2019년 1학기
자료구조 최종 결과 보고서

Movie Database Management Service

제출일 : 2019년 6월 12일

담당교수	이영구 교수님
이름	우수진
Email	wsj1998@naver.com

1. 설계 배경

A. 설계 작품의 중요성/필요성

시장조사전문 기업[1]에서는 성인남녀 1,000명을 대상으로 '영화'에 관한 설문조사를 진행하였다. 그 결과, 약 90%의 사람들이 영화 관람을 가장 쉽게 접할 수 있는 문화생활이라 답했다. 이러한 근거들을 토대로 '영화'가 이제 현대인들의 일상에서 어느 한 부분을 차지하게 되었다고 말해도 과언이 아닐 것이다.

이에 현대인들이 SNS, 사진, 블로그 등을 통해 자신의 일상을 틈틈이 기록하는 모습이 떠올랐고, 이를 영화에 접목시켜보았다. 이제 영화도 우리 일상의 한 부분이 된 만큼, 고객들은 영화와 관련된 다양한 서비스를 필요로 하고 있을 것이다.

이러한 근거를 토대로 자신이 시청한 영화 정보를 관리할 수 있는 'Movie Database Management Service'를 구상하게 되었다.

B. 설계 목적

사람들은 의미 있는 일에 대해 기록을 남기는 것을 중요하게 생각한다. 예전에는 그것이 주로 종이와 펜을 통해 이루어졌다면, 최근에는 기술의 발전과 함께 휴대폰, 컴퓨터, 카메라 등 다양한 매개체를 통해 이루어지고 있다. 그리고 이를 통해 사람들은 바쁜 일상 속에서도 맛있는 음식, 예쁜 장소, 감동적인 순간 등을 끊임없이 기록해두려 한다.

이는 영화도 예외는 아닐 것이다. 영화에 관심이 있는 사람들에게는 자신이 본 영화에 대한 기록이 의미 있는 활동이 될 것이다. 이에 'Movie Database Management Service' 개발을 통해 사람들이 자신이 본 영화를 언제든지 간편하게 기록할 수 있게끔 하고자 한다.

---

1) https://www.trendmonitor.co.kr/tmweb/trend/allTrend/detail.do?bIdx=1485&code=0303&trendType=CKOREA&prevMonth=&currentPage=2

C. 기존 유사제품과의 비교[2)]

다양한 영화 관련 서비스가 제공되고 있다. 그러나 기존의 서비스는 보통 영화 예매나 결제를 지원하며, 이에 대한 시청 기록이 남는 형태 정도로 영화 정보가 관리된다. 또한 관람 장소, Device 등에 따라 영화 정보가 여기저기 저장된다는 한계도 있다. 고객들이 자신이 시청한 영화를 한 개의 서비스에서 한 번에 관리할 수 있도록 하고자 하였으며, 이에 'Movie Database Management Service'를 개발하게 되었다. 고객들은 이 Service를 통해 원하는 영화를 직접 등록할 수 도 있다.

D. 설계 작품의 독창성/창의성

영화예매, 영화 다시보기와 같은 앱이나, 서비스는 이미 많이 존재한다. 그러나 아직 자신이 본 영화를 체계적으로 저장해둘 수 있는 서비스는 잘 없다. 기존의 영화 관련 프로그램은 영화를 예매하고, 자신이 예매해서 본 영화의 기록을 확인하는 것과 같은 서비스를 제공해왔다. 물론, 블로그 등을 활용하여 자신이 본 영화에 대한 줄거리, 평 등을 기록할 수는 있지만 이는 다소 공을 들어야 하는 작업이다. 바쁜 현대인들에게 이렇게 까지 영화목록을 관리하는 것은 다소 부담스러울 수 있을 것이다. 따라서 나는 기존의 프로그램들과 달리 이름, 영화배우 등의 간단한 내용 기입만으로도 자신이 본 영화에 대한 정보를 관리할 수 있도록 하였다. 뿐만 아니라,

---

2) https://www.tbroad.com/service/viewBsServicePage.tb?viewName=viewBsVodPage&lnbActiveCd=S901&innerTabCd=S9

누구나 쉽고 빠르게 자신의 영화목록을 관리할 수 있는 서비스를 개발하였다.

E. 기대효과

영화를 좋아하는 사람 중 하나로서, 감상한 영화에 대한 '기록'은 내게 굉장히 의미 있는 일이다. 따라서 나처럼 영화를 좋아하는 사용자들에게는 'Movie Database Management Service'가 일기처럼 가치 있는 기록을 남기도록 도와줄 것이다. 그들에게는 영화가 일상생활의 한 부분이기도 하니까 말이다. 그런 만큼 사람들이 이 Service를 통해 자신이 본 영화에 대한 기록을 꾸준히 하며, 바쁜 삶속에서도 여가생활을 이어나가길 바란다. 그리고 자신만의 Movie Database를 만들어가며, 그곳에서 소소한 행복도 느낄 수 있으면 좋겠다.

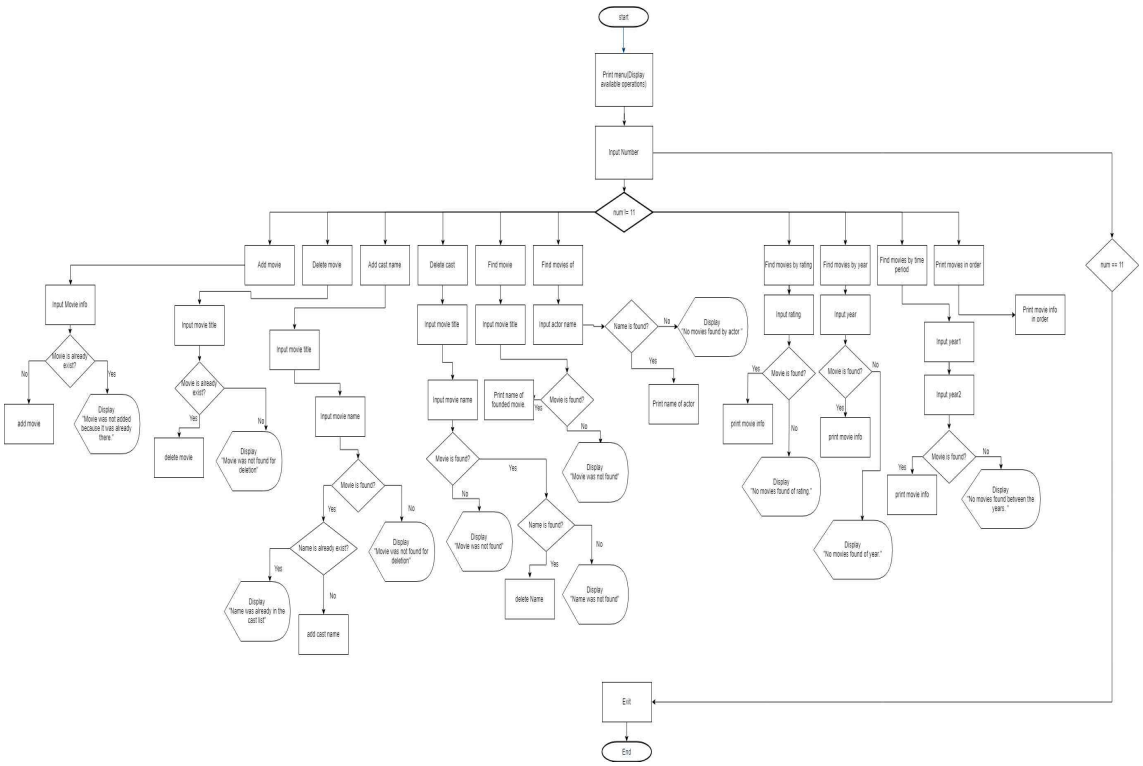
F. 참고문헌

- "여전히 이렇다 할 여가활동 부족한 한국사회, 모두 '영화관'으로?", https://www.trendmonitor.co.kr/tmweb/trend/allTrend/detail.do?bIdx=1485&code=0303&trendType=CKOREA&prevMonth=&currentPage=2, 2016년 7월

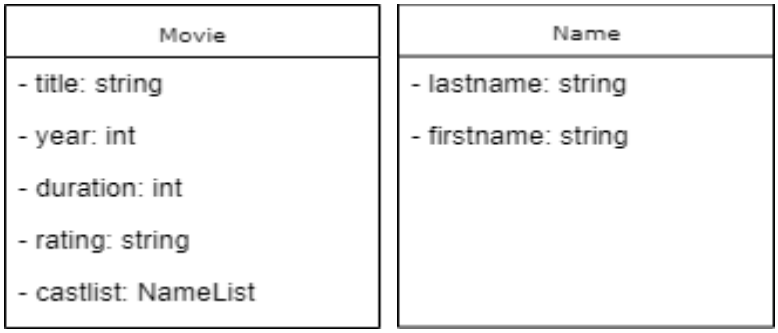
- Nell Dale, *C++ Data Structures* (JONES AND BARTLETT PUBLISHERS, 1998년)

2. 설계 결과

A. 작품의 개념도 및 동작 설명



- Member parameter info



(1) Add movie

우선 BST에 사용자가 저장하려는 영화가 이미 존재하는지 확인한다. 여기서는 movie title이라는 key를 이용하여 이에 부합하는 값을 찾는다. 이때 recursion 방식을 이용한다. tree에서 주어진 object를 찾으면 pre-order tree iterator가 반환된다. 반면에 찾지 못하면 empty iterator가 반환된다. 그리고 이 정보를 통해 iterator가 current position에서 반환할 data를 가지고 있는지를 확인함으로써 영화의 존재 여부를 확인해준다.(이때 Queue 자료구조가 사용된다.) 그리고 사용자가 입력하려는 영화가 존재하지 않는다면 Binary Search Tree에 recursion을 사용해서 Movie정보를 저장해준다.

* 여기서 사용되는 Queue는 node들을 순서대로 가지고 있으며, iterator에 대한 current node는 queue의 front에 위치한다. Queue가 empty상태라면 이는 iterator에 data가 없음을 의미한다.

⇨ 사용된 자료구조: Binary Search Tree, List(배우 이름 저장), Queue

(2) Delete movie

우선 BST에 사용자가 저장하려는 영화가 이미 존재하는지 확인한다. 여기서는 movie title이라는 key를 이용하여 이에 부합하는 값을 찾는다. 이때도 recursion방식이 이용된다. tree에서 주어진 object를 찾으면 pre-order tree iterator가 반환되고, 찾지 못하면 empty iterator가 반환된다. 그리고 이 정보를 가지고 iterator가 current position에서 반환할 data를 가지고 있는지 확인함으로써 영화의 존재 여부를 확인해준다. (이때 Queue 자료구조가 사용된다.) 그리고 영화가 존재한다면 BST에서 element를 recursive하게 제거해준다.

⇨ 사용된 자료구조: Binary Search Tree, List(배우 이름 저장), Queue

* 위에서 이미 언급하여 반복되는 내용은 일정부분 생략하여 아래 내용을 작성 하였습니다.

(3) Add cast

이번에는 movie의 title key를 이용하여 BST에서 검색을 진행해준다. 마찬가지로 검색 결과 pre-order tree iterator가 반환된다. 그리고 iterator에 data가 있다면 castlist에 사용자가 입력하고자 하는 배우의 name이 circular doublye linked list에 있는지 검색한다. 없다면 name을 삽입하되, 삽입 후에도 list는 sorted상태로 유지되어야 한다.

⇨ 사용된 자료구조: Binary Search Tree, Sorted Doubly Linked list, Queue

(4) Delete cast

마찬가지로 movie의 title key를 이용하여 BST에서 검색을 진행해준다. 검색 결과 pre-order tree iterator가 반환된다. 그리고 iterator에 data가 존재한다면 castlist에 삭제하고자 하는 배우의 name이 존재하는지도 확인해준다. 이 역시 존재함이 확인되면 list에서 cast name을 제거해주되, sorted order가 유지된다.

