

# CSE3013 컴퓨터공학설계및실험I: 테트리스 프로젝트

담당교수: 서강대학교 컴퓨터공학과 김승욱

전공: 컴퓨터공학과

학년: 2학년

학번: 20171635

이름: 박상리

## 1. 설계 문제 및 목표

ncurses 라이브러리가 제공하는 리눅스 터미널 상에서의 GUI를 이용하여 테트리스 게임 프로그램을 제작하고 여기에 랭킹 시스템과 블록 배치 추천 기능을 추가 구현한다.

테트리스 게임은, 블록을 90도씩 반 시계 방향으로 회전하거나 세 방향(좌, 우, 하)으로 움직여 필드에 쌓으면, 빈틈없이 채워진 줄은 지워지고 그에 따른 스코어를 얻어 결과적으로 가장 높은 최종 스코어를 얻는 것이 목적이다.

프로그램을 실행하면 사용자는 메뉴를 선택할 수 있다. 1번을 선택하면 테트리스 게임을 플레이할 수 있고, 2번을 선택하면 랭킹 정보를 확인할 수 있고, 3번을 선택하면 블록 배치 추천 기능을 따라 플레이 되는 모드로 테트리스 게임이 실행되고, 마지막으로 4번을 선택하면 프로그램이 종료된다.

### 1) 1주차

1주차 실습/과제에서는 테트리스 게임의 기능을 구현하는 것이 주 실습 내용이다. **play** 함수와 **blockdown** 함수를 구현하는 것이 게임을 구현하는데 있어서 가장 중요한 기능이라고 할 수 있다. 1주차에서 구현할 내용은 아래와 같다.

- ① 1초마다 블록이 떨어지며, 입력 받은 키에 따라 블록이 알맞게 움직여야 한다.
- ② 블록이 떨어져 바닥, 혹은 다른 블록에 닿게 될 경우 닿는 면적에 따라 점수를 추가해야 한다.
- ③ 한 줄의 블록이 채워질 경우 해당 블록을 삭제하고 사라진 줄 수에 맞추어 점수를 추가하며 사라진 줄 수에 맞게 다른 블록들을 아래로 옮겨주어야 한다.
- ④ 블록이 떨어질 위치를 그림자를 이용해 필드에서 보여주어야 한다.

### 2) 2주차

2주차에는 랭킹 시스템을 구현하는 것이 주 내용이다. 구현할 내용은 아래와 같다.

- ① 랭킹의 정보를 담고 있는 rank.txt를 불러온다.
- ② 불러온 자료를 이용해 점수가 큰 순서부터 작은 순서로 자료구조에 저장한다.

- ③ 입력 받은 범위의 랭킹을 출력, 입력 받은 이름의 사람을 검색, 입력 받은 랭킹의 사람을 삭제하는 기능을 구현한다.
- ④ 테트리스 게임 종료 시(중도 종료가 아닌 게임 오버 때) 이름을 입력 받아 랭킹을 데이터에 추가한다.
- ⑤ 프로그램 종료 시 rank.txt에 저장된 랭킹을 새로 작성한다.

### 3) 3주차

3주차에는 블록의 위치를 추천하는 프로그램과 그 프로그램을 이용하는 자동 플레이 기능을 구현한다. 구현해야 할 기능은 아래와 같다.

- ① 현재 블록이 쌓일 수 있는 모든 경우의 수를 계산하고, 그 다음 블록과 그 다음 블록이 쌓일 모든 경우의 수를 계산하여 최고의 점수를 내는 경우의 위치를 추천한다.
- ② 블록이 추천되는 위치의 표시는 'R'로 표시한다.
- ③ 자동플레이의 경우엔 블록이 바로 추천되는 위치로 옮겨지도록 한다.
- ④ 교재에 추천된 방법보다 더 효율적인 방법을 생각해서 추천 위치를 구상한다.

