**기타등등**

/\*

수 정렬하기--sort

\*/

vector<int> arr; //int 원소를 갖는 벡터 배열 생성

arr.push\_back(temp);// 벡터 arr원소에 값 추가

sort(arr.begin(), arr.end());//퀵 정렬기반으로 구현

/\*

PRINT 함수

\*/

#include <iostream>

using namespace std;

void PRINT(int\* pArr, int num) {

for (int i = 0; i < num; i++) {

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

/\*

난수생성

\*/

#include <random>

#define arrMAXSIZE 15;

int main(void){

int arr[arrMAXSIZE] = { 0 };

random\_device rd;

mt19937 gen(rd);

uniform\_int\_distribution<int> dis(0, 99);

for (int i = 0; i < 15; i++)

\*(arr + i) = dis(gen);

}

/\*

교환 작업

\*/

void SWAP(int\* pa, int\* pb) {

int temp;

temp = \*pa;

\*pa = \*pb;

\*pb = temp;

}

/\*

십진수, 이진수 출력

\*/

int validateBinary(int num) {

while (num > 0) {

if (num % 10 != 0 && num % 10 != 1)

return FALSE;//십진수

num /= 10;

}

return TRUE;//이진수

}

int binaryToDecimal(int num) {

int res = 0, pow = 1;

for (int i = 0; num > 0; i++) {

res += (num % 10) \* pow;

num /= 10;

pow \*= 2;

}

return res;

}

/\*

학생 성적 처리 예제 : 동적 메모리 할당과 포인터 참조

\*/

void stuGrade((const double\* pArr, int num, double& rSum, double& rAve, double& rMax);

double\* pArr = nullptr;

pArr = new double[stuSize];

for (int i = 0; i < stuSize; i++) {

cin >> \*(pArr + i);

}

delete[] pArr;

/\*

로또 복권 : 동적 메모리 할당과 2차원 배열 자동 생성

\*/

//동적 메모리 할당 : 2차원 배열

int\*\* table = new int\* [num + 1];

for (i = 0; i < num; i++) {

\*(table + i) = new int[COL];

}

\*(table + i) = nullptr;

//로또 복권 번호 생성

bool state;

int temp;

srand(time(nullptr));

for (int i = 0; i < num; i++) {

for (int j = 0; j < COL; j++) {

state = TRUE;

temp = rand() % 45 + 1;

for (int k = 0; k <= j; k++) {

if (table[i][k] == temp) {

state = false;

break;

}

}

if (state) table[i][j] = temp;

else j--;

}

}

/\*

은행 계좌의 인증과 출금 : 응용

\*/

#include <iostream>

#include <string>

using namespace std;

class Account {

const string \_\_name;

const string \_\_account;

string \_\_pass;

int \_\_balance;

public:

Account(string name, string account, string pass = "11111", int balance = 1);

void setPass(string);

void setBalance(int);

string getName(void)const;

string getAccount(void) const;

string getPass(void)const;

int getBalance(void)const;

void showAccount(void)const;

};

Account::Account(string name, string account, string pass, int balance)

: \_\_name(name), \_\_account(account) {//멤버 초기화 리스트

\_\_pass = pass;

\_\_balance = balance;

}

int main(void) {

Account user("홍길동", "111-2222-3333", "12345", 1000000);

string userAccount, userPass;

user.showAccount;

try {

cout << "\n###### 은행 계좌 인증 #####" << endl;

cout << "계좌번호 입력: "; cin >> userAccount;

cout << "비밀번호 입력: "; cin >> userPass;

if (user.getAccount() != userAccount || user.getPass() != userPass)

throw Account("", userAccount, userPass, 0);

int money;

cout << "출금액 입력: "; cin >> money;

if (user.getBalance() < money)

throw money;

int num = a.getBalance() - money;

user.getBalance(num);

cout << user.getName() << "님의 현재 잔액은: " << user.getBalance() << "입니다." << endl;