⇨ 사용된 자료구조: Binary Search Tree, Sorted Doubly Linked list, Queue

(5) Find movie

movie의 title key를 이용하여 BST에서 검색을 진행해준다. 검색 결과 pre-order tree iterator가 반환된다. 그리고 iterator에 data가 있다면 traverse된 tree의 current node에 있는 data를 반환해준다. 그리고 이 정보가 사용자가 입력한 정보와 같은지 비교한다.

⇨ 사용된 자료구조: Binary Search Tree, Sorted Doubly Linked list, Queue

(6) Find movie actor

새로운 in-order iterator를 생성하고, 이로써 root로부터 in-order tree iterator가 반환된다. iterator가 data를 가지고 있다면 list에 사용자가 찾고자 하는 배우의 name이 존재하는지 확인한다. 그리고 list에 배우의 name도 존재한다면 그 movie에 대한 전체정보를 출력해준다. 그리고 count변수에 1을 더해준다. 이로써 0은 적합한 영화를 찾지 못한 경우, 1이상은 적합한 영화를 찾은 경우로 판단할 수 있다.

⇨ 사용된 자료구조: Binary Search Tree, Sorted Doubly Linked list, Queue

(7) Find movie rating

새로운 in-order iterator를 생성하고, 이로써 root로부터 in-order tree iterator가 반환된다. iterator가 data를 가지고 있다면 이에 대한 rating을 반환받는다. 그리고 사용자가 입력한 rating값과 비교하여 같다면 그 movie에 대한 전체정보를 출력해준다. 그리고 count변수에 1을 더해준다. 따라서 0은 적합한 영화를 찾지 못한 경우, 1이상은 적합한 영화를 찾은 경우가 된다.

⇨ 사용된 자료구조: Binary Search Tree, Sorted Doubly Linked list, Queue

(8) Find movie year

새로운 in-order iterator를 생성하고, 이로써 root로부터 in-order tree iterator가 반환된다. iterator가 data를 가지고 있다면 이에 대한 year을 반환받는다. 그리고 사용자가 입력한 year값과 비교하여 같다면 그 movie에 대한 전체정보를 출력해준다. 그리고 count변수에 1을 더해준다. 따라서 0은 적합한 영화를 찾지 못한 경우, 1이상은 적합한 영화를 찾은 경우가 된다.

⇨ 사용된 자료구조: Binary Search Tree, Sorted Doubly Linked list, Queue

(9) Find movie period

새로운 in-order iterator를 생성하고, 이로써 root로부터 in-order tree iterator가 반환된다. iterator가 data를 가지고 있다면 이에 대한 year을 반환받는다. 여기서는 사용자가 두 개의 year(year1, year2)에 대한 정보를 입력하게 되는데, year1은 반환받은 year의 값보다 작거나 같고, year2는 반환받은 year의 값보다 크거나 같을 경우 출력이 이루어진다. 그리고 출력이 되는 조건에서는 count변수에 1을 더해준다. 이로써 0은 적합한 영화를 찾지 못한 경우, 1이상은 적합한 영화를 찾은 경우로 생각할 수 있다.

⇨ 사용된 자료구조: Binary Search Tree, Sorted Doubly Linked list, Queue

(10) print movies order

in-order iterator를 통해 Binary Search Tree에 있는 data에 순차적으로 접근하여 출력해준다.

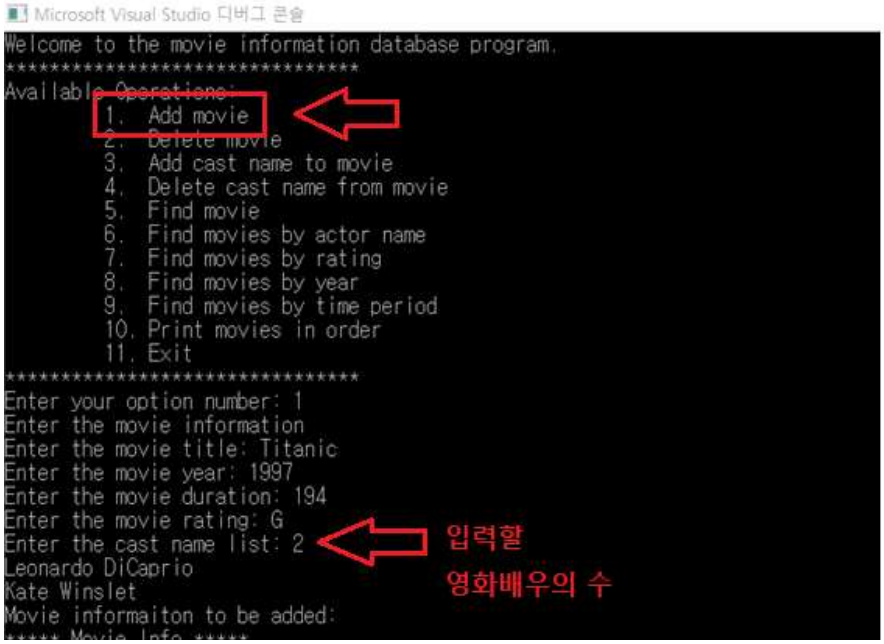
⇨ 사용된 자료구조: Binary Search Tree, Queue

B. 작품 주요화면 및 특성 설명

아래 세 개의 영화에 대한 정보를 입력하여 데모 방법과 데모 시 주의할 점에 관해 설명하겠다.

번호	영화정보	
1	title	Titanic
	year	1997
2	duration	194
	rating	G
	cast	Leonardo DiCaprio, Kate Winslet
3	title	3Idiots
	year	2009
	duration	141
	rating	G
	cast	Aamir Khan, Kareena Kapoor, Sharman Joshi
	title	TheMazeRunner
	year	2009
	duration	113
	rating	G
	cast	Dylan O'Brien, KiHong Lee, Kaya Scodelario

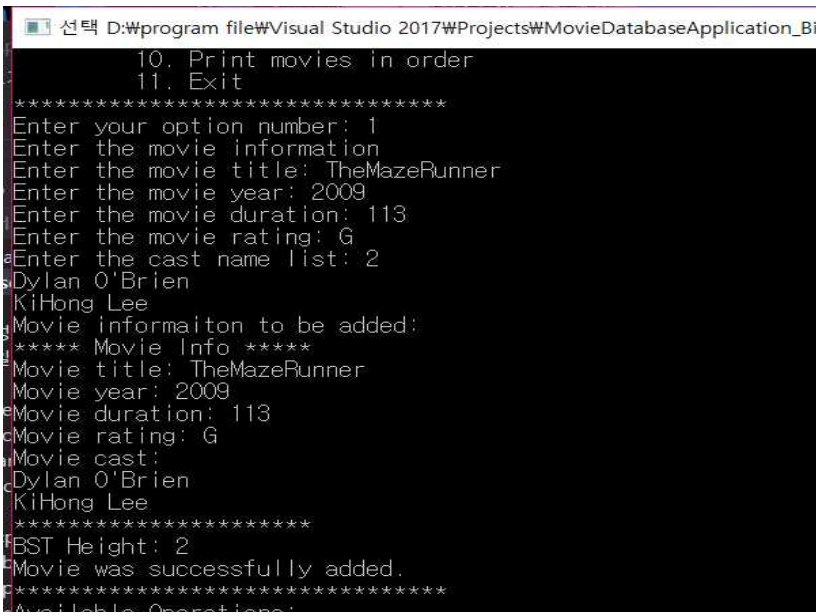
1) Add movie



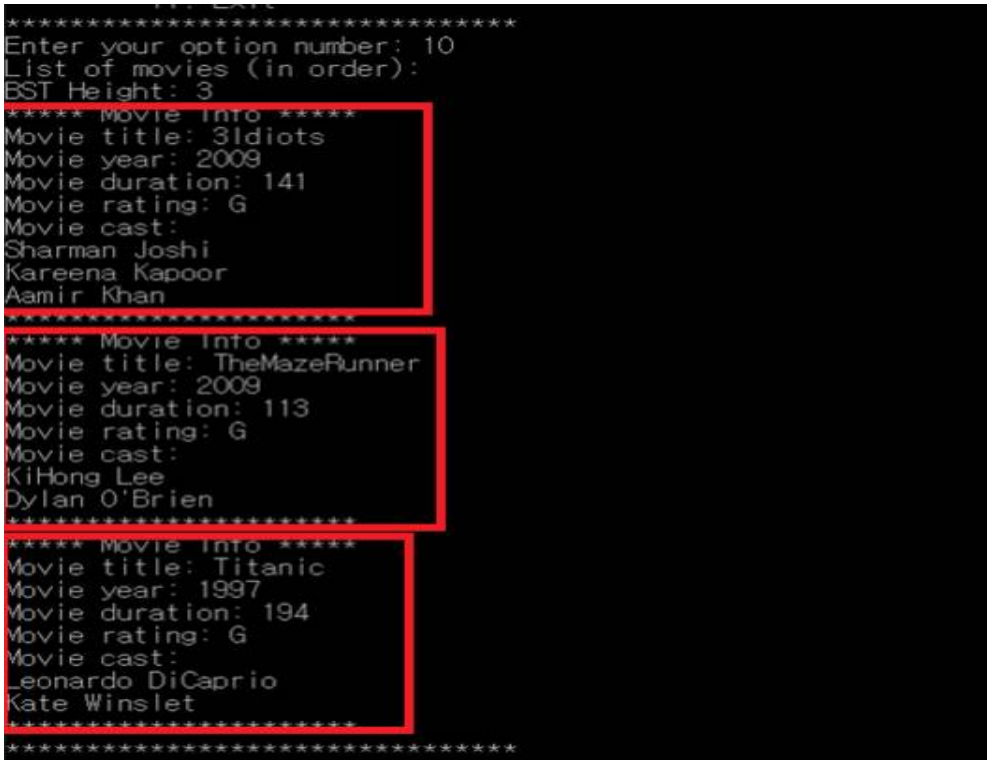
▷ 영화 이름은 특별한 자료구조 없이 string type으로 받아오기 때문에 띄워 쓰기 없이 입력해주어야 한다.

▷ 입력 가능한 Rating종류는 아래와 같다.
ex) G, PG, PG-13, R, NC-17, NR

▷ 영화배우 이름을 입력하기 전에, 총 몇 명의 배우를 입력할 것인지 int값으로 넣어준다. 그 다음 두개의 단어로 이루어진 배우의 이름을 입력한다. 만일 int 값으로 3을 입력한다면 세 명의 배우 이름을 저장할 수 있는 것이다.

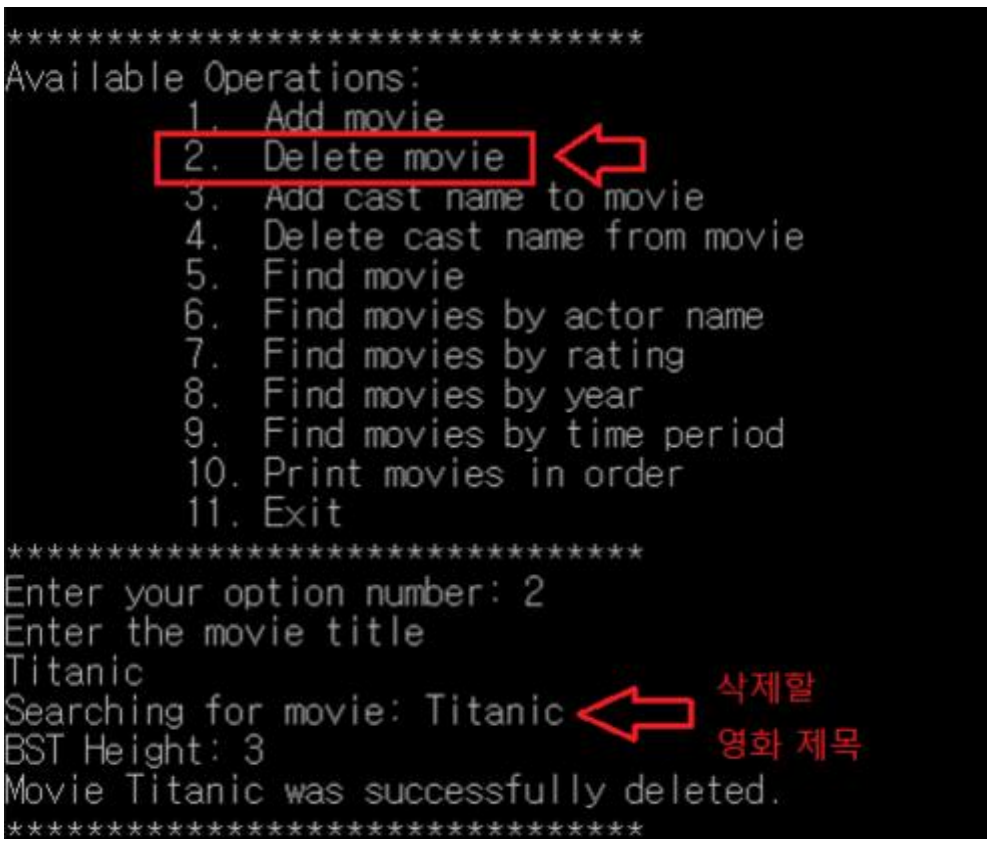


'10. Print movies in order'를 통해 저장된 영화에 대한 정보를 출력한 내용이다.