## 1.1 현실적 제한조건

- 게임이 실제 이미지가 아닌 문자로 구현 되어있다. 블록이나 필드의 틀, 등등 실제로 게임을 만들 때 시중의 테트리스 게임처럼 게임의 그래픽을 보기 좋게 만들기가 어렵다. 또한 사용하는 언어가 c라는 점에서, 게임을 만드는 것에 쉽게 이용되는 언어가 아닌 만큼 다른 언어로 게임을 만드는 데에 있어서 원하는 기능을 구현할 때 어려울 수 있다.
- 이 테트리스 프로그램은 ncurses 라이브러리가 사용 가능한 특정 환경에서만 실행될 수 있다. 예를 들어 ncurses 라이브러리가 없는 윈도우 환경에서는 이 프로그램을 실행할 수가 없다.
- 추천기능을 구현할 때 이론적으로는 가장 좋은 점수를 내는 최고의 위치를 추천한다. 해당 기능을 작동시키는 원리는 어렵지 않지만, 이를 실제 자료구조와 알고리즘으로 구현하는 것은 쉽지 않다. 또한 이를 기능을 실제로 실행했을 때 추천되는 블록의 위치가 항상 최선의 위치는 아니다. 이런 단점을 개선하기 위해 탐색의 범위를 넓힌다면 시간/공간 복잡도가 너무 커져서 게임을 플레이 하는데 있어서 버퍼링이 너무 커진다. 때문에 이를 고려한 최적화된 추천 기능을 구현해야 한다.

## 2. 요구사항

## 2.1 설계 목표 설정

	목표	기능
1주차	테트리스 게임 구현	① 블록 자동 낙하 ② 입력에 맞춘 블록 이동 (이동, 회전) ③ 게임 스코어 필드 다음 블록 2개 출력 ④ 게임 점수 증가 (닿는 면적, 사라지는 줄) ⑤ 필드 블록 쌓기 ⑥ 블록 낙하 위치 예측
2주차	랭킹 시스템 구현	① rank.txt를 자료구조에 저장 ② 입력 받은 범위에 맞추어 랭킹 출력 ③ 입력 받은 이름의 사용자/점수 출력 ④ 입력 받은 랭킹 삭제 ⑤ 게임 플레이 후 생긴 점수 저장
3주차	추천 시스템 구현	① 실제 게임 플레이 시 위치를 추천 ② 자동 추천되는 위치에 블록을 쌓는 플레이 기능 구현 ③ 추천되는 위치에 'R'표시 ④ 추천 기능 구현 시 트리 구조 이용 ⑤ 자동 추천 플레이 기능 구현 시 개선된 추천 기능 구현

## 2.2 합성

### I. Play()에 필요한 자료구조/알고리즘

- ① field: 게임이 플레이 되면서 쌓이는 블록을 필드로 표현한다. field는 HEIGHT\*WIDTH 크기의 배열이며, 각 배열에 값이 0이면 블록이 없는 것이고 1이면 블록이 있는 것이다.
- ② block: 블록의 모양과 정보는 전역변수로 저장되어 있다. 블록 id와 블록 회전에 따라 4\*4, 2차원 배열로 저장되어 있다.

### II. Rank()에 필요한 자료구조/알고리즘

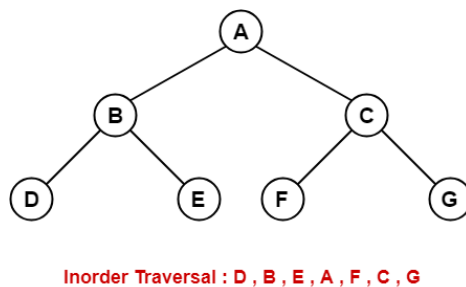
### ① Binary Search Tree

Rank를 불러오고 저장하는 데에 있어서 조금 더 시간/공간 복잡도를 효율적으로 이용하기 위해서 binary search tree를 이용하였다. 점수를 효율적으로 출력하고 저장하기 위해서 기존의 binary tree search와는 조금 다른 자료구조를 지닌다. Linked list보다 더 효율적으로 랭킹을 불러올 수 있다. struct는 아래와 같이 구성되어 있다.

```
typedef struct _Node{
    char name[NAMELEN];
    int score;
    struct _Node* leftChild;
    struct _Node* rightChild;
}Node;
```

### ② Inorder Traversal

Binary Search Tree에서 자료를 큰 데이터부터 작은 데이터로 출력하려면 inorder 탐색을 이용해야 한다. 방문 순서는 아래의 그림과 같다.



