2023 October 18

**자료구조와 알고리즘 과제제출**

실습보고서 2

로고이(가) 표시된 사진

자동 생성된 설명

**담당교수 : 서두옥교수님**

**학과 : 자동차it융합과**

**학번 : 20215229**

**이름 : 이채훈**

**제출일 : 2023년10월18일**

**1. 피보나치 수열 : 성능평가**

**소스코드**

#include <iostream>

#include <ctime> // clock 또는 time 함수를 이용하여 코드의 실행 시간을 측정

using namespace std;

int Fibo(int num);

int main(void)

{

clock\_t start, finish;

double seconds;

int num;

cout << "20215229 이채훈" << endl;

cout << endl;

cout << "### 피보나치 수열 구하기 ###" << endl;

cout << endl;

cout << "몇 번째 수열까지 출력할까요? : "; cin >> num;

start = clock();

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

cout.width(8);

if (i % 5) cout << Fibo(i);

else cout << Fibo(i) << endl;

}

finish = clock();

seconds = (double)(finish - start) / CLOCKS\_PER\_SEC;

cout << num << "번째 피보나치 수열 계산 시간: " << seconds << endl;

return 0;

}

//재귀적 용법

int Fibo(int num) {

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

return 1;

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

}

//비 재귀적 용법

int Fibo(int num) {

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

return 1;

int first, second, res = 0;

first = second = 1;

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

res = first + second;

first = second;

second = res;

}

return res;

}#include <iostream>

#include <ctime> // clock 또는 time 함수를 이용하여 코드의 실행 시간을 측정

using namespace std;

int Fibo(int num);

int main(void)

{

clock\_t start, finish;

double seconds;

int num;

cout << "20215229 이채훈" << endl;

cout << endl;

cout << "### 피보나치 수열 구하기 ###" << endl;

cout << endl;

cout << "몇 번째 수열까지 출력할까요? : "; cin >> num;

start = clock();

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

cout.width(8);

if (i % 5) cout << Fibo(i);

else cout << Fibo(i) << endl;

}

finish = clock();

seconds = (double)(finish - start) / CLOCKS\_PER\_SEC;

cout << num << "번째 피보나치 수열 계산 시간: " << seconds << endl;

return 0;

}

//재귀적 용법

int Fibo(int num) {

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

return 1;

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

}

//비 재귀적 용법

int Fibo(int num) {

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

return 1;

int first, second, res = 0;

first = second = 1;

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

res = first + second;

first = second;

second = res;

}

return res;

}#include <iostream>

#include <ctime> // clock 또는 time 함수를 이용하여 코드의 실행 시간을 측정

using namespace std;

int Fibo(int num);

int main(void)

{

clock\_t start, finish;

double seconds;

int num;

cout << "20215229 이채훈" << endl;

cout << endl;

cout << "### 피보나치 수열 구하기 ###" << endl;

cout << endl;

cout << "몇 번째 수열까지 출력할까요? : "; cin >> num;

start = clock();

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

cout.width(8);

if (i % 5) cout << Fibo(i);

else cout << Fibo(i) << endl;

}

finish = clock();

seconds = (double)(finish - start) / CLOCKS\_PER\_SEC;

cout << num << "번째 피보나치 수열 계산 시간: " << seconds << endl;

return 0;

}

//재귀적 용법

int Fibo(int num) {

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

return 1;

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

}

/\*

//비 재귀적 용법

int Fibo(int num) {

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

return 1;

int first, second, res = 0;

first = second = 1;

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

res = first + second;

first = second;

second = res;

}

return res;

