소 출력

if (dList.isEmpty()) {

cout << "입력된 데이터가 없습니다..." << endl;

}

else {

printf("\n ### 입력된 데이터 ### \n\n");

dList.printSLinkedList();

dList.revPrintSLinkedList();

}

dList.~DLinkedList();

return 0;

}

**# 연구조사**

**exit()**

프로그램을 종료시키는 함수이다. exit(0)~exit(255)까지 가능하며,

exit(0)일 경우 프로그램이 정상적으로 종료되었음을 의미, exit(1)~exit(255)는 에러 발생으로 인해 프로그램이 종료되었음을 의미한다.

**리스트의 구성 함수는 생성, 조사, 출력, 삽입, 삭제로 나누어 이해할 수 있다**

생성 : makeDNode

조사 : isEmpty, countDNode, searchSNode

출력 : PrintSLinkedList

삽입 : insertRear

삭제 : deleteFront, deleteMid, deleteRear

**//생성 알고리즘**

1. 노드 동작할당 받기

2. data,Llink,Rlink 초기화

**//삽입 알고리즘**

1. 노드 생성 (n+1번째 노드)

2. n번째 노드의 Rlink = n+1번째 노드

3. n+1번째 노드의 Llink = n번째 노드

4. tail = n+1번째 노드

기존 tail는 n번째 노드에 대한 정보를 가지고 있으므로, n번째 노드 정보를 사용하는 모든 작업을 마친 뒤, tail을 n+1번째 노드로 바꾼다.

**//삭제 알고리즘**

**deleteFront**

head와 2번째 노드를 이어준 후, 1번째 노드를 삭제한다.

1. 빈 리스트 확인

2. old = 1번째 노드

3. 2번째 노드의 Llink = NULL

4. head = 2번째 노드

5. old 삭제

기존 head는 2번째 노드에 대한 정보를 알게 해준다. 따라서 반환하기 전에 2번째 노드에 대한 정보를 모두 사용한 후 반환한다.

**deleteMid**

k-1번째 노드와 k+1번째 노드를 이어준 후 , k번째 노드를 삭제한다.

1. 빈 리스트 확인

2. old = k번째 노드

3. k-1번째 노드의 Rlink = k+1번째 노드

4. k+1번째 노드의 Llink = k-1번째 노드

5. old 삭제

**deleteRear**

n-1번째 노드와 tail을 이어준 후 , n번째 노드를 삭제한다.

1. 빈 리스트 확인

2. old = n번째 노드

3. 노드가 한 개일 경우 예외처리

3. n-1번째 노드의 Rlink = NULL

4. tail = n-1번째 노드

5. old 삭제

**#chatgpt**

이 코드는 C++로 작성된 이중 연결 리스트(이중 링크드 리스트)를 구현하는 프로그램입니다. 이중 연결 리스트는 각 노드가 이전 노드와 다음 노드에 대한 포인터를 갖는 자료구조로, 원하는 위치에서 삽입과 삭제를 빠르게 수행할 수 있는 장점이 있습니다. 이 코드는 다음과 같은 주요 구성 요소와 작업을 수행합니다.

**DNode 클래스**

: DNode 클래스는 이중 연결 리스트의 각 노드를 나타내며, 노드에 저장될 데이터(\_\_data), 왼쪽 링크(\_\_Llink), 오른쪽 링크(\_\_Rlink)를 포함합니다.

DNode 클래스는 DLinkedList 클래스와 친구 클래스로 지정되어 있어서 DLinkedList 클래스에서 DNode 클래스의 private 멤버에 접근할 수 있습니다.

생성자 DNode::DNode는 노드를 초기화합니다.

**DLinkedList 클래스**

: DLinkedList 클래스는 이중 연결 리스트를 나타내며, 첫 번째 노드(\_\_head), 맨 마지막 노드(\_\_tail), 그리고 노드의 총 개수(\_\_count)를 추적합니다.

생성자 DLinkedList::DLinkedList는 빈 연결 리스트를 초기화합니다.

소멸자 DLinkedList::~DLinkedList는 모든 노드를 삭제하여 메모리 누수를 방지합니다.

addRear 메서드는 주어진 데이터를 이중 연결 리스트의 맨 뒤에 삽입합니다.

removeFront 메서드는 첫 번째 노드를 삭제합니다.

frontDNode 메서드는 첫 번째 노드를 반환합니다.

rearDNode 메서드는 맨 마지막 노드를 반환합니다.

isEmpty 메서드는 연결 리스트가 비어 있는지 확인합니다.

countDNode 메서드는 노드의 총 개수를 반환합니다.

printSLinkedList 메서드는 연결 리스트의 모든 노드를 순방향으로 출력합니다.

revPrintSLinkedList 메서드는 연결 리스트의 모든 노드를 역방향으로 출력합니다.