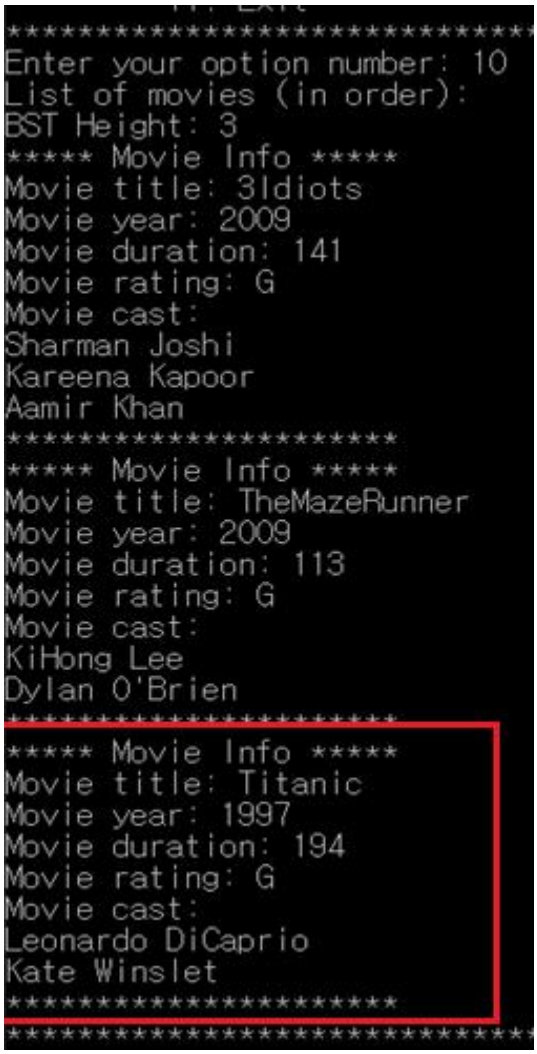
세 개의 영화를 모두 입력한 결과이다.

2) Delete movie



▷ Database에 영화가 하나 저장되어 있을 경우에는 delete가 불가하다. 따라서 두 개 이상의 영화가 저장되어 있을 때 'Delete movie'를 실행시켜줘야 한다.

▷ 영화 내에 2명 이상의 배우가 저장되어 있을 경우, 확실하게 삭제해주기 위해 'Delete cast name from movie'를 통해 삭제하고자 하는 영화 내의 모든 cast 정보를 delete해준 후 'Delete movie'를 실행시켜주면 된다.

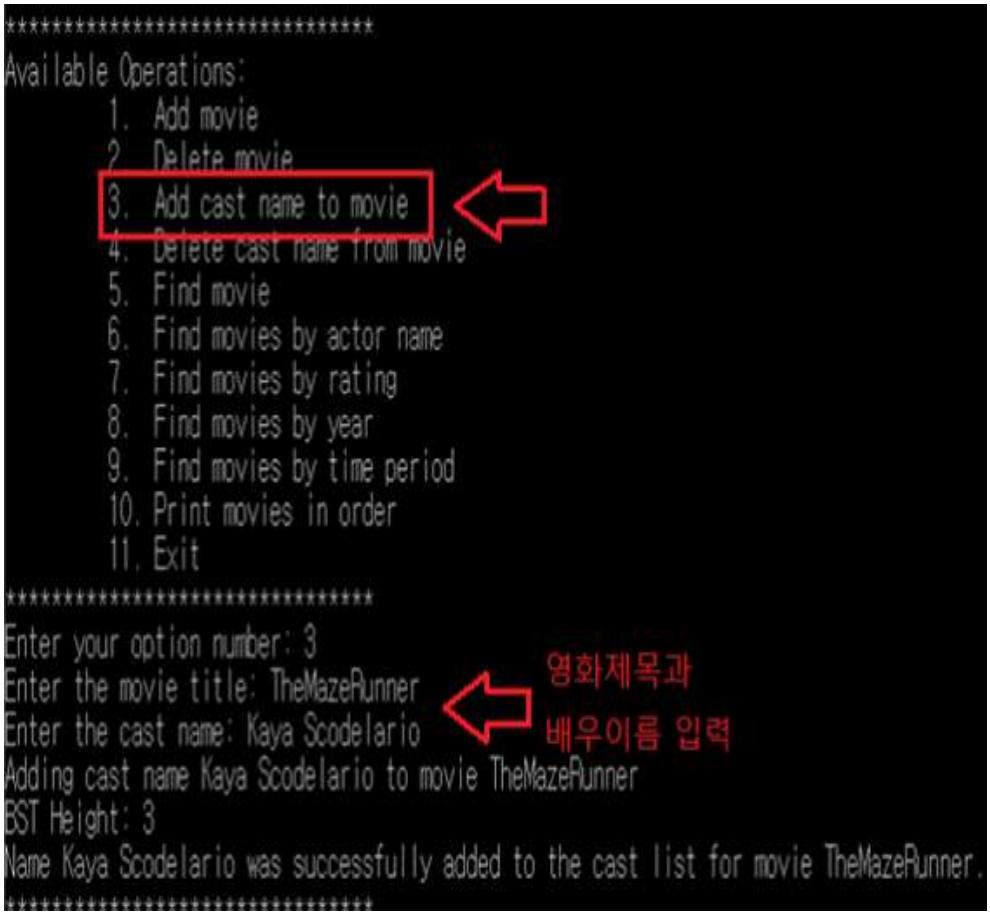


<*Before*>

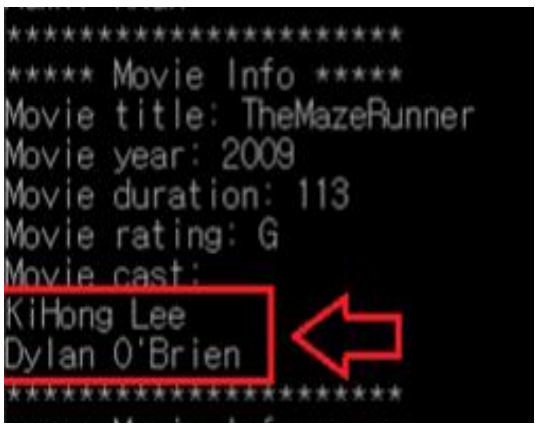
<*after*>

'10. Print movies in order'를 통해 출력해 본 결과 잘 삭제되었음을 확인할 수 있다.

3) Add cast



▷ 내가 추가하고자 하는 한 명의 영화배우 이름을 입력한다. 한명이기 때문에 숫자 입력 없이 바로 두 단어짜리 이름을 입력한다.



<Before>

<After>

'The Maze Runner'라는 제목의 영화정보에 한명의 배우가 add되었음으로 알 수 있다.

4) Delete cast

▷ 지우고자 하는 한 명의 영화배우 이름을 입력한다. 한명의 이름을 입력하기 때문에 숫자 입력 없이 바로 두 단어짜리 이름만 입력하면 된다.

<Before>

<한명의 배우이름을 Delete한 결과>

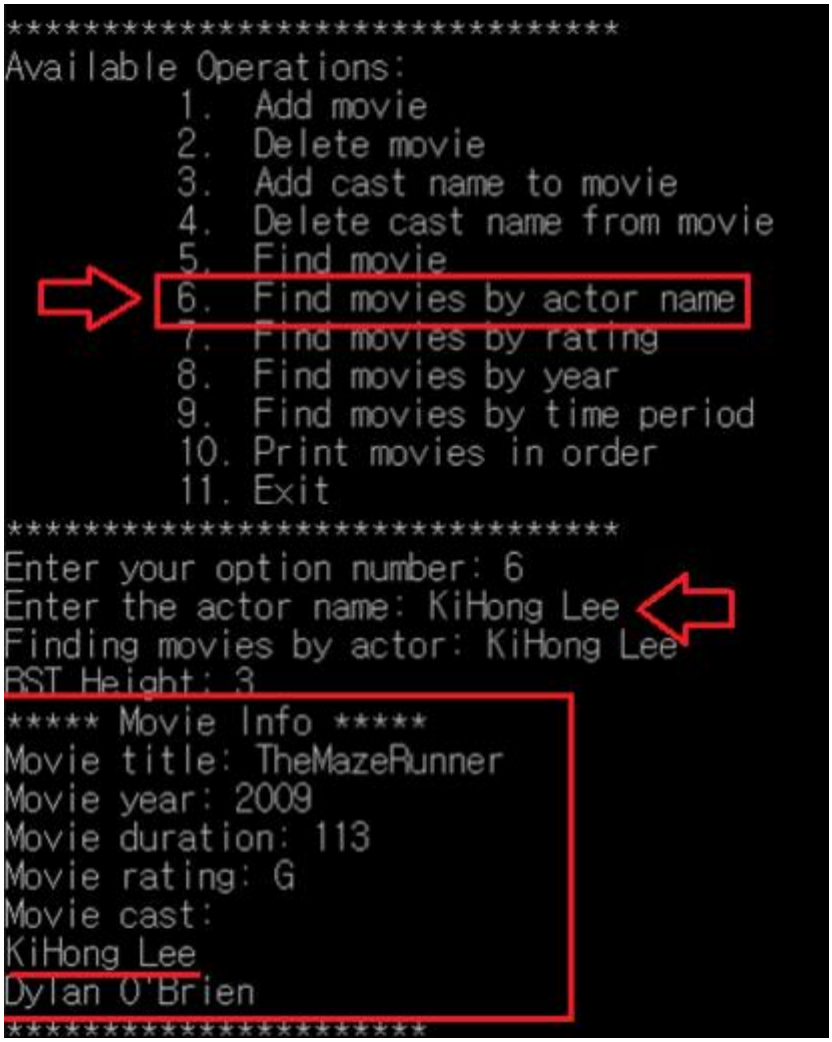
'Titanic'이라는 영화에 내가 입력한 배우의 이름이 delete되었음을 확인할 수 있다.

<저장되어있던 모든 배우의 이름을 Delete한 결과>

5) Find movie

▷ 영화 제목은 string type으로 받아오기 때문에, 찾고 싶은 영화의 제목을 띄워쓰기 없이 입력해준다.