}

catch (int money) {//예외처리 : 정수형

cout << "현재 잔액: " << a.getBalance() << endl;

cout << "희망 출금액: (" << money << ")이 너무 많네요." << endl;

}

catch (Account& exception) {//예외처리 : 객체

cout << "\n다음 입력을 다시 한 번 확인하세요!!!" << endl;

exception.showAccount();

}

return 0;

}

/\*

Point Class

\*/

Point operator+(const Point& p1, const Point& p2) {

Point temp(p1.x + p2.x, p1.y + p2.y);

return temp;

}

Point operator-(const Point& p1, const Point& p2) {

Point temp(p1.x - p2.x, p1.y - p2.y);

return temp;

}

/\*

Albamon Class

\*/

class Albamon : private Student, private Worker {

// Person 클래스를 상속 받은 Worker, Student클래스를 상속받은 Albamon 클래스 정의

public:

Albamon(const string& name, int age, int admissionYear, int graduationYear, int workingTime)

: Student(name, age, admissionYear, graduationYear), Worker(name, age, workingTime) {}

**피보나치**/\*

피보나치 수열 #1

재귀적 용법

\*/

int Fibo1(int num){

//재귀 함수 탈출 조건

if (num == 1 || num == 2)

return -1;

return Fibo1(num - 1) + Fibo1(num - 2);

}

/\*

피보나치 수열 #2

비재귀적 용법

\*/

int Fibo2(int num) {

if (num == 1 || num == 2)

return 1;

int first, second, res = 0;

for (int i = 3; i <= num; i++) {

res = first + second;

first = second;

second = res;

}

return res;

}

/\*

피보나치 수열 #3

동적 프로그래밍

\*/

int Fibo3(int num) {

if (num == 1 || num == 2)

return 1;

//동적 메모리 할당

int\* pArr = new int[num];

if (pArr == NULL){

exit(100);

}

int i, temp;

\*pArr = \*(pArr + 1) = 1;

for (i = 2; i < num; i++)

\*(pArr + i) = \*(pArr + i - 1) + \*(pArr + i - 2);

temp = \*(pArr + i - 1);

delete[] pArr;

return temp;

}

**스택 활용**

/\*

다양한 스택활용#1

문자열 역순 출력

\*/

int main(void) {

string str;

cout << "문자열 입력: "; getline(cin, str);

stack<char> s;

for (int i = 0; i < str.length(); i++)

s.push(str[i]);

cout << "문자열 역순 출력: ";

while (!s.empty()) {

cout << s.top();

s.pop();

}

cout << endl;

return 0;

}

/\*

다양한 스택활용#2

십진수를 이진수로 출력

\*/

int main(void) {

int num;

cout << "십진수 입력: "; cin >> num;

stack<int> s;

while (num) {

s.push(num % 2);

num /= 2;

}

cout << "2진수 변환: ";

while (!s.empty()) {

cout << s.top();

s.pop();

}

cout << endl;

return 0;

}

/\*

다양한 스택활용#3

수식괄호검사

\*/

int main(void) {

string str;

bool state;

cout << "수식 입력: ";

getline(cin, str);

state = testPair(str);

if (state == true)

cout << "수식의 괄호 쌍이 맞습니다." << endl;

else

cout << "수식의 괄호 쌍이 틀립니다." << endl;

return 0;

}

int testPair(string str) {

stack<int> s;

char symbol;

for (int i = 0; i < str.length(); i++) {

//1) 여는 괄호 : 스택에 PUSH

if (str[i] == '(' || str[i] == '{' || str[i] == '[')

s.push(str[i]);

//2) 닫는 괄호 : 스택에서 POP하여 여는 괄호와 짝을 비교

else if (str[i] == ')' || str[i] == '}' || str[i] == ']') {

if (s.empty())

return false;

symbol = s.top();

if (str[i] == ')' && symbol != '('

|| str[i] == '}' && symbol != '{'

|| str[i] == ']' && symbol != ']')

return false;

s.pop();