}

**실행결과**

**재귀적용법**

**텍스트, 스크린샷, 디스플레이, 폰트이(가) 표시된 사진

자동 생성된 설명**

**비재귀적용법**

**텍스트, 스크린샷, 디스플레이이(가) 표시된 사진

자동 생성된 설명**

**연구조사**

**피보나치 수열**

: n번째 항이 n-1번째와 n-2번째 항의 합인 수열

**폰트, 친필, 타이포그래피, 텍스트이(가) 표시된 사진

자동 생성된 설명**

**1. 재귀적 용법**

텍스트, 폰트, 도표, 친필이(가) 표시된 사진

자동 생성된 설명

**2. 비재귀적 용법**

텍스트, 친필, 폰트, 도표이(가) 표시된 사진

자동 생성된 설명

**결론**

두 가지 방법으로 구현한 피보나치 수열을 비교했을 때, 비재귀적 용법의 속도가 더 빠른 것을 확인할 수 있다. 이는 재귀적 용법을 사용해, 피보나치 수열을 구현할 경우 함수의 중복호출로 인해 O(2^n)의 지수함수에 비례하는 시간이 들기 때문이다.

**2. 삽입 정렬: 알고리즘 구현**

**소스코드**

#include <iostream>

#include <random>

#define arrMAXSIZE 15

using namespace std;

void insertionSort(int\* pArr, int num); // 삽입 정렬

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

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

//배열 생성

int arr[arrMAXSIZE] = { 0 };

//C++ 스타일 난수 생성

random\_device rd;

mt19937 gen(rd());

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

cout << "20215229 이채훈" << endl;

cout << endl;

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

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

cout << "정렬 전: ";

PRINT(arr, arrMAXSIZE);

insertionSort(arr, arrMAXSIZE);

cout << "정렬 후: ";

PRINT(arr, arrMAXSIZE);

return 0;

}

// 삽입 정렬: 오름차순

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

int i, j, temp;

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

temp = \*(pArr + i);

// temp 가 들어갈 공간 확보

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

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

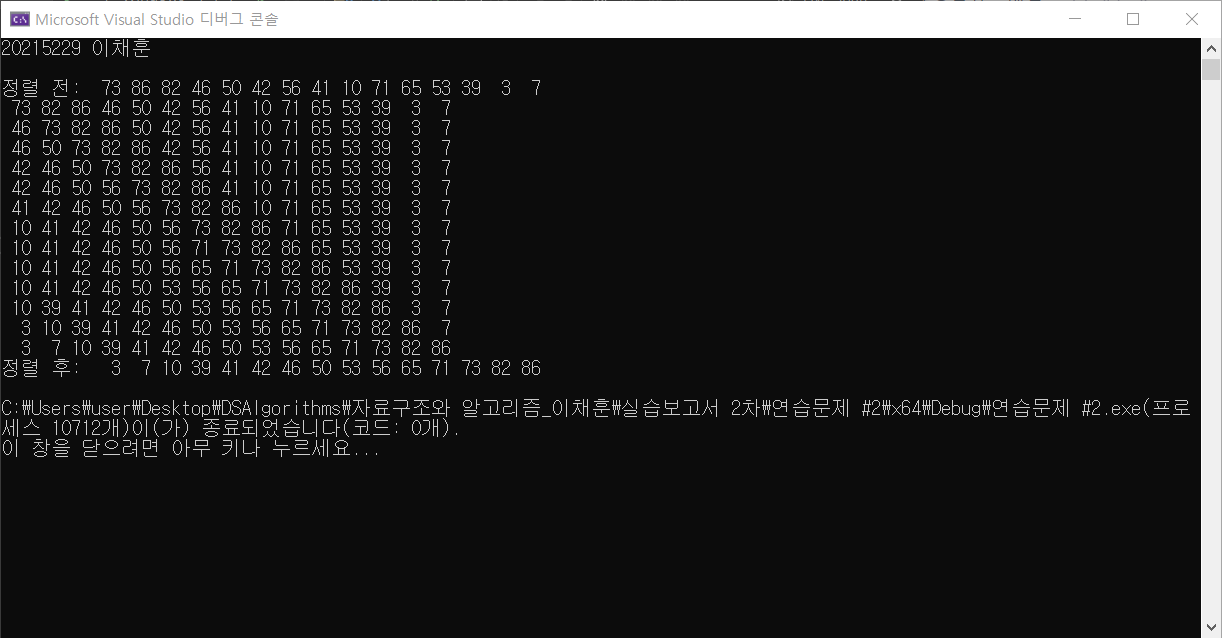
\*(pArr + j + 1) = temp;