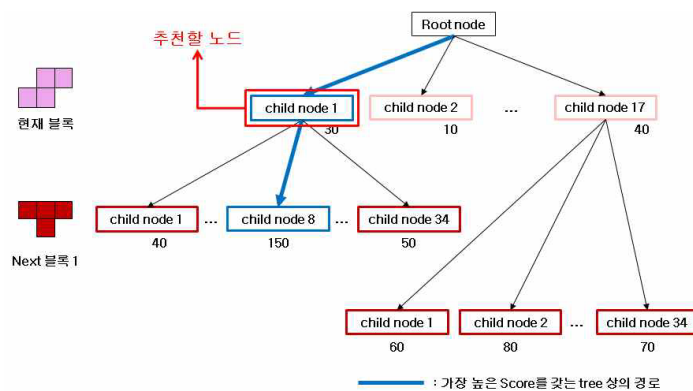
## III. RecommendPlay()에 필요한 자료구조/알고리즘

### ① Tree

Tree 구조를 이용하여 추천하는 위치를 설계하도록 제작하였다. 현재의 블록을 root node라고 하면, 그 다음 블록이 놓일 위치를 계산한 것이 child 노드이다. 기존의 프로그램은 play()에서 교재에 있는 방식으로 recommend()를 이용하여 RecNode 구조체를 이용한 트리를 형성하여 위치를 추천하지만, 마지막에 프로그램을 구현하면서 아래의 MdfRec 구조만을 이용하여 프로그램이 동작하도록 설계되었다. 두개의 구조체 모두 구현될 당시 아래의 그림과 같은 트리 구조를 이용했다. 개선된 추천 모드가 더 효율적이라고 생각하여 개선된 추천 모드만을 이용하게 만들었다.

```
typedef struct _MdfRec{
    int lv, score;
    char f[HEIGHT][WIDTH];
    int x, y, r;
    struct _MdfRec **child;
} MdfRec;
```

```
typedef struct _RecNode{
    int lv, score;
    char f[HEIGHT][WIDTH];
    int x, y, r;
    struct _RecNode *c[CHILDREN_MAX];
} RecNode;
```



## ② Greedy 알고리즘

개선된 추천모드에서는 greedy 알고리즘을 이용한다. 점수가 높은 경우만을 탐색하는 형식으로 작동하기 때문이다. 모든 child 노드 중에서 딱 중간의 값을 찾아내어 그 값보다 결과값이 작다면, 그 이후의 노드는 탐색하지 않는 식의 방식을 이용하여 개선된 추천 모드를 만들고자 하였다.

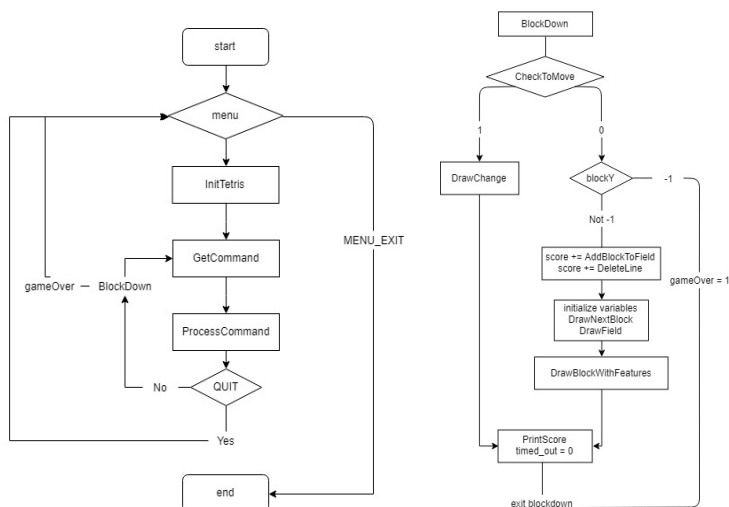
## 2.3 분석

### I. 1주차

```
void play()
{
    InitTetris()
    do:
        BlockDown()
        command = GetCommand()
        if(command == QUIT) gameOver = 1
    while(!gameOver)
}
```