}

}

int state;

if (s.empty()) state = true;

else state = false;

//s.~LinkedStack();

return state;

}

**선택정렬**

/\*

선택정렬

1. 정렬 안된 리스트에서 가장 작은 값의 위치 탐색(pSm)

2. 정렬되지 않은 리스트의 시작 위치의 원소와 교환(SWAP)

3. i++(리스트의 경계를 한개의 원소만큼 이동)

\*/

void selectionSort(int\* pArr, int num) {

int\* pSm;

for(int i=0; i<num; i++){

pSm = pArr + i;

for (int j = i; j < num; j++) {

//가장 작은 값을 가진 원소의 위치(주소) 탐색

if (\*pSm > \*(pArr + j))

pSm = pArr +j;

}

//가장 작은 값을 선택 : 현재 값과 교환

SWAP(pSm, pArr + i);

}

}

**단순연결리스트**

/\*

단순연결리스트 구현#1

tail, count 쓰는 버전

\*/

#include <iostream>

using namespace std;

//SNode class

class SNode {

int \_\_data;

SNode\* \_\_link;

friend class SlinkedList;

public:

SNode(const int& data);

};

//생성자와 메소드 정의

SNode::SNode(const int&data)

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

//SlinkedList Class

class SlinkedList {

SNode\* \_\_head;//첫 번째 노드

SNode\* \_\_tail;//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);//생성

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 : 첫번째 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;//비어 있으면 1

}

//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;

SlinkedList sList = SlinkedList();

while (true) {

cout << "임의의 정수 입력(종료:0): ";

cin >> num;

if (num == 0)break;

sList.addRear(num);//맨 마지막 노드로 입력값 삽입

}

sList.printSLinkedList();

sList.~SlinkedList();

return 0;

}

/\*

단순연결리스트 구현#2

tail, count 안 쓰는 버전

\*/

#include <iostream>

using namespace std;

//SNode class

class SNode {

int \_\_data;

SNode\* \_\_link;

friend class SLinkedList;

public:

SNode(const int& data);

};

//생성자와 메소드 정의

SNode::SNode(const int& data)

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

//SlinkedList Class

class SLinkedList{

SNode\* \_\_head;//첫 번째 노드

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

};

SLinkedList::SLinkedList()

:\_\_head(nullptr){}

SLinkedList::SLinkedList() {

SNode\* tNode = \_\_head;

while (tNode) {

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

delete tNode;

tNode = \_\_head;

}

}

//삽입 : 맨마지막 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;

}

//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()) {

return;

}

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) {

if (num == 0)

break;

sList.addRear(num);//맨 마지막 노드로 삽입

}

sList.printSLinkedList();

return 0;

}

/\*

단순연결리스트#3

ver.스택

\*/

#include <iostream>

#include "Linkedstack.h"

using namespace std;

int main(void) {

int num, choice;

LinkedStack<int>S;// 빈 스택 생성

while (true) {

system("cls");

cin >> choice;

switch (choice) {

case 1:

S.push(num);

break;

case 2:

S.pop();

break;

case 3:

S.printStack();

break;

case 4:

return 0;

default:

cout << "wrong choice" << endl;

}

system("pause");

}

return 0;

}

#include<iostream>

using namespace std;

inline void error(const char\* message) {

cout << message;

exit(100);

}

template<typename E>

class LinkedList;

//SNode Class

template<typename E>

class SNode {

public :

SNode(const E& data);

E getData(void)const;

private:

E \_\_data;

SNode<E>\* \_\_link;

template<typename E>

friend class LinkedStack;

};

//SNode Class : 멤버 함수 외부 정의

template<typename E>

SNode<E>::SNode(const E& data)

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

template<typename E>

E SNode<E>::getData(void)const {

return \_\_data;

}

//LinkedStack Clasee

template<typename E>

class LinkedStack {

public:

LinkedStack();

~LinkedStack();

void push(const E& e);

void pop(void);

E& top(void) const;

bool empty(void) const;

int size(void) const;

void printStack(void) const;

private:

SNode<E>\* \_\_top;

int \_\_count;

};

//LinkedStack class:생성자(소멸자)와 소멸자

template<typename E>

LinkedStack<E>::LinkedStack()

:\_\_top(nullptr), \_\_count(0){}

template<typename E>

LinkedStack<E>::~LinkedStack() {

SNode\* temp = \_\_top;

while (temp) {

\_\_top = temp->\_\_link;

delete temp;

temp = \_\_top;

}

}

//pop:스택에서 맨 마지막에 삽입된 데이터 항목 삭제

template<typename E>

void LinkedStack<E>::pop(void) {

if(empty()

error("ERROR :: STACK IS EMPTY");

SNode<int> \*temp=\_\_top;

if (temp) {

\_\_top=temp->\_\_link;

delete temp;

\_\_count--;

}

}

//top(peek) : 스택에서 맨 마지막에 삽입된 데이터 항목 확인

template<typename E>

E& LinkedStack<E>::top(void) const {

if (empty())

error("STACK IS EMPTY");

return \_\_top->\_\_data;

}

//empty : 스택의 공백 여부 확인

template<typename E>

bool LinkedStack<E> ::empty(void) const {

return \_\_top == nullptr;

}

//size : 스택의 크기(항목의 개수)

template<typename E>

int LinkedStack<E> ::size(void) const {

return \_\_count;

}

//printStack : 스택의 전체 데이터 항목 출력

template<typename E>

void LinkedStack<E>::printStack(void) const {

SNode<E>\* temp = \_\_top;

cout << "\n STACK[";

while (temp) {

cout.width(3);

cout << temp->\_\_data;

temp = temp->\_\_link;

}

cout << "]" << endl;

}

/\*

단순연결리스트#4

송

\*/

#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);

}

//생성자&소멸자

SLinkedList::SLinkedList()

:head(nullptr), tail(nullptr), count(0){}

SLinkedList::~SLinkedList() {

while (!isEmpty)

removeRear();

}

//생성

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

SNode\* s = new SNode;

s->data = e;

s->link = nullptr;

return s;

}

//조사

bool SLinkedList::isEmpty()const {

return head == nullptr;

}

int SLinkedList::countSNode(){

return count;

}

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

SNode\* temp = head;

while (temp != NULL) {

count++;

if (temp->data == e)

return count;

else

temp = temp->link;

}

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

//첫번째 노드부터 삽입

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

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

newSNode->link = head;//head 값 물려받음

head = newSNode;

}

//중간 노드 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;

}

}

//마지막 노드부터 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

//첫번째 노드부터 삭제

void SLinkedList::removeFront() {

if (isEmpty())

return;

SNode\* old = head;

head = old->link;

delete old;

count--;

}

//중간 노드삭제

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

if (isEmpty())

return;

SNode\* old = preSNode->link;

preSNode->link = old->link;

delete old;

count--;

}

//맨 마지막 노드 삭제

void SLinkedList::removeRear() {

SNode\* temp = head;

if (isEmpty())

return;

if (temp == tail) {//노드 하나일 때

delete head;

head = nullptr;

tail = nullptr;

count--;

return;

}

SNode\* old = tail;

count = 1;

//이전 노드 탐색

while (temp->link != tail) {

temp = temp->link;

count++;

}

temp->link = nullptr;

tail = temp;

delete old;

count--;

}

//메인 함수

int main(void) {

int num;

SLinkedList s = SLinkedList();

while (true) {

if (num == 0)

break;

s.insertRear(num);

}

//전체 원소 출력

if (s.isEmpty()){

cout<<"No"<<endl;

return 0;

}

s.printSLinkedList();//입력된 데이터

s.countSNode();//총 데이터 갯수

s.~SLinkedList();

return 0;

}

**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\_node에서 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.

**이중연결리스트**

/\*

이중연결리스트#1

tail, count 없는 ver.