**main 함수**

: 프로그램의 진입점입니다.

사용자로부터 임의의 정수를 입력받아 이중 연결 리스트의 맨 뒤에 삽입합니다. 입력 값이 0이면 입력을 종료합니다.

입력이 완료되면 입력된 데이터를 순방향과 역방향으로 출력합니다.

마지막으로, DLinkedList 클래스의 소멸자를 호출하여 동적으로 할당된 메모리를 해제합니다.

**04-8. 중간고사 족보**

**Test3**

#include <iostream>

using namespace std;

class SNode {

int data;

SNode\* link;

friend class SLinkedList;

};

class SLinkedList {

SNode\* head;

SNode\* tail;

int count;

public:

SLinkedList();

~SLinkedList();

bool isEmpty() const;

int countSNode() const;

SNode\* frontSNode() const; //첫 노드 탐색

SNode\* makeSNode(const int& e);

void addRear(const int& e);

void removeFront();

void printSLinkedList();

};

SLinkedList::SLinkedList() {

head = NULL;

tail = NULL;

}

SLinkedList::~SLinkedList() {

while (!isEmpty()) removeFront();

}

bool SLinkedList::isEmpty() const {

return head == NULL;

}

int SLinkedList::countSNode() const{

SNode\* temp = head;

int count = 0;

while (temp != NULL) {

temp = temp->link;

count++;

}//while의 결과 : temp = NULL;

return count;

}

SNode\* SLinkedList::frontSNode() const {

if (isEmpty()) {

cout << " 빈 리스트입니다." << endl;

return NULL;

}

return head;

}

SNode\* SLinkedList::makeSNode(const int& e) {

SNode\* nNode = new SNode; //SNode를 저장할 메모리 공간 부여

nNode->data = e;

nNode->link = NULL;

return nNode;

}

void SLinkedList::addRear(const int& e) {

SNode\* NewNode = makeSNode(e);

if (isEmpty()) {

head = NewNode;

return;

}

//탐색

SNode\* temp = head;

while (temp->link != NULL) {

temp = temp->link;

}//while 결과 : 마지막 노드

temp->link = NewNode;

}

void SLinkedList::removeFront() {

SNode\* old = head;

head = old->link;

delete old;

}

void SLinkedList::printSLinkedList() {

SNode\* temp = head;

while (temp!= NULL) {

cout << temp->data << " ->> ";

temp = temp->link;

}//temp = NULL

cout << "NULL" << endl;

}

int main() {

int x;

SLinkedList s;

while (true) {

cout << "임의의 정수 입력(종료 : 0): ";

cin >> x;

if (x == 0) break;

else {

s.addRear(x);

}

}

cout << "### 입력된 데이터(총 : " << s.countSNode() << " ) ###" << endl;

s.printSLinkedList();

return 0;

}

**04-9. 송\_단순연결리스트**

**# 코드**

#include <iostream>

using namespace std;

class SNode {

private:

int data;

SNode\* link;

friend class SLinkedList;

};

class SLinkedList {

private:

SNode\* head;

SNode\* tail;

int count;

public:

SLinkedList();

~SLinkedList();

//생성

SNode\* makeSNode(const int& e);

//조사

bool isEmpty() const;

int countSNode();

int searchSNode(const int& e);

//출력

void PrintSLinkedList() const;

//삽입

void insertFront(const int& e);

void insertMid(const int& e, SNode\* preSNode);

void insertRear(const int& e);

//삭제

void removeFront();

void removeMid(SNode\* preSNode);

void removeRear();

};

inline void error(const char\* message) {

cout << message;

exit(100); //exit :

}

//생성자 & 소멸자

SLinkedList::SLinkedList() {

head = NULL;

tail = NULL;

count = 0;

}

SLinkedList::~SLinkedList() {

while (!isEmpty()) removeRear();

}

//생성

SNode\* SLinkedList::makeSNode(const int& e) {

SNode\* s = new SNode;

s->data = e;

s->link = NULL;

return s;

}

//조사

bool SLinkedList::isEmpty() const {

return head == NULL;

}

int SLinkedList::countSNode() {

return count;

}

int SLinkedList::searchSNode(const int& e) {

SNode\* temp = head;

count = 0;//count 0으로 초기화해두고 시작

while (temp != NULL) { //마지막 노드 while문 실행

count++;

if (temp->data == e) return count;

else temp = temp->link;

}//if-else 아니여도 똑같음

//while의 결과 temp=null

if (temp == NULL) return 0;

}

//출력

void SLinkedList::PrintSLinkedList() const {

SNode\* temp = head;

while (temp != NULL) {

cout << temp->data << " --> ";

temp = temp->link;

}

if (temp == NULL) cout << "NULL";