6) Find movie actor



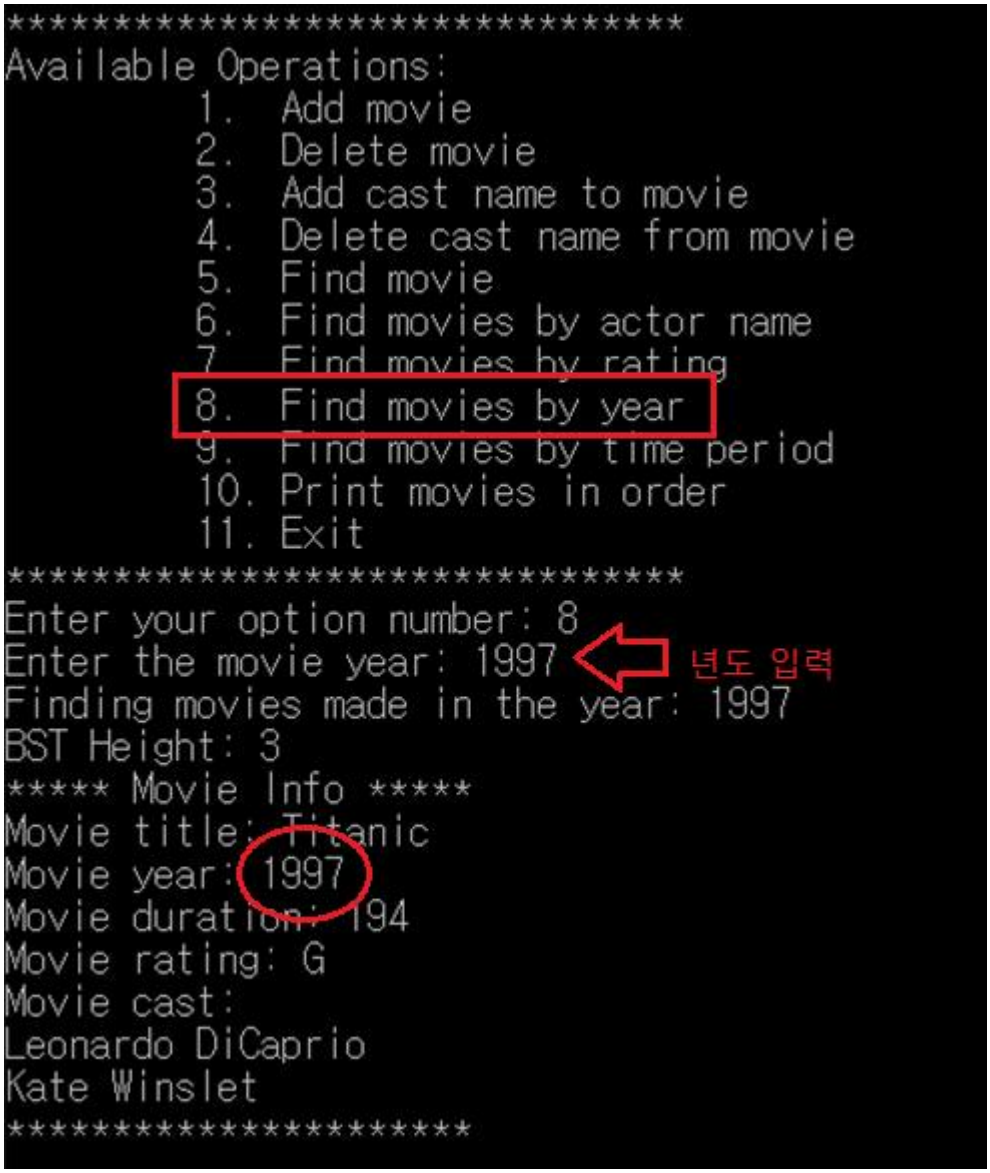
▷ 찾고자 하는 한명의 영화배우 이름을 입력한다. 이 역시 숫자 입력 없이 두 단어짜리 이름만 입력한다.

7) Find movie rating

▷ 찾고자 하는 영화의 등급을 입력해준다. 프로젝트에서 다루고 있는 등급의 종류는 아래와 같다.

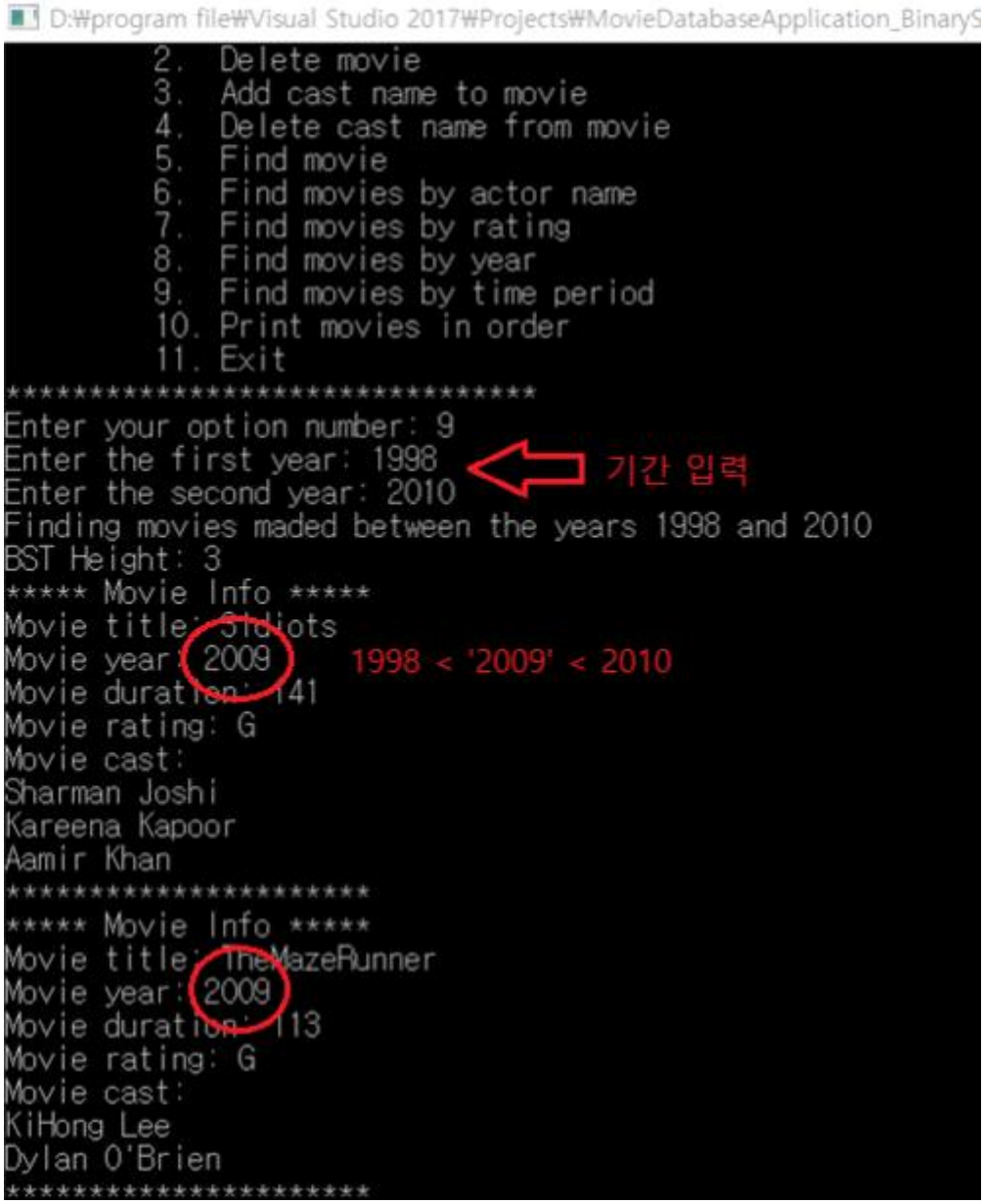
▷ G, PG, PG-13, R, NC-17, NR

8) Find movie year



▷ 특정 년도에 개봉된 영화를 찾을 수 있다. int값을 입력해주면 된다.

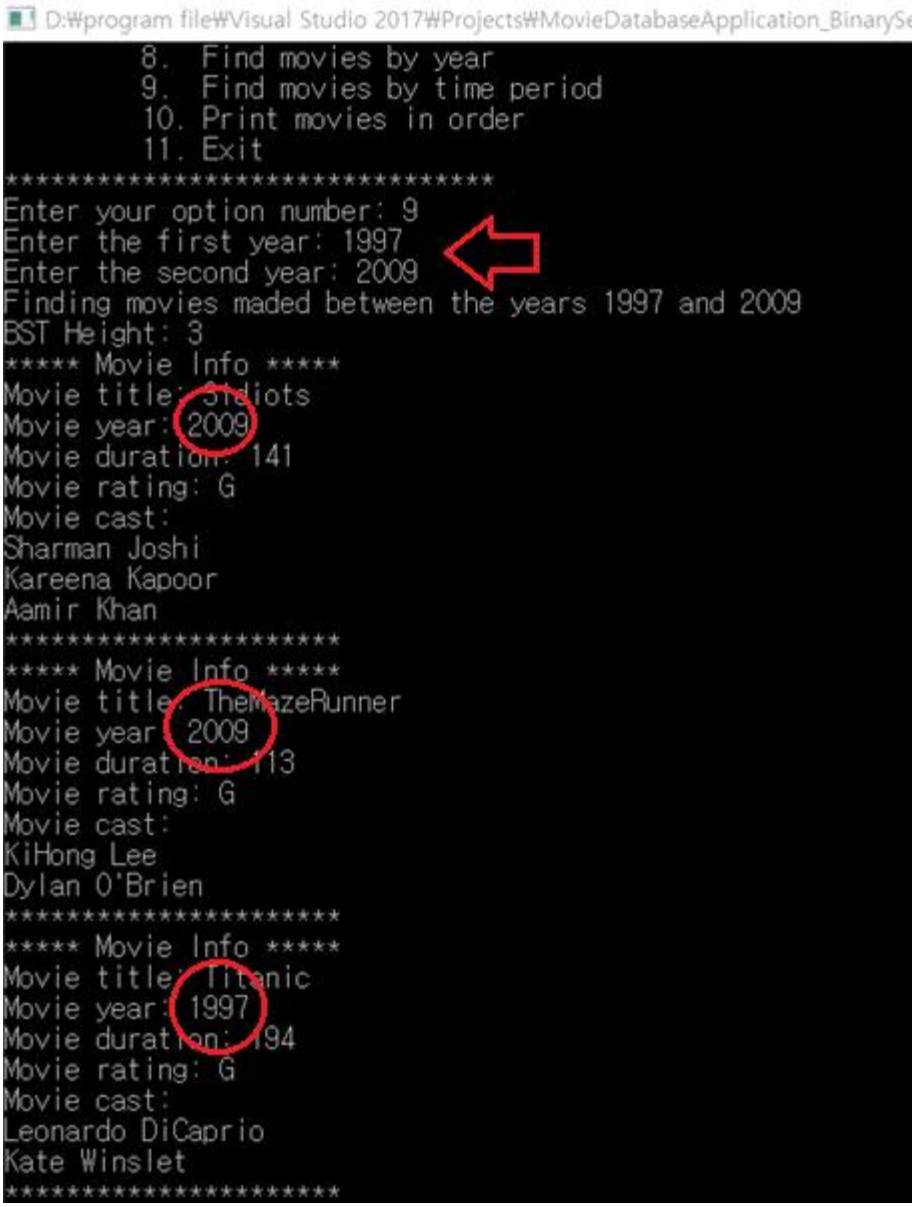
(9) Find movie period



▷ 특정 기간 내에 개봉된 영화를 찾아주기 때문에, 두개의 int값을 입력해준다.

▷ 만일 first year를 2015, second year를 2019로 입력하면, 2015년에서 2019년 사이에 개봉한 영화를 찾아준다.(2015년, 2019년 포함)

10) print movies order

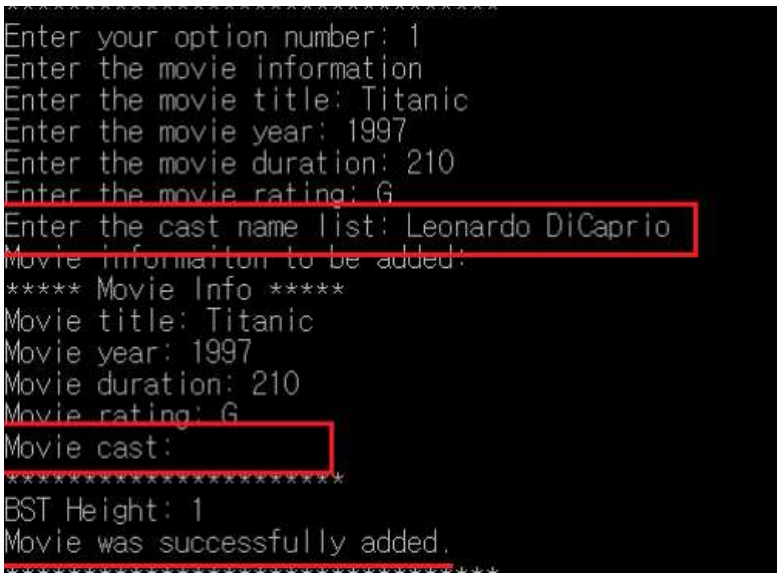


▷ input값으로 10을 주면 Database내에 저장된 모든 영화에 대한 정보가 한번에 출력된다.