\*/

#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

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(cont)

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

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

};

//DLinkedList : 생성자(소멸자)와 메소드 정의

DLinkedList::DLinkedList()

: \_\_head(nullptr){}

DLinkedList::~DLinkedList() {

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 번재 노드

void DLinkedList::removeFront() {

DNode\* old;

while (\_\_head) {

old = \_\_head;

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::printDLinkedList()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::revprintDLinkedList() 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.printDLinkedList();

dList.revprintDLinkedList();

return 0;

}

/\*

이중연결리스트#2

tail, count 있는 ver.

\*/

#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

void removeFront1(); // 삭제 : 첫 번째 노드(교수님 코드)

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(cont)

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

void revprintDLinkedList() 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 노드

void DLinkedList::removeFront() {

if (!isEmpty()) {

DNode\* old = \_\_head;

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

if (\_\_head != nullptr)

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

delete old;

\_\_count--;

if (isEmpty()) {

\_\_tail = nullptr;

}

}

};

//노드 삭제 : 첫 번째 노드(교수님 코드)

void DLinkedList::removeFront1() {

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::printDLinkedList()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::revprintDLinkedList() 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.printDLinkedList();

dList.revprintDLinkedList();

dList.~DLinkedList();

return 0;

}

/\*

이중연결리스트#3

송

\*/

#include<iostream>

using namespace std;

inline void error(const char\* message) {

cout << message << endl;

exit(100);

}

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\* preDNode);

void deleteRear();

//출력

void printBy\_RLink()const;

void printBy\_Llink()const;

};

//생성자 & 소멸자

DLinkedList:: DLinkedList()

: head(nullptr), tail(nullptr), count(0){}

DLinkedList::~DLinkedList() {

while (!isEmpty())

deleteRear();

if (isEmpty())

cout << "NULL";

exit(0);

}

//생성

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

DNode\* nNode = new DNode;

nNode->data = e;

nNode->Llink = nullptr;

nNode->RLink = nullptr;

return nNode;

}

//조사

bool DLinkedList::isEmpty()const {

return head == nullptr;

}

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 = nullptr;

head = old->RLink;

delete old;

count--;

}

void DLinkedList::deleteMid(DNode\* preDNode) {

if (isEmpty())

error("빈 리스트입니다.");

DNode\* old = preDNode -> RLink;//삭제할 노드

preDNode->RLink = old->RLink;

old->RLink->Llink = nullptr;

delete old;

count--;

}

void DLinkedList::deleteRear() {

if (isEmpty())

error("빈 리스트입니다.");

DNode\* old = tail;//삭제할 노드

//노드 하나 남은 경우

if (head == tail) {

head = nullptr;

tail = nullptr;

}

else {

//n-1노드 RLink->NULL로

//tail->n-1로

old->Llink->RLink = nullptr;

tail = old->Llink;

}

delete old;

count--;

}

//출력

void DLinkedList::printBy\_RLink()const {

DNode\* temp = head;

while (temp) {

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

temp = temp->RLink;

}

cout << "NULL" << endl;

}

void DLinkedList::printBy\_Llink()const {

DNode\* temp = tail;

while (temp) {

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

temp = temp->Llink;

}

cout << "NULL" << endl;

}

int main(void) {

int num;

DLinkedList d = DLinkedList();

while (true) {

cin >> num;

if (num == 0)

break;

d.insertRear(num);

}

if (d.isEmpty()) {

return 0;

}

d.printBy\_RLink();

d.printBy\_Llink();

d.~DLinkedList();

return 0;

}

**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

**순차 자료구조**

/\*

순차자료구조

스택ver.