1주차에는 게임의 기본적인 기능을 구현했으며 게임의 대부분의 기능 작동은 **BlockDown** 함수에서 이루어 지기 때문에 **play()** 함수의 시간 복잡도는 **BlockDown**과 같다고 할 수 있다.

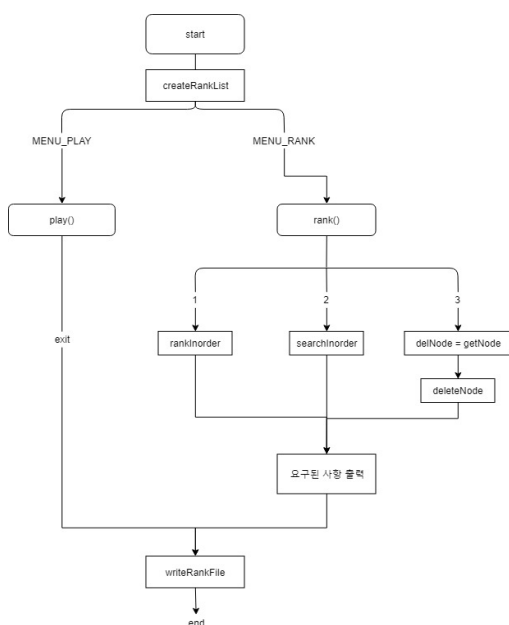
**play()**와 **BlockDown**을 순서대로 그리면 아래의 그림과 같이 그릴 수 있다.



**BlockDown** 함수의 주 역할은 게임이 작동하도록 역할에 맞는 다른 함수들을 호출한다. 그에 따라 내부에서 호출되는 함수들의 시간 복잡도를 살펴보면, **CheckToMove** 함수의 시간 복잡도는 블록의 크기에 따라 바뀌기 때문에 블록 크기의 변수인 **BLOCK\_HEIGHT**와 **BLOCK\_WIDTH**에 따라서  $O(\text{BLOCK\_HEIGHT} * \text{BLOCK\_WIDTH})$ 이다. **AddBlockToField** 함수나 **DrawChange** 함수의 시간 복잡도도 **CheckToMove** 함수처럼 블록의 크기에 따라 바뀌기 때문에 같은 이유로  $O(\text{BLOCK\_HEIGHT} * \text{BLOCK\_WIDTH})$ 이다.

**DeleteLine** 함수의 시간 복잡도는 시간 복잡도가 가장 클 경우, 길이가 가장 긴 블록의 크기만큼의 시간 복잡도를 갖게 된다. 이때 길이가 가장 긴 블록은 4이므로 상수가 되기 때문에 필드 전체를 탐색하는 경우의 시간 복잡도를 고려하면 된다. 그러므로  $O(\text{HEIGHT} * \text{WIDTH})$ 만큼의 시간 복잡도를 지닌다. 결국 **BlockDown** 함수의 시간 복잡도는 가장 큰 시간 복잡도를 지닌 **DeleteLine**의 시간 복잡도,  $O(\text{HEIGHT} * \text{WIDTH})$ 라고 할 수 있다. 공간 복잡도의 경우, 이 함수는 **field**와 생성하는 **nextblock**들의 공간이 필요하므로  $O(\text{HEIGHT} * \text{WIDTH})$ 라고 할 수 있다.