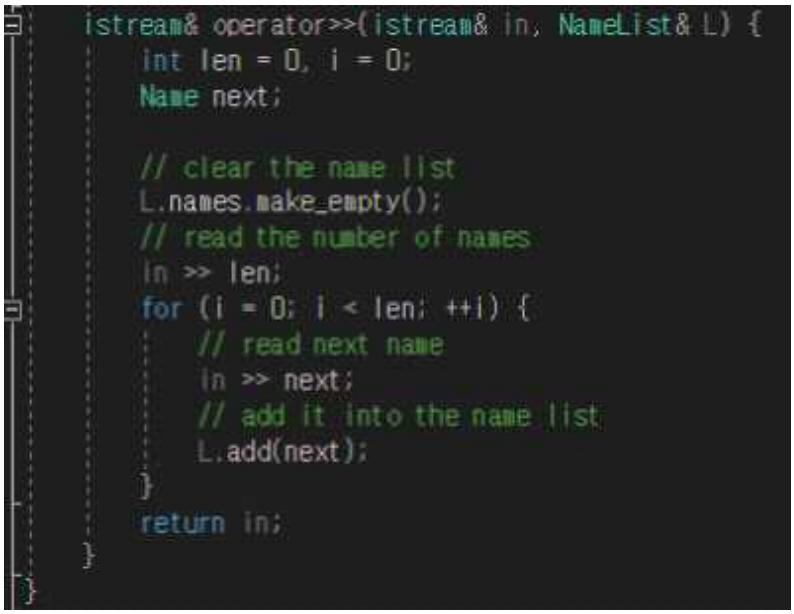
C. 문제점 및 개선된 내용

1) 실행 시 문제점

가) Add movie



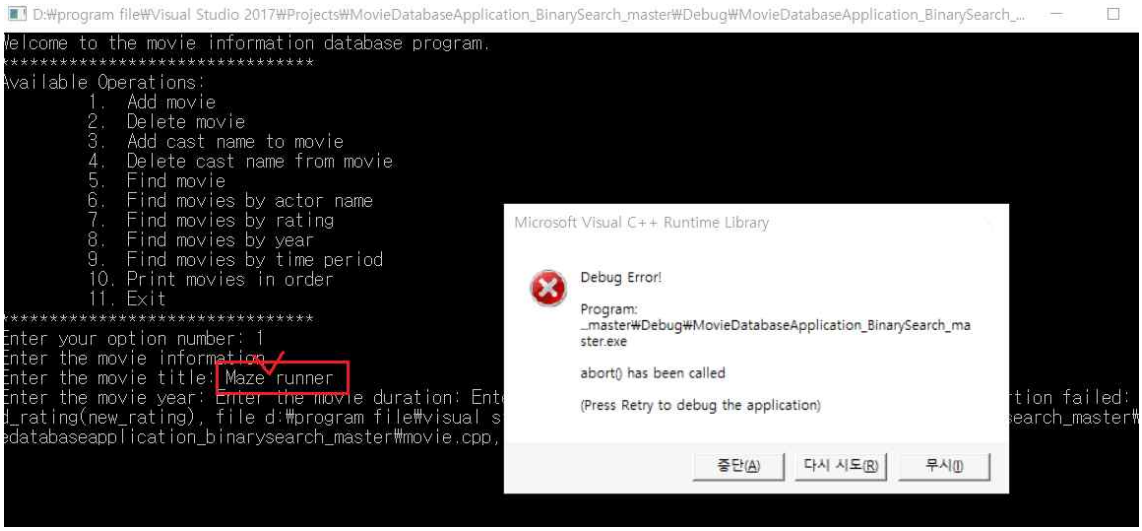
위와 같이 'Enter the cast name list: "다음에 바로 cast name을 입력해도 'Movie was successfully added'라는 문장이 출력된다. 그러나 'Movie Info'에는 Movie cast에 대한 정보가 아무것도 없음을 확인 할 수 있다. 이 문제는 코드확인을 통해 해결할 수 있었다.

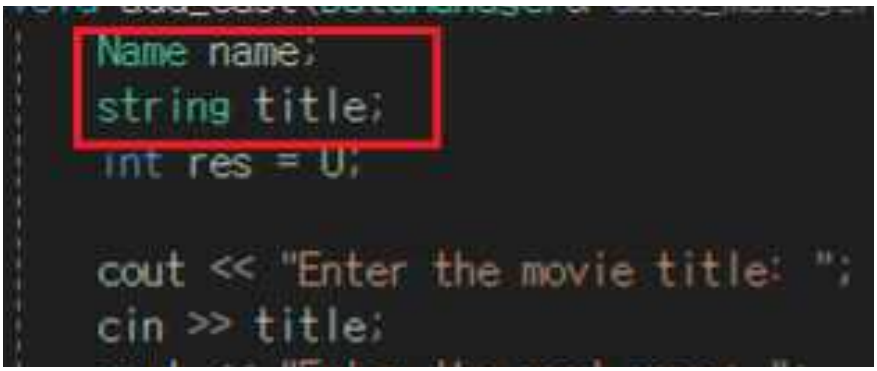


```
istream& operator>>(istream& in, NameList& L) {
    int len = 0, i = 0;
    Name next;

    // clear the name list
    L.names.make_empty();
    // read the number of names
    in >> len;
    for (i = 0; i < len; ++i) {
        // read next name
        in >> next;
        // add it into the name list
        L.add(next);
    }
    return in;
}
```

위와 같이 이름에 대한 정보를 입력하기 전에 len값을 먼저 입력하도록 코드를 작성했음을 확인할 수 있다. 따라서 입력할 영화배우의 수를 먼저 입력하고, 그 다음에 두 단어로 이루어진 배우이름을 입력해주면 문제가 해결된다.

나) Add movie

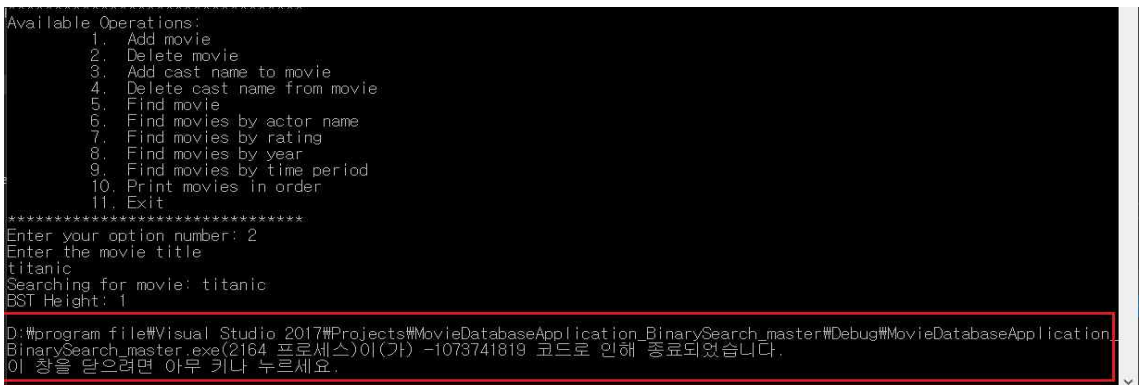


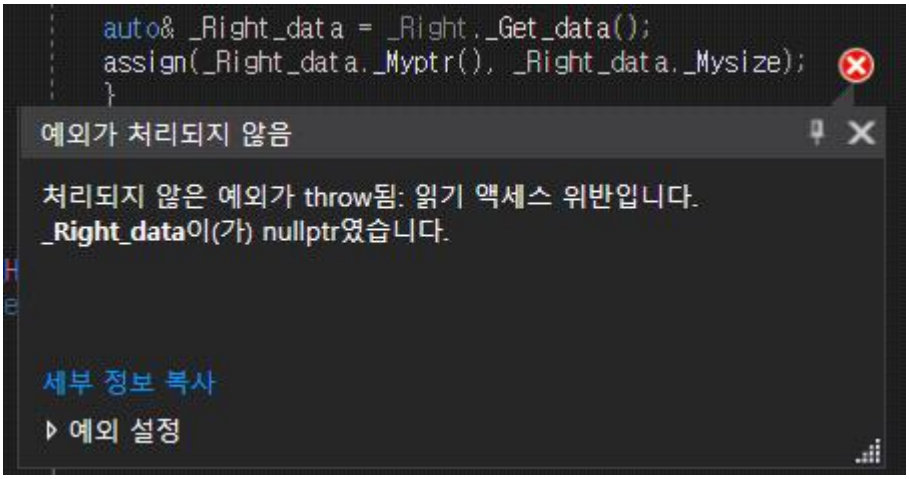


title 입력 시 띄워 쓰기를 하면 위와 같이 Error가 발생함을 확인할 수 있다. 배우 이름의 경우 Name type으로 이루어져 있어 first name과 last name 즉, 두 단어를 입력할 수 있다. 그러나 title의 경우 string type이기 때문에 띄워 쓰기 없이 입력을 해줘야 문제없이 실행된다.

다) Delete movie

아래는 'titanic'이란 제목의 영화 하나만 저장된 상태에서 Delete movie를 실행시킨 결과이다.





Delete movie시 Database에 저장된 movie가 하나인 경우 위와 같은 error가 발생할 수 있다. 따라서 Delete movie는 두 개 이상의 영화가 저장된 상태에서 실행시켜줘야 한다.

2) Code상 error 개선

프로젝트에서는 여러 가지 key에 따라 영화를 찾아주는 'Find movie'기능을 제공한다. 이와 관련하여 계획서상의 알고리즘과 최종 알고리즘을 비교해보았다.

- find_movies_of

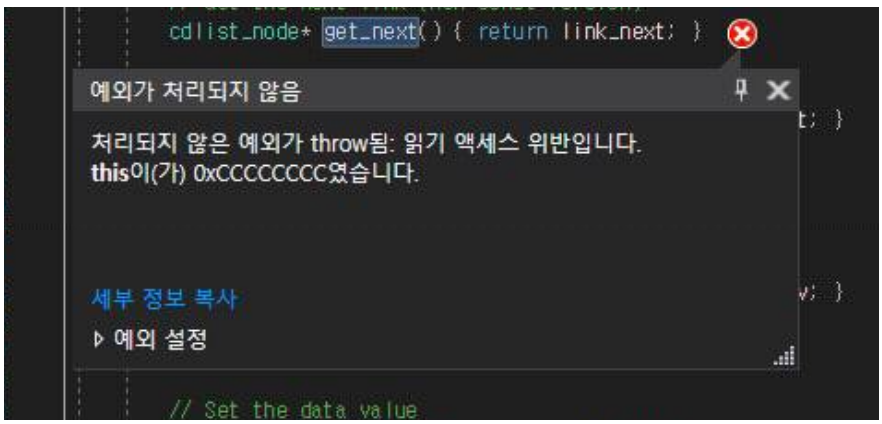
```
void find_movies_of(DataManager& data_manager) {
    Name name;
    List<Movie> L;

    cout << "Enter the actor name: ";
    cin >> name;
    cout << "Finding movies by actor: " << name << endl;
    L = data_manager.find_movies_of(name);
    if (L.is_empty()) {
        cout << "No movies found by actor: " << name << endl;
    }
    else {
        cout << "List of movies by actor: " << name << endl;
        L.print_forward(cout);
    }
}
```