\*/

#include<iostream>

#define stackMAXSIZE 100

using namespace std;

inline void error(const char\* message);

//배열 스택 : arrayStack -- 클래스 설계

template <typename E>

class arrayStack {

private:

E stack[stackMAXSIZE];

int top;

public:

arrayStack();

~arrayStack();

void push(const E& e);

E pop(void);

E peek(void)const;

bool isEmpty(void)const;

bool isFull(void)const;

void printStack(void)const;

};

//생성자 : 빈 스택 생성

template <typename E>

arrayStack<E>::arrayStack()

: top(-1){}

//소멸자

template <typename E>

arrayStack<E>::~arrayStack(){}

//PUSH : 스택에 데이터 삽입

template <typename E>

void arrayStack<E>::push(const E& e) {

if (isFull())error("스택 포화 에러");

return stack[++top] = e;

}

//POP : 스택에서 데이터 삭제

template <typename E>

E arrayStack<E>::pop(void) {

if (isEmpty())error("스택 공백 에러");

return stack[top--] = e;

}

//PEEK : 스택 맨 위의 원소 확인

template <typename E>

E arrayStack<E>::peek(void) const {

if (isEmpty())error("스택 공백 에러");

return stack[top];

}

//스택의 공백 상태 여부 판단

template <typename E>

bool arrayStack<E>::isEmpty(void)const {

return top == -1;

}

//스택의 포화 상태 여부 판단

template <typename E>

bool arrayStack<E>::isFull(void)const {

return top == stackMAXSIZE - 1;

}

//스택의 전체 원소 출력

template <typename E>

void arrayStack<E>::printStack(void)const {

cout << "\n STACK [";

for (int i = 0; i <= top; i++) {

cout.width(3);

cout << stack[i];

}

cout << " ] " << endl;

}

inline void error(const char\* message) {

cout << message;

exit(100);

}

int main(void) {

int num, choice;

arrayStack<int> s = arrayStack<int>();//빈 스택 생성

while (true) {

system("cls");

cin >> choice;

system("pause");

}

return 0;

}

**이진탐색**

/\*

이진탐색#1

재귀적 용법

\*/

#include <iostream>

#define arrMAXSIZE 10;

using namespace std;

int main(void) {

int key;

while (true) {

cin >> key;

if (key == 0)

break;

int index = binarySearch(arr, 0, arrMAXSIZE - 1, key);

if (index == EOF)cout << "없다고!!!" << endl;

else cout << "검색 데이터: " << arr[index] << "번째 위치" << index + 1 << endl;

}

return 0;

}

int binarySearch(int\* pArr, int first, int last, int key) {

if (first > last)//재귀함수의 탈출 조건

return EOF;

//검색 범위의 중간 원소의 위치 값 계산

int index, mid = (first + last) / 2;

if (\*(pArr + mid) == key) return mid;

else if (\*(pArr + mid) > key) index = binarySearch(pArr, first, mid - 1, key);

else if (\*(pArr + mid) < key) index = binarySearch(pArr, mid + 1, last, key);

return index;

}

/\*

이진탐색#2

비재귀적 용법

\*/

int\* binarySearch(int\* pArr, int\* pFirst, int\* pLast, int key) {

while (pFirst <= pLast) {

int\* pMid = pFirst + (pLast - pFirst) / 2;

if (\*pMid == key) return pMid;

else if (\*pMid < key) pFirst = pMid + 1;

else if (\*pMid > key) pLast = pMid - 1;

}

return NULL;

}

**삽입정렬**

/\*

삽입정렬#1

\*/

void insertionSort(int\* pArr, int num) {

int i, j, temp;

for (int i = 1; i < num; i++) {

temp = \*(pArr + i);

for (j = i - 1; j >= 0 && \*(pArr + j) > temp; j--)

\*(pArr + j + 1) = \*(pArr + j);

/\*

j=i-1;

while (j >= 0 && \*(pArr + j)>temp) {

\*(pArr + j + 1) = \*(pArr + j);

j--;

}

\*/

\*(pArr + j + 1) = temp;