}

//삽입

//앞부터 삽입

void SLinkedList::insertFront(const int& e) {

SNode\* newSNode = makeSNode(e); //새로운 노드 생성

//if(isEmpty()) head=newSNode else{}; //없어도 됨

newSNode->link = head;//head값 물려받음

head = newSNode;

}

//중간 삽입

void SLinkedList::insertMid(const int& e, SNode\* preSNode) {

SNode\* newSNode = makeSNode(e);//새로운 노드 생성

if (isEmpty()) {

head = newSNode;

tail = newSNode;

}

else {

newSNode->link = preSNode->link;

preSNode->link = newSNode;

}

}

//뒤부터 삽입

void SLinkedList::insertRear(const int& e) {

SNode\* newSNode = makeSNode(e);//새로운 노드 생성

if (isEmpty()) {

head = newSNode;

tail = newSNode;

}

else {

tail->link = newSNode;

tail = newSNode;

}

count++;

}

//삭제

//앞에서 삭제

void SLinkedList::removeFront() {

if (isEmpty()) return;

SNode\* old = head;

head = old->link;

delete old;

count--;

}

//중간에서 삭제

void SLinkedList::removeMid(SNode\* preSNode) {

if (isEmpty()) return;

SNode\* old = preSNode->link;

preSNode->link = old->link;

delete old;

count--;

}

//뒤에서 삭제

void SLinkedList::removeRear() {

SNode\* temp = head;

if (isEmpty()) {

cout << "빈 리스트입니다 " << endl;

return;

}

if (temp == tail) {//노드 하나일 때

delete head;

head = NULL;

tail = NULL;

count--;

return;

}

SNode\* old = tail;

count = 1;

//이전 노드 탐색

while (temp->link != tail) {

temp = temp->link;

count++;

}//while문 결과 : n-1 노드

temp->link = NULL;

tail = temp;

delete old;

count--;

}

//메인함수

int main() {

int num;

SLinkedList s = SLinkedList();

while (true) {

cout << " 임의의 정수 입력(종료 : 0): ";

cin >> num;

if (num == 0) break;

//맨 마지막 노드로 삽입

s.insertRear(num);

}

// 전체 원소 출력

if (s.isEmpty()) {

cout << "입력된 데이터가 없습니다..." << endl;

return 0;

}

printf("\n ### 입력된 데이터 ### \n\n ");

s.PrintSLinkedList();

printf("\n ### 데이터 갯수 ### \n\n ");

cout << "총 데이터 갯수 : " << s.countSNode() << endl;

s.~SLinkedList();

return 0;

}

**# 연구조사**

함수 뒤에 const 의 의미 : 읽기 전용 함수이다.

리스트의 구성 함수는 생성, 조사, 출력, 삽입, 삭제로 나누어 이해할 수 있다

생성 : makeDNode

조사 : isEmpty, countDNode, searchSNode

출력 : PrintSLinkedList

삽입 : insertRear

삭제 : deleteFront, deleteMid, deleteRear

//삽입 알고리즘 과정 요약

1. 새로운 노드를 생성한다

2. head나 tail, pre\_node 에서 물려받을 값 먼저 처리

3. 이후 head,tail,pre\_node 갱신

//삭제 알고리즘 과정 요약

1. 살릴 정보 작업

2. 노드 삭제

\*\* 뒤에서 삭제하는 경우, 노드가 한 개일 때에 대한 예외처리가 필요하다. 그 이유는 n-1번째의 노드에 대한 정보를 가져와야하는데, 노드가 한 개인 경우엔 n-1번째 노드가 존재하지 않기 때문이다.

**04-10. 송\_이중 연결 리스트**

**# 코드**

#include <iostream>

using namespace std;

inline void error(const char\* message) {

cout << message << endl;

exit(100);

}

using namespace std;

class DNode {

int data;

DNode\* Llink;

DNode\* Rlink;

friend class DLinkedList;

};

class DLinkedList {

DNode\* head;

DNode\* tail;

int count;

public:

//생성자, 소멸자

DLinkedList();

~DLinkedList();

//생성

DNode\* makeDNode(const int& e);

//조사

bool isEmpty() const;

int countDNode();

//삽입

void insertRear(const int& e);

//삭제

void deleteFront();

void deleteMid(DNode\* pre\_DNode);

void deleteRear();

//출력

void printBy\_Rlink() const;

void printBy\_Llink() const;

};

//생성자 & 소멸자

DLinkedList::DLinkedList() {

head = NULL;

tail = NULL;

count = 0;

}

DLinkedList::~DLinkedList() {

while (!isEmpty()) deleteRear();

//while(!isEmpty()) deleteFront();

if (isEmpty()) cout << "NULL";

exit(0);

}

//생성

DNode\* DLinkedList::makeDNode(const int& e) {

DNode\* nNode = new DNode;

nNode->data = e;

nNode->Llink = NULL;

nNode->Rlink = NULL;

return nNode;