PRINT(pArr, num);

}

}

**실행결과**

****

**연구조사**

**삽입정렬**

: 정렬되지 않은 원소를 정렬된 영역의 적당한 위치에 넣어주는 정렬

**삽입 정렬 수행 과정**

1) 각 단계에서 비 정렬 구역의 첫 번째 원소를 선택한다.

2) 첫 번쨰 원소를 정렬 구역의 원소와 비교하여 적당한 자리에 삽입한다.

**메모리 사용 공간**

n개의 원소에 대하여 n개의 메모리 사용

**연산 시간**

1. 최선의 경우 : 원소들이 이미 정렬 되어 있을 때 원소 비교횟수가 최소

- 이미 정렬되어 있는 경우에는 바로 앞자리 원소와 한번만 비교

- 전체 원소 비교횟수 = n-1

- 시간 복잡도 : O(n)

2. 최악의 경우 : 모든 원소가 역순으로 되어 있을 경우 원소 비교 횟수가 최대

- 전체 원소 비교 횟수 = 1+2+3+…+(n-1) = n(n-1)/2

- 시간 복잡도 : O(n^2)

3. 삽입 정렬의 평균 원소 비교 횟수 = n(n-1)/4

**🡪 평균 시간 복잡도 : O(n^2)1)**

**3-1. 이진탐색 : 재귀적 용법**

**소스코드**

#include <iostream>

#define arrMAXSIZE 10

using namespace std;

// 이진 탐색 : 재귀적 용법

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

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

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 5, 9, 13, 17, 21, 28, 37, 46, 55, 88 };

cout << "20215229 이채훈" << endl;

cout << endl;

cout << "원시 데이터: "; PRINT(arr, arrMAXSIZE);

int key;

while (true) {

cout << "검색 데이터 입력(검색 종료: 0) : "; cin >> key;

if (key == 0)

break;

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

if (index == -1)

cout << "없다고!!!" << endl;

else

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

}

return 0;

}

// 이진탐색: 재귀적용법

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

// 재귀함수의 탈출 조건

if (first > last)

return -1;

// 검색범위의 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);

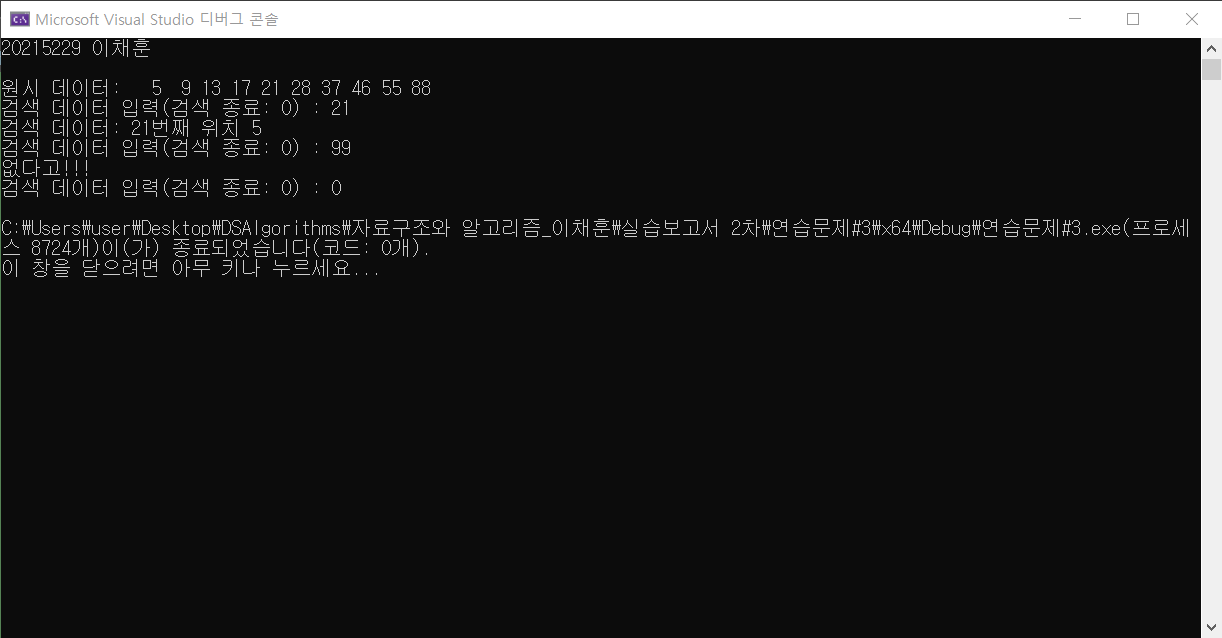
else if (\*(pArr + mid) < key)