}

}

**버블정렬**

/\*

버블정렬 #1

\*/

void bubbleSort(int\* pArr, int num) {

for (int i = 0; i < num; i++) {

for (int j = num - 1; j > 0; j--)

if (\*(pArr + j) < \*(pArr + j - 1))

SWAP(pArr + j, pArr + j - 1);

}

}

/\*

버블정렬 #2

\*/

void bubbleSortUp(int\* pArr, int num) {

for (int i = 0; i < num; i++)

{

bool state = true;

for (int j = num - 1; j > 0; j--)

{

if (\*(pArr + j) < \*(pArr + j - 1))

SWAP(pArr + j, pArr + j - 1);

state = false;

}

if (state = true)

return;

}

}

**쉘정렬**

/\*

쉘 정렬

\*/

//쉘 정렬에 쓰이는 삽입 정렬 : interval 간격만큼 삽입 정렬

void intervalSort(int\* pArr, int num, int start, int interval) {

int i, j, temp;

for (i = start; i < num; i += interval) {

temp = \*(pArr + i);

for (j = i - interval; j >= 0 && \*(pArr + j) > temp; j -= interval)

\*(pArr + j + interval) = \*(pArr + j);

\*(pArr + j + interval) = temp;

}

}

//쉘 정렬 : 오름차순

void shellSort(int\* pArr, int num) {

int interval = num;

while (interval >= 1) {

interval /= 2;

for (int i = 0; i < interval; i++) {

intervalSort(pArr, num, i, interval);

}

print("intervla: %d, ", interval);

}

}

**병합정렬**

//병합 정렬 : 오름차순

void mergeSort(int\* pArr, int\* pFirst, int\* pLast) {

int\* mid = NULL;

if ((pFirst - pArr) < (pLast - pArr)) {

mid = pFirst + (pLast - pFirst) / 2;//first와 last 사이의 중간 원소의 위치

mergeSort(pArr, pFirst, mid); //왼쪽 부분집합 정렬

mergeSort(pArr, mid + 1, pLast); //오른쪽 부분집합 정렬

merge(pArr, pFirst, mid, pLast);//정렬된 두 부분집합 병합

}

}

void merge(int\* pArr, int\* pFirst, int\* pMid, int\* pLast) {

int i = pFirst - pArr;

int j = (pMid - pArr) + 1;

int t = i;

int\* tempArr = new int[arrMAXSIZE];

while (i <= (pMid - pArr) && j <= (pLast - pArr)) {

if (pArr[i] <= pArr[j]) {

tempArr[t++] = pArr[i++];

}

else {

tempArr[t++] = pArr[j++];

}

}

while (i <= (pMid - pArr)) tempArr[t++] = pArr[i++];

while (j <= (pLast - pArr)) tempArr[t++] = pArr[j++];

//정렬된 상태로 재구성된 temp 배열을 원본 배열 A에 복사

i = pFirst - pArr;

t = i;

while (i <= (pLast - pArr)) pArr[i++] = tempArr[t++];

delete[] tempArr;

}

**계수정렬**

//계수 정렬

void contingSort(int\* pArr, int num) {

int\* countArr, \* sortArr;

int maxNum = 0;

for (int i = 0; i < num; i++)

if (maxNum < pArr[i])

maxNum = pArr[i];

// countArr : 원본 데이터에서 각 항목들의 발생 횟수를 저장한다.