}

//조사

bool DLinkedList::isEmpty() const {

return head == NULL;

}

int DLinkedList::countDNode() {

return count;

}

//삽입

void DLinkedList::insertRear(const int& e) {

DNode\* nNode = makeDNode(e); //노드 생성

if (isEmpty()) { //빈 노드 확인

head = nNode;

tail = nNode;

}

else {

tail->Rlink = nNode;

nNode->Llink = tail;

tail = nNode;

}

count++;

}

//삭제

void DLinkedList::deleteFront() {

if (isEmpty()) error("빈 리스트입니다.");

DNode\* old = head;

old->Llink = NULL;

head = old->Rlink;

delete old;

count--;

}

void DLinkedList::deleteMid(DNode\* pre\_DNode) {

if (isEmpty()) error("빈 리스트입니다.");

DNode\* old = pre\_DNode->Rlink;//삭제할 노드

pre\_DNode->Rlink = old->Rlink;

old->Rlink->Llink = pre\_DNode;

delete old;

count--;

}

void DLinkedList::deleteRear() {

if (isEmpty()) error("빈 리스트입니다.");

DNode\* old = tail; //삭제할 노드

//노드 하나 남은 경우

if (head == tail) {

head = NULL;

tail = NULL;

}

else {

// n-1 노드 Rlink -> NULL로

// tail -> n-1로

old->Llink->Rlink = NULL;

tail = old->Llink;

}

delete old;

count--;

}

//출력

void DLinkedList::printBy\_Rlink() const {

DNode\* temp = head;

while (temp) {

cout << temp->data << " ->> ";

temp = temp->Rlink;

}//마지막 노드도 while문 실행

cout << "NULL" << endl;

}

void DLinkedList::printBy\_Llink() const {

DNode\* temp = tail;

while (temp) {

cout << temp->data << " ->> ";

temp = temp->Llink;

}

cout << "NULL" << endl;

}

//메인함수

int main() {

int num;

DLinkedList d = DLinkedList();

while (true) {

cout << "임의의 정수 입력(종료 : 0) : ";

cin >> num;

if (num == 0) break;

d.insertRear(num);

}

if (d.isEmpty()) {

cout << "입력된 데이터가 없습니다..." << endl;

return 0;

}

printf("%c", '\n');

cout << "### 입력된 데이터 ### " << endl;

printf("%c", '\n');

d.printBy\_Rlink();

d.printBy\_Llink();

d.~DLinkedList();

return 0;

}

**# 연구조사**

exit()

프로그램을 종료시키는 함수이다. exit(0)~exit(255)까지 가능하며,

exit(0)일 경우 프로그램이 정상적으로 종료되었음을 의미, exit(1)~exit(255)는 에러 발생으로 인해 프로그램이 종료되었음을 의미한다.

리스트의 구성 함수는 생성, 조사, 출력, 삽입, 삭제로 나누어 이해할 수 있다

생성 : makeDNode

조사 : isEmpty, countDNode, searchSNode

출력 : PrintSLinkedList

삽입 : insertRear

삭제 : deleteFront, deleteMid, deleteRear

//생성 알고리즘

1. 노드 동작할당 받기

2. data,Llink,Rlink 초기화

//삽입 알고리즘

1. 노드 생성 (n+1번째 노드)

2. n번째 노드의 Rlink = n+1번째 노드

3. n+1번째 노드의 Llink = n번째 노드

4. tail = n+1번째 노드

기존 tail는 n번째 노드에 대한 정보를 가지고 있으므로, n번째 노드 정보를 사용하는 모든 작업을 마친 뒤, tail을 n+1번째 노드로 바꾼다.

//삭제 알고리즘

deleteFront

head와 2번째 노드를 이어준 후, 1번째 노드를 삭제한다.

1. 빈 리스트 확인

2. old = 1번째 노드

3. 2번째 노드의 Llink = NULL

4. head = 2번째 노드

5. old 삭제

기존 head는 2번째 노드에 대한 정보를 알게 해준다. 따라서 반환하기 전에 2번째 노드에 대한 정보를 모두 사용한 후 반환한다.

deleteMid

k-1번째 노드와 k+1번째 노드를 이어준 후 , k번째 노드를 삭제한다.

1. 빈 리스트 확인

2. old = k번째 노드

3. k-1번째 노드의 Rlink = k+1번째 노드

4. k+1번째 노드의 Llink = k-1번째 노드

5. old 삭제

deleteRear

n-1번째 노드와 tail을 이어준 후 , n번째 노드를 삭제한다.

1. 빈 리스트 확인

2. old = n번째 노드

3. 노드가 한 개일 경우 예외처리

3. n-1번째 노드의 Rlink = NULL

4. tail = n-1번째 노드

5. old 삭제