## II. 2주차



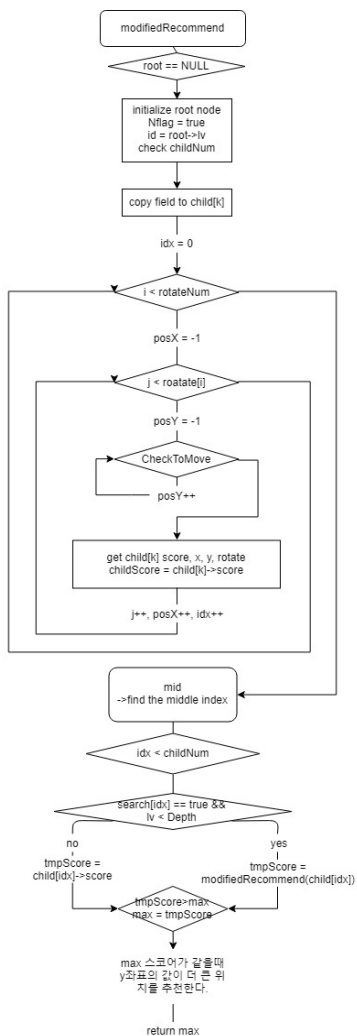
**createRankList** 함수는 **rank**를 위한 자료구조를 만들기 위해 호출되는 함수이다. 이 함수는 **binary search tree**를 만들기 위해 **insertNode** 함수를 호출하는데 이때 트리 구조를 만드는 데 걸리는 시간 복잡도는 트리의 깊이 **h**와 같다. 그러므로 시간 복잡도는  $O(h)$ 이다. **rankInorder** 함수의 경우 트리를 **inorder traversal**로 탐색한다. 이때 최악의 경우 모든 노드를 방문해야 하기 때문에 시간 복잡도는  $O(n)$ 이며 **n**은 데이터의 개수이다.

`searchInorder` 함수와 `getNode` 함수도 마찬가지로  $O(n)$ 의 시간 복잡도를 갖고 있다.

`deleteNode` 함수는 일단 `getNode` 함수가 지정해준 노드를 삭제하기 때문에 이후의 과정은 삭제된 노드의 자식들을 어디에 연결하는 가인데, 이 트리를 연결하는 최악의 경우에 트리의 가장 마지막 노드에 자식들을 연결하는 경우가 최악의 경우이므로 시간 복잡도는  $O(h)$ 라고 할 수 있다.

이에 따라 1번의 경우  $O(h)$ , 2번의 경우  $O(n)$ , 3번의 경우  $O(n)$ 의 시간 복잡도를 지닌다고 할 수 있다. 공간 복잡도의 경우  $n$ 개의 노드를 제외하고는 더 이상의 공간이 필요하지 않으므로  $O(n)$ 이다.

### III. 3주차



`recommend play`의 경우 탐색할 깊이의 값으로 설정한 `VISIBLE_BLOCKS`라는 변수의 값에 따라서 시간 복잡도와 공간 복잡도가 바뀐다. `root`의 `child`의 개수가 얼마인가에 따라서 탐색해야 할 범위가 지정되는데, 최악의 경우 `child`의 개수가 34개이다. 즉 최악의 경우 시간 복잡도는  $O(34^{VISIBLE\_BLOCKS})$ 이다. 하지만 `child`의 개수가 34개인 경우의 블록은 한가지 밖에 없고, `modifiedRecommend`의 경우 탐색하는 트리의 개수가 줄어들기 때문에 실제로 프로그램이 작동될 때 최악의 시간 복잡도에 도달하는 경우가 흔하지 않다.

공간 복잡도의 경우, `recommended play`가 실행될 때 최악의 경우  $O(34^{VISIBLE\_BLOCKS} * sizeof(MdfRec))$ 의 크기이다. 다만 시간 복잡도의 설명에서 마찬가지로 실제 프로그램 실행 시 그만큼의 공간을 사용하지 않는다.