index = binarySearch(pArr, mid + 1, last, key);

return index;

}

**실행결과**

****

**3-2. 이진탐색 : 비재귀적 용법**

**소스코드**

#include <iostream>

#define arrMAXSIZE 10

using namespace std;

// 이진 탐색 : 비재귀적 용법

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

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

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 5, 9, 13, 17, 21, 28, 37, 46, 55, 88 };

cout << "20215229 이채훈" << endl;

cout << endl;

cout << "원시 데이터: "; PRINT(arr, arrMAXSIZE);

int key;

while (true) {

cout << "검색 데이터 입력(검색 종료: 0) : ";

cin >> key;

if (key == 0)

break;

int\* pindex = binarySearch(arr, arr, arr + arrMAXSIZE - 1, key);

if (pindex == NULL)

cout << "없다고!!!" << endl;

else cout << "검색 데이터: " << \*pindex << "번째 위치 " << pindex - arr + 1 << endl;

}

return 0;

}

// 이진탐색: 비재귀적용법

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

while (pFirst <= pLast) { // 재귀함수의 탈출 조건

// 검색범위의 중간 원소의 위치 값 계산

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;

}

**실행결과**

텍스트, 스크린샷, 디스플레이, 컴퓨터이(가) 표시된 사진

자동 생성된 설명

**연구조사**

**이진 탐색**

: 중간값을 이용하여 찾고자하는 값인 key값을 찾아내는 알고리즘

**전제조건**

1) 배열이 정렬 상태여야 한다.

2) 각 데이터는 유일한 값을 갖는다.

**과정**

1) 탐색할 범위 경계값의 중간 값을 구한다.

2) 중간값과 key값을 비교한다

3) 중간값 > key값 : 탐색 범위를 첫번째 값 ~ 중간값-1 로 바꾼다

중간값 < key값 : 탐색 범위를 중간값+1 ~ 마지막 값 으로 바꾼다.

**Q. mid 찾을 때, (pFirst+pLast)/2 를 못하는 이유?**

포인터끼리의 연산은 뺄셈만 의미를 가진다. 다른 연산은 임의의 위치를 나타낼 뿐이므로 의미를 갖지 않는다.

포인터끼리의 뺄셈은 두 포인터 간의 상대적 길이이다. 이 때 길이는 포인터의 자료형에 따라 다르게 나온다.