moviedb.cpp

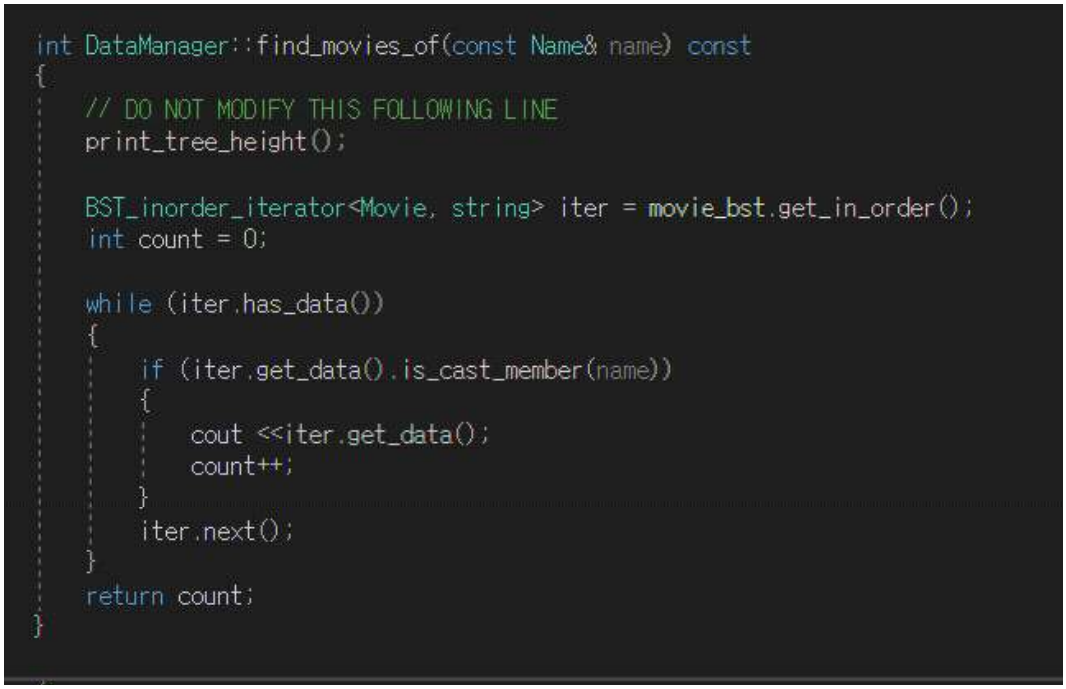
```
*/
List<Movie> DataManager::find_movies_of(const Name& name) const {
    // DO NOT MODIFY THIS FOLLOWING LINE
    print_tree_height();

    List<Movie> result;
    BST_inorder_iterator<Movie, string> iter = movie_bst.get_in_order();
    while (iter.has_data())
    {
        if (iter.get_data().is_cast_member(name))
        {
            result.insert(iter.get_data());
        }
        iter.next();
    }
    return result;
}
```

DataManage.cpp

계획서 작성 당시에는 특정 key를 가지고 find한 영화에 대한 정보를 List<Movie> type의 변수에 삽입하고자 했다. 그리고 조건에 부합하는 모든 영화를 List<Movie> type의 result변수에 삽입한 후 이를 List<Movie> type의 L이라는 변수에 전달하여 출력해주고자 하였다.



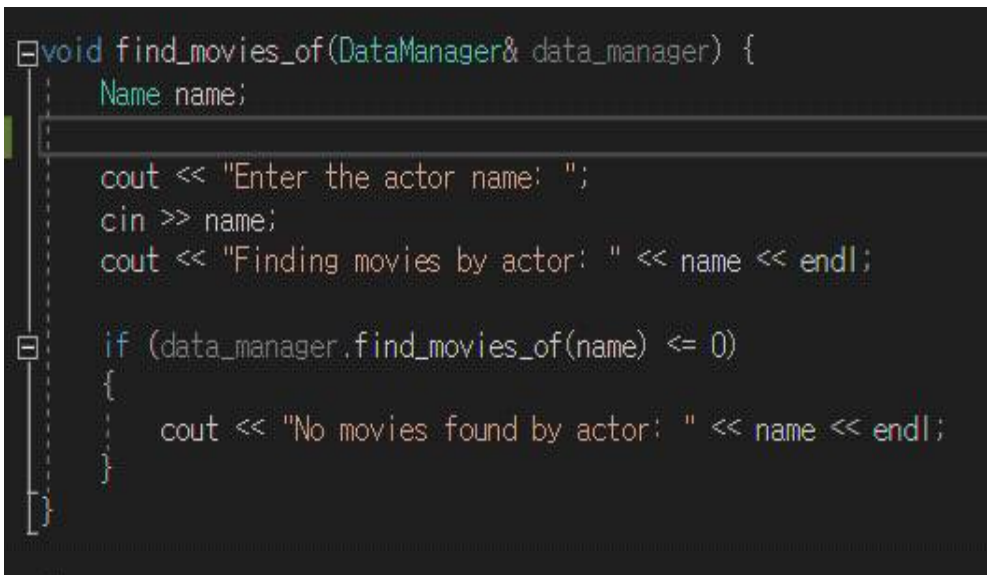
그러나 이런 식으로 find를 구현하였을 때는 위와 같은 error가 해결되지 않았다. 디버깅 결과 List<Movie> type의 변수에 insert하는 과정에서 문제가 있음을 알 수 있었다. 이에 List<Movie> type변수에 insert를 하지 않고도 find를 할 수 있도록 새로운 알고리즘을 떠올려 기능을 구현해보았다.



```
int DataManager::find_movies_of(const Name& name) const
{
    // DO NOT MODIFY THIS FOLLOWING LINE
    print_tree_height();

    BST_inorder_iterator<Movie, string> iter = movie_bst.get_in_order();
    int count = 0;

    while (iter.has_data())
    {
        if (iter.get_data().is_cast_member(name))
        {
            cout <<iter.get_data();
            count++;
        }
        iter.next();
    }
    return count;
}
```

DataManage.cpp

이 알고리즘의 경우 우선, 'DataManage.cpp'에서는 특정 key에 부합하는 영화가 find될 때마다 int type의 count에 +1을 해준다. 그리고 이번에는 error발생을 막고자, find된 영화에 대한 정보를 insert없이 바로 출력해주도록 코드를 수정하였다. 그리고 find된 영화가 없으면 0, 있으면 1이상의 값이 반환된다.



```
void find_movies_of(DataManager& data_manager) {
    Name name;

    cout << "Enter the actor name: ";
    cin >> name;
    cout << "Finding movies by actor: " << name << endl;

    if (data_manager.find_movies_of(name) <= 0)
    {
        cout << "No movies found by actor: " << name << endl;
    }
}
```

moviedb.cpp

'moviedb.cpp'에서는 find_moives_of(name) 라인을 if문에 넣어주었다. 이때 return

값이 0(find된 영화없음)이면 "No movies found by actor..."라는 문장이 출력될 것이고, return값이 0보다 크면 find_movies_of(name)함수가 실행되어 find가 이루어질 때마다 그 영화에 대한 정보가 출력된다.

아래와 같이 rating, year, period를 통해 영화를 find하는 함수도 위와 같은 알고리즘을 토대로 성공적으로 구현할 수 있었다.

- find_movies_rating

가) 계획서상의 코드

```
void find_movies_rating(DataManager& data_manager) {
    string rating;
    List<Movie> L;

    cout << "Enter the movie rating: ";
    cin >> rating;
    cout << "Finding movies rated: " << rating << endl;
    L = data_manager.find_movies_rated(rating);
    if (L.is_empty()) {
        cout << "No movies found for rating: " << rating << endl;
    }
    else {
        cout << "List of movies for rating: " << rating << endl;
        L.print_forward(cout);
    }
}
```

```
List<Movie> DataManager::find_movies_rated(const string& rating) const {
    // DO NOT MODIFY THIS FOLLOWING LINE
    print_tree_height();


    List<Movie> result;
    BST_inorder_iterator<Movie, string> iter = movie_bst.get_in_order();
    while (iter.has_data())
    {
        if (iter.get_data().get_rating() == rating)
        {
            result.insert(iter.get_data());
        }
        iter.next();
    }
    return result;
}
```

나) 수정된 코드

```
void find_movies_rating(DataManager& data_manager) {
    string rating;

    cout << "Enter the movie rating: ";
    cin >> rating;
    cout << "Finding movies rated: " << rating << endl;


    if (data_manager.find_movies_rated(rating) <= 0)
    {
        cout << "No movies found by actor: " << rating << endl;
    }
}
```

```
*/
int DataManager::find_movies_rated(const string& rating) const {
    // DO NOT MODIFY THIS FOLLOWING LINE
    print_tree_height();

    BST_inorder_iterator<Movie, string> iter = movie_bst.get_in_order();
    int count = 0;

    while (iter.has_data())
    {
        if (iter.get_data().get_rating() == rating)
        {
            cout << iter.get_data();
            count++;
        }
        iter.next();
    }

    return count;
}
```

- find_movies_year

가) 계획서상의 코드

```
void find_movies_year(DataManager& data_manager) {
    int year = 0;
    List<Movie> L;

    cout << "Enter the movie year: ";
    cin >> year;
    cout << "Finding movies made in the year: " << year << endl;
    L = data_manager.find_movies_released(year);
    if (L.is_empty()) {
        cout << "No movies found for year: " << year << endl;
    }
    else {
        cout << "List of movies for year: " << year << endl;
        L.print_forward(cout);
    }
}
```

```
List<Movie> DataManager::find_movies_released(int year) const {
    // DO NOT MODIFY THIS FOLLOWING LINE
    print_tree_height();


    List<Movie> result;
    BST_inorder_iterator<Movie, string> iter = movie_bst.get_in_order();
    while (iter.has_data())
    {
        if (iter.get_data().get_year() == year)
        {
            result.insert(iter.get_data());
        }
        iter.next();
    }
    return result;
}
```

나) 수정된 코드

```
void find_movies_year(DataManager& data_manager) {
    int year = 0;

    cout << "Enter the movie year: ";
    cin >> year;
    cout << "Finding movies made in the year: " << year << endl;

    if (data_manager.find_movies_released(year) <= 0)
    {
        cout << "No movies found by actor: " << year << endl;
    }
}
```

```
int DataManager::find_movies_released(int year) const {
    // DO NOT MODIFY THIS FOLLOWING LINE
    print_tree_height();

    BST_inorder_iterator<Movie, string> iter = movie_bst.get_in_order();
    int count = 0;

    while (iter.has_data())
    {
        if (iter.get_data().get_year() == year)
        {
            cout << iter.get_data() << endl;
            count++;
        }
        iter.next();
    }
    return count;
}
```

- find_movies_period

가) 계획서상의 코드

```
void find_movies_period(DataManager& data_manager) {
    int year1 = 0, year2 = 0;
    List<Movie> L;

    cout << "Enter the first year: ";
    cin >> year1;
    cout << "Enter the second year: ";
    cin >> year2;
    cout << "Finding movies maded between the years " << year1 <<
        " and " << year2 << endl;

    L = data_manager.find_movies_between(year1, year2);
    if (L.is_empty()) {
        cout << "No movies found between the years " << year1 <<
            "and " << year2 << "." << endl;
    }
    else {
        cout << "List of movies released between years " << year1 <<
            " and " << year2 << "." << endl;
        L.print_forward(cout);
    }
}
```

```
List<Movie> DataManager::find_movies_between(int year1, int year2) const {
    // DO NOT MODIFY THIS FOLLOWING LINE
    print_tree_height();


    List<Movie> result;
    BST_inorder_iterator<Movie, string> iter = movie_bst.get_in_order();
    while (iter.has_data())
    {
        if (year1 <= iter.get_data().get_year() && iter.get_data().get_year() <= year2)
        {
            result.insert(iter.get_data());
        }
        iter.next();
    }
    return result;
}
```

나) 수정된 코드

```
void find_movies_period(DataManager& data_manager) {
    int year1 = 0, year2 = 0;

    cout << "Enter the first year: ";
    cin >> year1;
    cout << "Enter the second year: ";
    cin >> year2;
    cout << "Finding movies maded between the years " << year1 <<
        " and " << year2 << endl;

    if (data_manager.find_movies_between(year1, year2) <= 0)
    {
        cout << "No movies found between the years " << year1 <<
            " and " << year2 << "." << endl;
    }
}
```

```
int DataManager::find_movies_between(int year1, int year2) const {
    // DO NOT MODIFY THIS FOLLOWING LINE
    print_tree_height();

    BST_inorder_iterator<Movie, string> iter = movie_bst.get_in_order();
    int count = 0;

    while (iter.has_data())
    {
        if (year1 <= iter.get_data().get_year() && iter.get_data().get_year() <= year2)
        {
            cout << iter.get_data();
            count++;
        }
        iter.next();
    }
    return count;
}
```

3) 계획 단계에서는 최종적으로 application을 만들어 프로젝트를 완성시키고자 하였다. 그러나 이를 실현하기에는 앱 프로그래밍에 대한 지식이 부족하다고 판단하였다. 따라서 웹 프로그래밍을 활용하여 프로젝트를 완성시켜보았다.