`mid`라는 함수는 `child`의 `score`를 정렬해서 정렬된 값의 중간 인덱스의 값보다 작으면 탐색을 하지 않도록 해주는 함수이다. 이때 정렬하는 과정에서 버블 정렬을 사용했기 때문에 `mid`의 시간 복잡도는  $O(n^2)$ 이다. 공간 복잡도는  $O(n)$ 이다.

## 2.4 제작

### I. play

- **CheckToMove**

2중 반복문을 이용하여 블록의 좌표가 지정된 **HEIGHT**와 **WIDTH**를 넘어가면 **0**을 반환하여 움직일 수 없다는 것을 나타낸다. 움직일 수 있을 경우 **1**을 반환한다.

- **DrawChange**

어떠한 키를 입력 받았는지를 고려하여 움직일 수 있는지 없는지를 **check to move** 함수를 이용하여 확인하고 입력 받은 키에 따른 움직임에 남는 흔적들을 지운다. 이전에 블록이 있던 위치에 다시 필드에 해당하는 문자를 출력하고 움직인 위치에 블록을 출력할 수 있도록 한다.

- **AddBlockToField**

블록을 필드로 추가하는 함수이다. 2중 반복문을 이용해서 필드에 안착한 블록의 값을 현재 필드 값으로 옮겨준다. 또한 점수 계산을 위해서 블록이 닿은 면적을 계산하여 이후 **score**를 계산할 때 **touched**라는 변수를 이용하여 점수를 추가한다.

- **DeleteLine**

필드 전체를 이중 반복문을 이용하여 탐색하며 가득 채워진 줄은 삭제하고 점수를 추가하며 남아있는 필드를 사라진 줄 수에 맞추어 내리는 역할을 한다.

- **BlockDown**

위의 2.3에서 그린 순서도에서 보이듯이 블록을 아래로 내리는 역할이지만, 이 기능을 위해서 다른 함수들을 호출하는 역할을 한다. **CheckToMove** 함수를 이용하여 블록이 아래로 내려갈 수 있는지를 확인하고 아래로 내려갈 수 있다면 **blockY**를 증가시킨다. 만약 그렇지 않다면 **blockY**의 값은 **-1**로 게임이 종료된다.

게임이 종료되지 않은 경우 **score**를 **AddBlockToField**와 **DeleteLine** 함수를 호출하여 점수를 계산할 수 있도록 한다. 점수가 추가된 후, 현재 생성되어 있는 블록들 다음의 블록을 생성하고 각 변수들을 초기화 한 뒤 블록과 그림자를 그려줄 수 있도록 한다.

- **DrawBlockWithFeatures**

**DrawShadow**, **DrawBlock** 등의 함수들을 호출하여 게임 플레이 하면서 그리는 블록들을 한번에 묶어주는 역할을 한다.

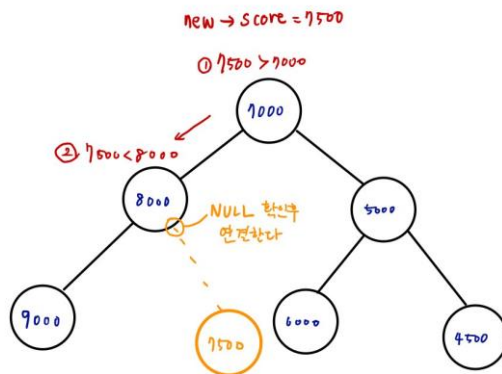
- **DrawShadow**

현재 블록의 위치에서 아래로 떨어질 위치를 보여주는 함수이다. 반복문을 이용하여 블록이 더 이상 내려갈 수 없는 위치를 찾아내어 해당 위치에 그림자를 출력한다.



## II. rank

- **createRankList**  
rank.txt에서 데이터 정보를 읽어온다. 그 다음에 **insertNode** 함수를 호출하여 **binary search** 트리의 자료구조를 만든다.
- **newRank**  
게임 종료 후 새로운 데이터를 랭크의 자료구조로 추가하는 역할을 한다. 마찬가지로 **insertNode** 함수를 이용하여 트리에 자료를 추가한다.
- **createNode**  
트리의 노드를 만드는 함수다. **malloc**을 이용하여 데이터를 저장할 공간을 만들고 데이터를 노드에 저장한다.
- **insertNode**  
**Binary search tree**를 만드는 데에 있어 가장 핵심적인 역할을 하는 함수이다.