**4. 퀵 정렬 : 알고리즘 구현**

**소스코드**

#include <iostream>

#include <random>

#define arrMAXSIZE 15

using namespace std;

void quickSort(int\* pArr, int\* pFirst, int\* pLast);// 퀵 정렬

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

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

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 0 };

//C++ 스타일 난수 생성

random\_device rd;

mt19937 gen(rd());

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

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

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

cout << "20215229 이채훈" << endl;

cout << endl;

cout << "정렬 전: ";

PRINT(arr, arrMAXSIZE);

quickSort(arr, arr, arr + arrMAXSIZE - 1);

cout << "정렬 후: ";

PRINT(arr, arrMAXSIZE);

return 0;

}

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

int temp;

temp = \*pa;

\*pa = \*pb;

\*pb = temp;

}

//퀵 정렬 : 오름차순

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

int\* pmid = NULL;

if (pfirst - pArr < plast - pArr) {

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

PRINT(pfirst, plast-pfirst+1);

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

quickSort(pArr, pmid + 1, plast);//오른쪽 부분 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[j] <= pivot)

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

}

SWAP(pArr + i + 1, plast);

return pArr + i + 1;//기준 값의 위치 값을 반환

}

**실행결과**

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

자동 생성된 설명

**연구조사**

**퀵정렬**

: 기준 값을 이용하여 부분집합을 두 개로 나누고, 각각을 정렬한다.

기준 값을 중심으로 작은 원소들은 왼쪽 부분집합으로, 큰 원소들은 오른쪽 부분 집합으로 분할하여 정렬하는 방식의 정렬 알고리즘

**퀵 정렬 수행 과정**

1) Left index를 리스트 가장 왼쪽에, Right index를 리스트 가장 오른쪽에 위치시킨다.

2) Left 에 위치한 원소가 pivot보다 큰 값이 나올 때까지 Left를 리스트의 오른쪽 방향으로 이동시킨다. Right에 위치한 원소가 pivot보다 작거나 같은 값이 나올 때까지 리스트의 왼쪽 방향으로 이동시킨다.

3) Left와 Right가 교차하지 않았다면, 두 인덱스에 위치한 원소를 교환한 후, 2번 과정으로 돌아간다.

4) Left 와 Right가 교차했다면, pivot과 Right에 위치한 원소를 교환한다. 교환한 후의 pivot의 위치를 기준으로 왼쪽과 오른쪽의 부분집합을 구분하고, 두 부분 집합에 대하여 퀵 정렬 알고리즘을 재귀 호출한다.

**메모리 사용 공간**

n개의 원소에 대하여 n개의 메모리 사용

**연산 시간**

1. 최선의 경우 : O(nlogn)

기준 값에 의해서 원소들이 왼쪽 부분 집합과 오른쪽 부분 집합으로 정확히 n/2개씩 이등분이 되는 경우가 반복되어 수행 단계 수가 최소가 되는 경우

2. 최악의 경우 : O(n^2)

기준 값에 의헤 원소들을 분할하였을 때 1개와 n-1개로 한쪽으로 치우쳐 분할되는 경우가 반복되어 수행 단계 수가 최대가 되는 경우

3. 평균 시간 복잡도 : O(nlogn)

같은 시간 복잡도는 가지는 다른 정렬 방법에 비해서 자리 교환 횟수를 줄임으로써 더 빨리 실행되어 실행 시간 성능이 좋은 정렬 방법

**5. 병합 정렬 : 알고리즘 구현**

**소스코드**

#include <iostream>

#include <random>

#define arrMAXSIZE 15

using namespace std;

void mergeSort(int\* pArr, int\* pFirst, int\* pLast);// 병합 정렬

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

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

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

cout.width(3);

cout << \*(pArr + i);

}

cout << endl;

}

int main(void)

{

int arr[arrMAXSIZE] = { 0 };

//C++ 스타일 난수 생성

random\_device rd;

mt19937 gen(rd());

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

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

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

cout << "20215229 이채훈" << endl;

cout << endl;

cout << "정렬 전: ";

PRINT(arr, arrMAXSIZE);

mergeSort(arr, arr, arr + arrMAXSIZE - 1);

cout << "정렬 후: ";

PRINT(arr, arrMAXSIZE);

return 0;

}

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

int temp;

temp = \*pa;

\*pa = \*pb;

\*pb = temp;

}

//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

int leftSize = mid - pFirst + 1;

int rightSize = pLast - mid;

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

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

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

PRINT(pFirst, leftSize + rightSize);

}

}

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;

}

**실행결과**

텍스트, 스크린샷, 디스플레이, 컴퓨터이(가) 표시된 사진

자동 생성된 설명

**연구조사**

**병합정렬**

: 정렬된 집합을 합하여 하나의 집합을 만든다.