// 단, 원본 데이터의 값 자체가 countArr의 첨자가 된다.

countArr = new (nothrow) int[maxNum + 1] { 0 };

if (countArr == nullptr) {

cout << "동적 메모리 공간 할당 실패!!!" << endl;

return;

}

for (int i = 0; i < num; i++) countArr[pArr[i]]++;

for (int i = 1; i <= maxNum; i++) countArr[i] += countArr[i - 1];

// sortArr : countArr의 값을 첨자로 하여 원본 데이터를 정렬된 데이터로 구성한다.

sortArr = new (nothrow) int[num] {0};

if (countArr == nullptr) {

cout << "동적 메모리 공간 할당 실패!!!" << endl;

exit(100);

}

for (int i = num - 1; i >= 0; i--) {

sortArr[countArr[pArr[i]] - 1] = pArr[i];

countArr[pArr[i]]--;

// PRINT(sort, num);

}

// 정렬된 데이터(sortArr)로 원본 데이터(pArr)를 재구성한다.

for (int i = 0; i < num; i++)

pArr[i] = sortArr[i];

delete[] countArr;

delete[] sortArr;

}

**계수 정렬** : 각 숫자의 갯수를 센 수, 작은 수부터 갯수만큼 출력

전제조건

1) 정수로 표현 가능한 배열만 정렬할 수 있다.

2) arr의 data값 중 최댓값을 알아야한다.

과정

1) 각 data의 갯수를 센다

2) 카운트 배열에 갯수를 저장한다.

\* 카운트 배열(count\_arr)이란 : count\_arr의 인덱스 = data값, count\_arr의 데이터 = data의 갯수

3) 누적합 배열 생성

-> count\_arr의 data값을 왼쪽부터 계속 누적하여 자기 자신 자리에 대입

4) for문으로 배열을 정리하여 출력

stable 과 unstable

배열 내에 동일한 값이 두개 이상 존재할 때 그 값의 순서가 보장되는 것을 stable하다고 표현한다.

ex) {1,3,2,1} 을 {1ㄱ,3,2,1ㄴ} 이라 표현할 때, 정렬 후의 모습이 {1ㄱ,1ㄴ,2,3}이면 stable하다

만약 {1ㄴ,1ㄱ,2,3} 일 경우 unstable 하다.

stable은 매번 동일한 값을 출력하고 싶은 경우에 사용한다.

stable : mergeSort, insertSort, countSort, bubbleSort

unstable : quickSort,heapSort, selectionSort, shellSort

**퀵 정렬(Pivot)**

/\*

퀵 정렬

\*/

int\* Partition(int\* pArr, int\* pfirst, int\* plast) {

int pivot = \*plast;//마지막 원소를 기준 값으로 선택

int i = (pfirst - pArr) - 1;

for (int j = (pfirst - pArr); j < (plast - pArr); j++) {

if (pArr[i] <= pivot)

SWAP(pArr + (++i), pArr + i);//기준 값을 가운데로 위치 시킨다.

}

SWAP(pArr + i + 1, plast);

return pArr + i + 1;//기준 값의 위치 값을 반환

}

//퀵 정렬: 오름차순

void quickSort(int\* pArr, int\* pfirst, int\* plast) {

int\* pmid = nullptr;

if(pfirst-pArr<plast-pArr){

pmid = Partition(pArr, pfirst, plast);//분할 후 기준 값의 위치 값을 반환

quickSort(pArr, pfirst, pmid - 1);//왼쪽 부분 정렬

quickSort(pArr, pmid + 1, plast);//오른쪽 부분 정렬

}

}

/\*

퀵정렬#2

교수님 코드

\*/

void quickSort(int\* pArr, int\* pfirst, int\* plast) {

//재귀 함수 탈출조건

if (pfirst > plast)

return;

//분할 : 기준 값의 왼쪽(작은 값)과 오른쪽(큰 값) 부분 집합

int\* mid, \* pi, \* pj;

pi = pfirst - 1;

for (pj = pfirst; pj < plast; pj++)

if (\*pj <= \*plast)//기준 값(pivot):마지막 원소

SWAP(++pi, pj);

SWAP(pi + 1, plast);//기준 값을 가운데로

mid = pi + 1;//기준 값의 위치(주소)

quickSort(pArr, pfirst, mid - 1);//왼쪽 부분 정렬

quickSort(pArr, mid + 1, plast);//오른쪽 부분 정렬

}