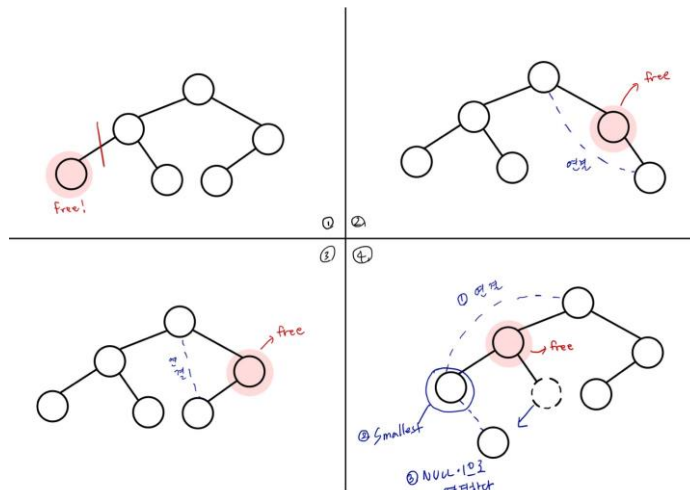


위의 그림의 순서로 작동하게 된다. **root** 노드와 값을 비교하며 이동하게 된 노드가 **NULL**이 아니라면 다시 **score** 값을 비교하게 된다. 값을 비교해서 추가되려는 점수가 크다면 왼쪽으로, 작다면 오른쪽으로 움직이게 되며 **NULL**값을 발견하게 되면 해당 부분에 새로운 노드를 연결하도록 한다. 노드를 이동하면서, 재귀적으로 **insertNode** 함수를 호출하게 된다.

- **getNode**  
삭제할 노드의 값과 해당 타깃의 부모 노드의 값도 반환한다. **inorder** 탐색을 이용하여 해당하는 랭크의 노드를 찾아내서 삭제할 수 있도록 한다.
- **deleteNode**  
**getNode** 함수가 삭제할 **target**을 가져오면 **target**에 연결되어 있는 노드들을 다시 트리에 알맞게 연결해주는 역할을 한다. 노드를 삭제할 때는 아래의 4가지의 경우로 나누어서 삭제하게 된다.

- ① **target**의 자식이 모두 **NULL**일 때
- ② **target**의 왼쪽 자식만 **NULL**일 때
- ③ **target**의 오른쪽 자식만 **NULL**일 때
- ④ **target**의 양쪽 자식이 모두 **NULL**이 아닐 때

위의 4가지 경우를 **if**문과 **else if** 문을 이용하여 구분한다. 4가지 경우를 그림으로 그려 표현하면 아래와 같다.



**deleteNode**는 삭제되는 데이터에 맞추어 전체 데이터의 개수의 변수인 **NumData**도 1 감소시킨다.

- **printInorder/searchInorder**

두 함수 모두 **inorder traversal**로 트리를 탐색하며, **print**는 해당하는 랭크 범위의 데이터들을 출력하고 **search**는 해당하는 이름을 가진 노드의 점수와 이름을 출력한다. **inorder traversal**의 방문 순서는 위의 2.2에서 설명 되어있다.