D. 자료구조

- class

1) "cdlist_node.h"

: 이 파일은 class cdlist_node의 선언을 포함한다. 그리고 이 파일에 활용된 자료구조와 함수들은 class List_Iterator, class List, class queue에서 활용된다.

(1) class cdlist_node

이 class는 circular-doubly linked list의 node를 나타낸다. 또한 Sorted circular doubly linked list를 구현한 함수도 포함하고 있다. 이때 node는 제약조건 아래에서 삽입되거나 삭제되며 이로써 특정 순서가 유지된다.

template parameter Item은 node에 저장될 object의 type이다. 각 Item의 상대적인 순서를 결정하기 위해 node에 저장될 object의 type은 relational comparison operator를 오버로드한 것으로 가정한다.

2) "Name.h"

: 이 파일은 name data type의 선언을 포함한다.

이 파일의 내용은 class NameList, class DataManager에서 활용된다.

3) "List.h" - #inlcude "cdlist_node.h"

: 이 파일은 List container class에 대한 interface 선언을 포함하고 있다. 이 List는

Circular Doubly Linked List toolkit과 관련된 코드를 사용하여 Sorted Circular Doubly Linked List로 구현된다.
이 파일에 활용된 자료구조와 함수들은 class Name, class NameList에서 활용된다.

(1) class List_Iterator
class List_Iterator는 List container class에 대한 external iterator를 구현한다. 이 iterator는 리스트의 다른 노드들에 대해 access, traverse할 때 사용된다.
Item type 변수는 리스트에 저장될 element들의 type을 나타낸다.

(2) class List
이는 Sorted Circular Doubly Linked List로 구현된 List container class이다. Item type parameter는 리스트에 저장될 element들의 type을 나타낸다.

4) "NameList.h" - #inlcude "Name.h", #inlcude "List.h"
: 이 파일에는 list of names에 관한 interface 선언이 포함되어있으며, name을 저장하기 위해 Sorted Doubly Linked List class container가 사용된다.
이 파일에 활용된 자료구조와 함수들은 class Movie, class DataManager에서 활용된다.

(1) class NameList
이 class는 name의 Sorted List를 구현하기위해 사용된다. name의 저장을 위해서는 Sorted Doubly Linked List class container가 사용된다.
이 class는 default constructor가 Linked list에 대해 destructor를 올바르게 호출하고, 이 경우 List내의 element를 삭제해야 하므로 destructor가 구현되지 않는다.

5) "Movie.h" - #include "NmaeList.h"
이 파일은 class Movie에서 구현되는 movie data type의 interface선언이 포함되어있다. 이 파일에 활용된 자료구조와 함수들은 class DataManager에서 활용된다.
(1) class Movie
이 class는 movie data type을 구현하는데 사용되는 class이다.
default destructor가 각 data member의 destructor를 올바르게 호출하고, NameList field에서 element를 삭제하기 때문에 이 class는 destructor를 구현하지 않는다.

6) "queue.h" - #inlcude "cdlist_node.h"
이 파일은 queue container class를 위한 interface의 선언을 포함한다. 이 class는 circular linked list toolkit으로 구현된다. 그러나 큐 자체는 circular도, sorted 형태도

아니다. 이 파일에 활용된 자료구조와 함수들은 class BST_Inorder_Iterator, class DataManager에서 활용된다.

(1) class queue

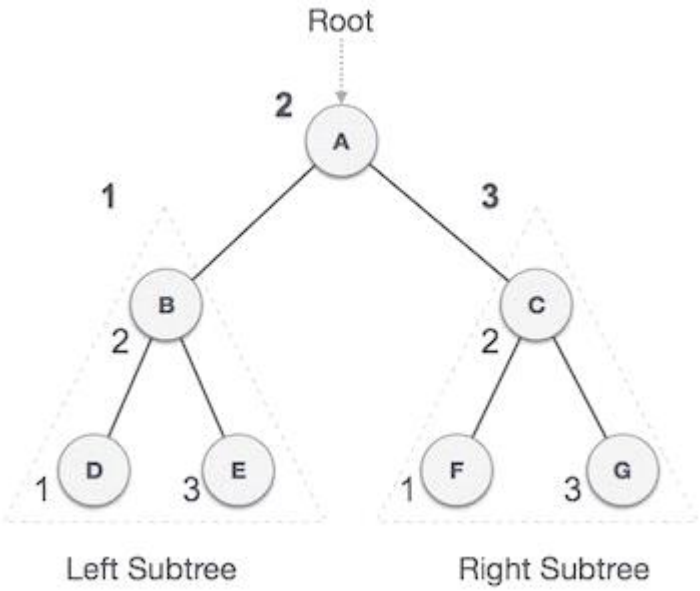
queue class container이다. 이 class는 queue data structure를 구현한다. Item Type parameter는 queue에 저장될 데이터 타입을 나타낸다.

7) "BinarySearchTree.h" - #inlcude "queue.h"

이 파일은 Binary Search Tree(BST)를 위한 interface 선언을 포함한다.

(1) class BST_inorder_iterator

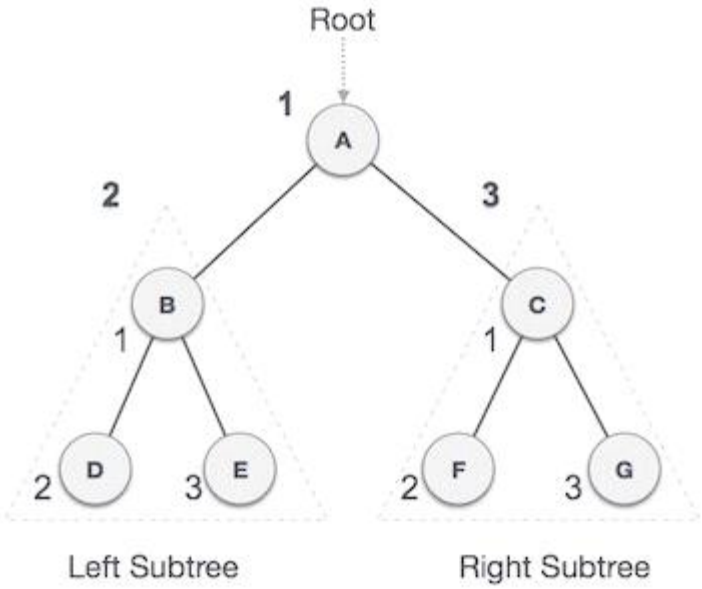
Binary Search Tree In-order Iterator class는 binary search tree의 inorder tree traversal을 수행하기위해 사용된다. BSTData는 search tree에 저장되는 data를 나타낸다. 그리고 DataKey type parameter는 tree에 저장될 object의 key를 나타낸다.



<Inorder Traversal>

(2) class BST_preorder_iterator

Binary Search Tree Pre-order Iterator class는 binary search tree의 preorder tree traversal을 수행하는데 사용된다. BSTData는 search tree에 저장될 data를 나타낸다. DataKey type parameter는 tree에 저장될 object의 key를 나타낸다.

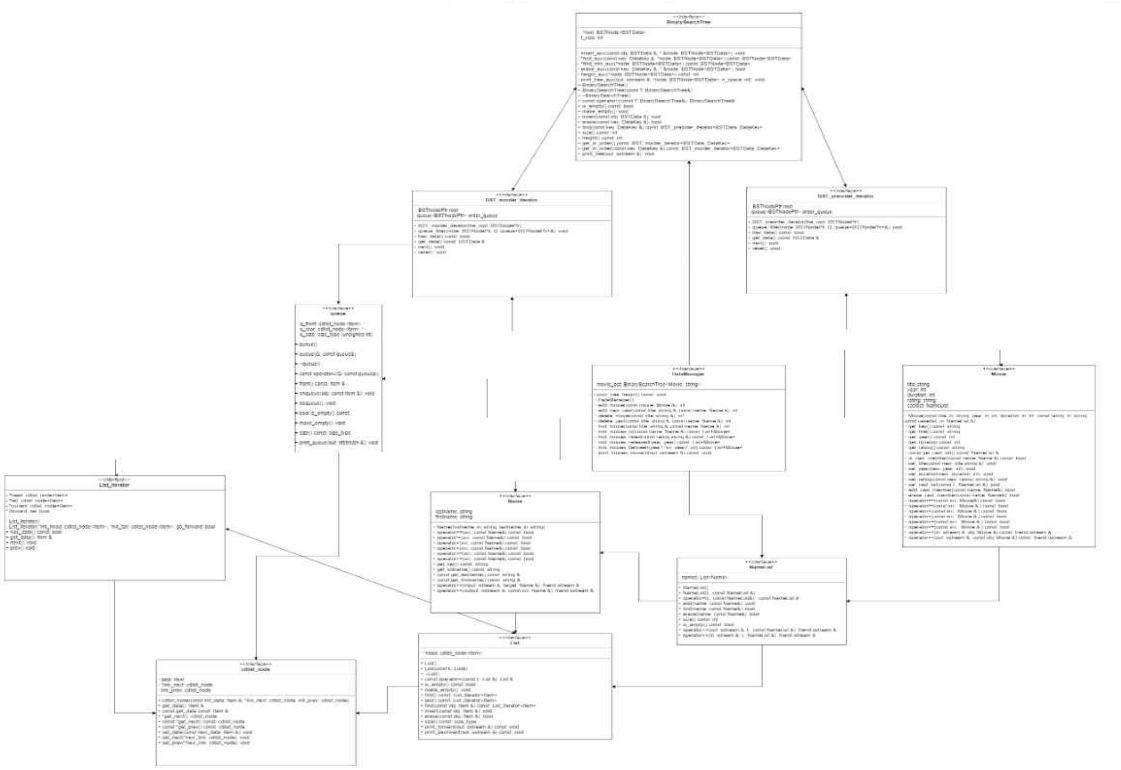


<preorder traversal>

(3) class BinarySearchTree

Binary Search Tree class이다. 여기서 BSTData는 search tree에 저장될 data를 나타낸다. 그리고 DataKey type parameter는 tree에 저장될 object의 key를 나타낸다.

- 클래스 다이어그램 (png파일을 첨부하였습니다.)



- 적절한 자료구조 선택과정

1) 다루는 자료의 삽입, 삭제 패턴은 어떤 것 인가?

⇨ 다루는 자료 중 Queue는 가장 먼저 들어온 것이 가장먼저 나가는 FIFO의 패턴을 가진다.

List는 삽입과 삭제 시에도 순서가 유지되는 sorted형태의 패턴을 가진다.

Tree의 경우, recursive하게 traverse하며 삽입 및 삭제가 진행된다.

1-1) 리스트, 스택, 큐 등 중에서 어느 것을 사용할까?

⇨ 프로젝트 내에서 사용된 자료구조는 '리스트, 스택, 큐' 중에서 고르자면 List와 Queue가 사용되었다.

2) 사용하려는 자료구조는 삽입, 삭제, 검색 중 어느 것에 효율적인가?

⇨ Doubly Linked List는 검색보다는 삽입 및 삭제 시에 유리한 자료구조이다.

Movie의 전체적인 정보(ex) Add movie)를 관리할 때는 BST를 사용하였다. 그러나 수정 및 삭제가 빈번하게 이루어져야 하는 경우(ex) Add cast name)에는 Sorted Linked List를 활용하였다.

3) 프로젝트의 환경은 동적인 환경(삽입/삭제가 빈번하게 발생) 또는 정적인 환경인가? 여기에 적합한 자료구조를 사용하는가?

⇨ 동적인 환경

'Movie Database Management Service'은 위에서도 언급했듯 데이터의 삽입과 삭제가 빈번하게 발생하는 동적인 환경이다. 따라서 이에 적합한 Linked list 자료구조를 사용하였다. 특히 프로젝트에서 Add cast name, delete cast name과 같은 기능 구현에 Linked list 자료구조가 사용되었다.