하나의 배열을 두 개의 균등한 크기로 분할하고 분할된 부분 집합을 정렬한 다음, 두 개의 정렬된 부분 집합를 합하여 전체가 정렬되게 하는 방법이다

**분할** : 입력 배열을 같은 크기의 2개의 부분 배열로 분할한다.

**정복** : 부분 배열을 정렬한다. 부분 배열의 크기가 충분히 작지 않으면 순환 호출 을 이용하여 다시 분할 정복 방법을 적용한다.

**결합** : 정렬된 부분 배열들을 하나의 배열에 합병한다

**병합 정렬 수행 과정**

1) 정렬되지 않은 리스트를 원소 1개의 부분리스트로 분할한다.

2) 부분 리스트가 하나만 남을 때까지 반복 병합하면서 정렬된 부분 리스트를 생성한다.

3) 마지막 하나 남은 부분 리스트가 정렬된 리스트가 된다.

**메모리 사용 공간**

n개의 원소에 대하여 (2\*n)개의 메모리 공간이 사용된다.

**연산 시간**

1. 분할 단계 : n개의 원소를 분할하기 위해서 log2n번의 단계 수행

2. 병합 단계 : 부분집합의 원소를 비교하면서 병합하는 단계에서 최대 n번의 비교연산 수행

**🡪 전체 병합정렬의 시간 복잡도 : O(nlog2n)**

**6. 간단한 데이터 처리 : 단순 연결 리스트 구현**

**소스코드**

#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;

}

**실행결과**

텍스트, 스크린샷, 디스플레이, 컴퓨터이(가) 표시된 사진

자동 생성된 설명

**연구조사**

**Q. 함수 뒤에 const 의 의미**

: 읽기 전용 함수

**addRear 함수**

: 연결 리스트의 맨 뒤에 새로운 노드를 추가함. 빈 리스트인 경우, \_\_head와 \_\_tail이 새로운 노드를 가리키게 되며, 그렇지 않으면 \_\_tail의 \_\_link를 새로운 노드로 갱신하고 \_\_tail을 업데이트함.

**removeFront 함수**

: 연결 리스트의 첫 번째 노드를 삭제함. 먼저 리스트가 비어 있는지 확인하고, 첫 번째 노드를 삭제한 후 \_\_head와 \_\_tail을 갱신함.

**frontSNode, rearSNode, isEmpty, countSNode 함수**

: 연결 리스트의 상태를 확인하고 조작하기 위한 함수

**7. 간단한 데이터 처리 : 이중 연결 리스트 구현**

**소스코드**

#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;

}

}

}

// 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; **🡨 //return \_\_tail == 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;

}

**실행결과**

텍스트, 스크린샷, 디스플레이, 컴퓨터이(가) 표시된 사진

자동 생성된 설명

**연구조사**

**Q. 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.

**# 단순. 원형 연결 리스트의 문제점 극복 : 연결리스트**

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

* 연결리스트는 각 노드가 이전 노드와 다음 노드에 대한 포인터를 갖는 자료구조로, **원하는 위치에서 삽입과 삭제를 빠르게 수행할 수 있는 장점**이 있음.

**DNode Class**

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

**\* DNode 클래스는 DLinkedList 클래스와 friend 선언되어 있기 때문에, DLinkedList 클래스에서 DNode 클래스의 private 멤버에 접근할 수 있음.**

**addRear 함수**

: 주어진 데이터를 이중 연결 리스트의 맨 뒤에 삽입함.

**removeFront 함수**

: 첫 번째 노드를 삭제함.

**frontDNode 함수**

: 첫 번째 노드를 반환함.

**rearDNode 함수**

: 맨 마지막 노드를 반환함.