4) 리스트가 적합할까? 이진탐색트리가 적합할까?

⇨ 이진탐색트리

이진탐색트리는 이진탐색(Binary Search)와 연결리스트(Linked list)를 결합한 자료구조의 일종이다. 이진탐색트리의 경우 균형이 잡혀있는 한, 각 항목에 대한 검색경로가 연결리스트 보다 훨씬 짧다. (삽입, 삭제, 검색의 평균 시간 복잡도: $LogN$) 따라서 효율적인 탐색이 가능하다는 장점이 있다.

'Movie Database Management Service'의 경우 영화이름, 배우이름, 등급, 개봉년도, 상영시간이라는 key에 대한 탐색기능을 제공한다. 이에 Find기능을 구현하는 데에는 연결리스트보다, 이진탐색트리를 사용하는 게 더 적합하다고 판단하였다.

5) 배열을 사용한 리스트가 적합할 까? 연결 구조를 사용한 리스트가 적합할까?

⇨ 연결구조

배열의 경우 index를 안다면 데이터 접근속도가 굉장히 빠르다는 장점이 있다. 그러나 데이터의 삽입이나 삭제 시, 삽입 및 삭제가 이루어진 위치의 다음부터 모든 데이터의 위치를 변경해야 한다는 단점이 있다.

반면에 연결리스트의 경우 데이터에 대한 접근속도는 배열보다 느리다. 그러나 데이터의 삽입 및 삭제 시에 주소만 바꾸어 주면 되기 때문에 배열보다 훨씬 용이하게 쓸 수 있다. 따라서 Add cast name과 같은 기능 구현 시에 연결 구조 리스트를 활용하였다.

6) Linked list가 적합할까? Doubly Linked list가 적합할까?

⇨ Doubly linked list

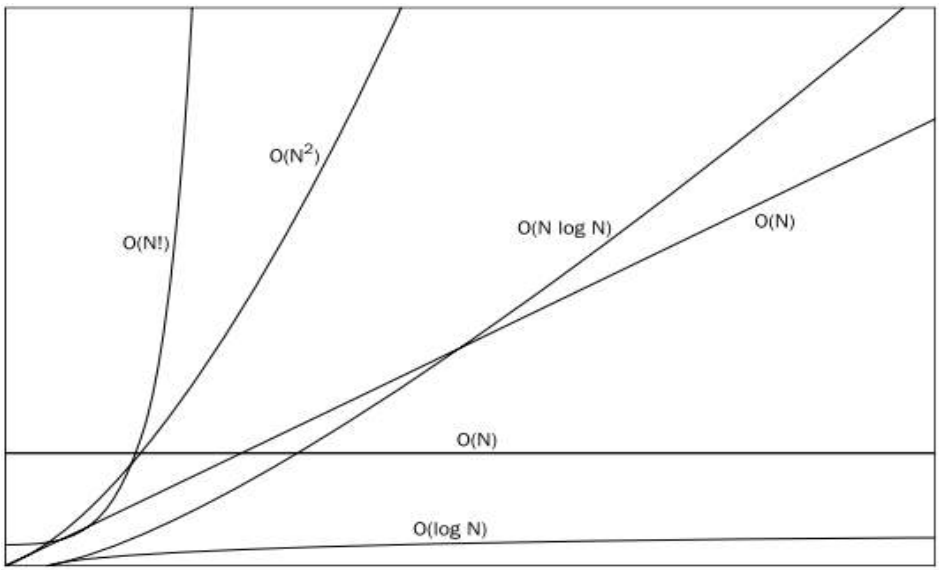
일반적인 Linked list는 다음 노드의 탐색만 가능하지만, Doubly linked list는 두 개의 포인터가 있어 앞뒤로 탐색이 가능하다. 따라서 일반적인 Linked list에 비해 효율적인 탐색이 가능하고, 최대 탐색시간을 반으로 줄일 수 있다. 이에 프로젝트 내에서 Doubly Linked list를 사용하여 코드를 구현하였다.

7) Linked list가 적합할까? Array가 적합할까?

데이터양이 많고, 데이터 접근이 빈번하게 이루어지며, 삽입 삭제가 거의 없는 경우는 배열을 사용하는 것이 좋다. 반면에 데이터의 양이 많지 않고, 데이터의 삽입 및 삭제가 빈번하게 이루어질 경우에는 연결리스트를 사용하는 것이 좋다.

'Movie Database Management Service'의 경우 영화이름의 삽입, 삭제, 영화배우 이름의 삽입, 삭제 등의 기능을 통해 삽입과 삭제가 빈번하게 발생한다. 그러나 개인에 의해 비교적 적은양의 정보가 입력되기 때문에 연결리스트가 적합하다고 판단하였다.

- 복잡도



복잡도는 공간과 시간으로 나뉠 수 있다. 공간 복잡도를 통해서는 알고리즘에 필요한 메모리 공간이 얼마인지를 계산할 수 있다. 시간 복잡도는 알고리즘이 동작하는 시간이 얼마인지 계산할 수 있게 해준다.
메모리의 경우, 충분히 보충할 수 있으나 시간은 추가할 수 없는 절대적인 가치를 가지고 있기 때문에 시간 복잡도가 더 우선시 된다.[3)]
이에 내가 사용한 자료구조의 시간 복잡도에 대해 분석해보았다.

1) Queue
1. 삽입: O(1)
2. 삭제: O(1)

2) Linked List(doubly, circular, sorted)
1. 삽입
1.1. 리스트에 노드가 없을 때: O(1)
1.2 노드가 하나 이상일 때: O(N)
1.3 tail에 새로운 노드 추가할 때: O(N)
1.4 그 외의 위치에 새 노드를 삽입할 때: O(N)
2. 삭제
2.1 노드가 1개 일 때: O(1)
2.2 노드가 2개 이상일 때: O(N)
2.3 tail노드를 삭제할 때: O(N)
2.4 그 외의 위치에서 삭제할 때: O(N)
3. 탐색: O(N)

3) Binary Search Tree
1. 삽입: O($\log N$)
2. 삭제: O($\log N$)
3. 탐색: O($\log N$)
4. 최악의 경우(Skewed) 시간 복잡도는 O(N)이 될 수 있다.

3. 결론 및 작품후기

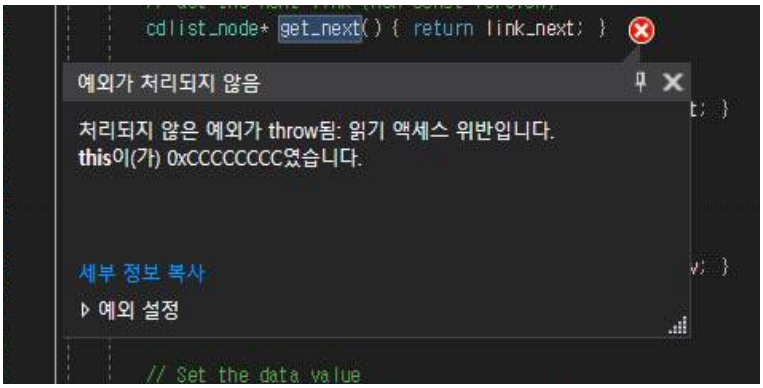
크게 프로젝트 주제선정, 자료구조에 대한 이해, 기능 구현 및 error 해결, interface 구현의 순서로 프로젝트를 진행하였다. 우선, 주제선정 시에는 독창적이면서도 다양한 자료구조를 활용할 수 있을 만한 프로젝트를 구상해야 했다. 그러나 프로젝트 구상 당시에는 tree와 같은 고급 자료구조에 대한 지식이 없었다. 따라서

3) https://grepsean.github.io/Algorithms-and-Data-Structures-with-Python-1/

어떤 주제를 선택해야 고급 자료구조를 적절히 사용할 수 있을지 판단이 잘 서지 않았다. 그런 만큼 주제선정 자체도 쉽지 만은 않았다. 그래서 tree라는 자료구조에 대해 스스로 간략하게 공부를 한 뒤, 'Movie database application'이란 주제를 선정하게 되었다. 이후에는 수업을 통해 고급 자료구조에 대해 좀 더 깊이 있는 공부를 할 수 있었다.

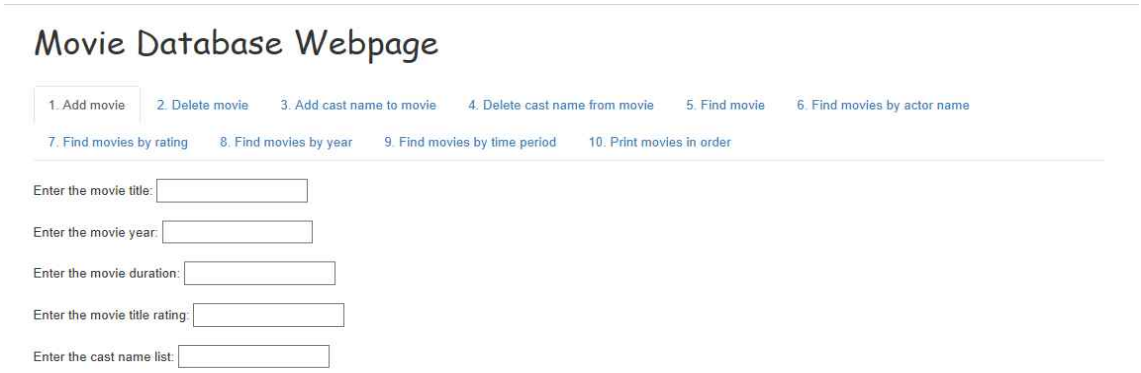
다음으로는 이렇게 얻은 지식을 기반으로 여러 가지 기능들을 구현해보았다. 그러나 '3개 이상의 자료구조 사용', '하나이상의 고급 자료구조 사용'과 같은 조건에 맞게 코드를 작성하는 것도 쉽지만은 않았다. 또한 내가 사용한 자료구조가 같은 기능을 수행할 수 있는 다른 자료구조보다 효율적인지도 판단할 수 있어야 했다. 처음에는 정말 막막했지만, 시간이 지날수록 어떤 자료구조를 언제 사용해야하는지에 대한 감이 조금씩 생겼다.

그리고 이렇게 코드를 작성해나가며 실습에는 보지 못했던 다양한 error들도 경험해볼 수 있었다. 디버깅, 인터넷 검색, text book등을 활용하면 해결되는 문제들도 많았지만, 그렇지 않은 경우도 종종 있었다.



그 중에서도 위와 같이 '예외가 처리되지 않음'이라는 error를 해결하는데 가장 오랜 시간이 걸렸다. 처음에는 계획서 상에 언급한 알고리즘을 최대한 유지한 채로 error를 해결해보려 했으나 잘 되지 않았다. 그래서 새로운 알고리즘을 떠올려 코드를 작성해보았고, 디버깅을 잘 활용하여 안정적인 코드를 작성할 수 있었다. 생각보다 오랜 시간이 소요되어 정말 힘들었지만, 그만큼 많은 것을 배울 수 있었던 과정이었다.

이렇게 코드를 다 완성시킨 후에는 web programming을 통해 interface를 구현해보았다.



<Web page>

물론 처음 계획대로 application으로 구현해보았다면 더욱 좋았겠지만, 현재 내가 가진 앱 프로그래밍에 대한 지식으로는 구현이 어렵다고 판단하였다. 따라서 현실적으로 내가 가진 웹 프로그래밍 지식을 활용하여 interface를 구현해보았다. 웹 프로그래밍도 이번학기에 처음 공부하게 된 분야라 아직은 많이 서툴다. 그러나 그동안 공부한 내용을 프로젝트와 접목시켜보며, 그 과정에서 또 새로운 것들을 배울 수 있었다. 그동안 프로젝트를 진행하며 정말 힘들기도 했지만, 이렇게 결과물을 보니 뿌듯하고, 의미 있는 경험이었다는 생각이 든다. 또한 프로그래밍 실력도 저번학기에 비해 훨씬 성장시킬 수 있었던 것 같다.