### III. recommended play

- **recommend**

**recommend** 함수의 작동 원리는 위의 2.2에서 나와있는 그림과 같다. 함수의 작동 원리는 인자로 들어온 **root**의 **lv**값이 **nextblock**의 **lv+1** 번째 블록임을 먼저 확인한다. 그후 이 블록을 떨어뜨릴 **field**가 **root->field**, 추가되어야 할 점수는 **root->score**임을 확인한다. 반복문을 이용하여 이 블록을 떨어뜨릴 모든 위치를 고려하고 그 모든 경우에 대하여 점수를 계산한다. 이 노드들의 주소를 **root**의 **child** 배열에 넣을 수 있도록 한다. 이 **child** 배열에는 해당하는 위치의 **rotation**, **x**, **y**좌표의 값을 집어넣고 **root->lv**이 0일 때 **recommendR**, **X**, **Y**에 저장하도록 한다. **recommendR**, **X**, **Y**는 추천 위치가 나타날 좌표의 값과 블록의 모양이다.

- **modifiedRecommend**

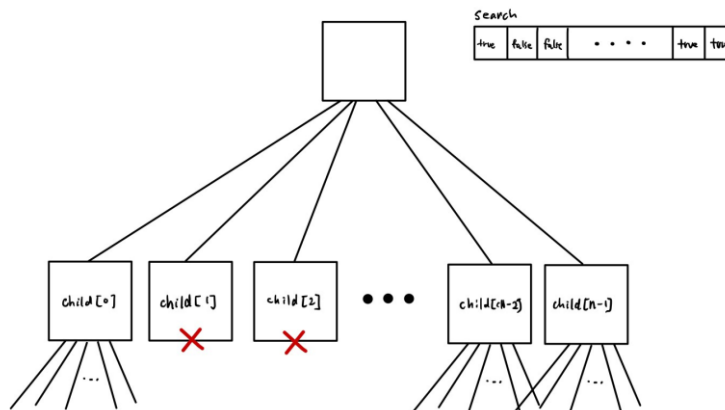
이 함수는 위의 **recommend**를 조금 더 개선한 추천 함수이다. 2.2에서 나온 그림처럼 트리의 구조로 뻗어 나가며 **max** 점수를 계산한다. 이때 이 함수에서 기존의 **recommend** 보다 개선된 점은 3가지이다.

- ① **malloc**을 이용하여 **child**를 만들어 메모리를 절약한다.
- ② 탐색하는 **child**의 수를 줄인다.
- ③ **level 0**에서 **max**의 값이 같을 경우 가장 낮은 위치를 추천한다.

1번은 위의 2.2에서 구조체를 확인하면 다른 메모리를 절약하는 것을 확인할 수 있다.

2번의 경우에는 아래의 **mid** 함수를 이용하여 탐색하는 경우의 수를 줄일 수 있다.

이 함수에는 **cScore**와 **search** 배열이 있다. 두 배열 모두 **flag**의 역할을 하기에 **bool** 자료형으로 선언되어 있다. **cScore**에는 해당하는 인덱스의 점수, **search**는 해당 인덱스의 **child**를 탐색할지 하지 않을지를 결정한다. **search**가 **true**인지 **false**인지에 따라서 아래의 그림과 같이 탐색 여부가 줄어들게 된다.

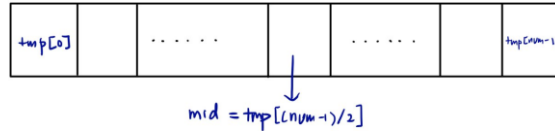


3번의 경우 **root->lv**이 0이 아닌 경우에는 항상 최대의 값을 갖는 경우를 저장한다. 하지만 **recommendR**, **X**, **Y**를 지정하는 레벨이 0인 경우에는 가장 값이 큰 **recommendY**를 찾아야한다. 과제로 제출했던 코드에서는 이전에 **recommendR**, **X**, **Y**가 저장되었는지만 판단하면 될 것이라 생각하고 **maxFlag**를 지정해서 방문여부만 확인하고 이전의 **recommendY**와 값을 비교하였다. 하지만 이런 경우에는 **max**스코어 값에 상관없이 값이 큰 **recommendY**만 탐색하게 된다. 그래서 이후에 추가적으로 코드를 더 추가하였다.

**maxFlag**를 이용하여 이전에 **recommend**좌표 값들을 지정했는지를 확인한다. 그리고 **max**의 값이 같은 경우에서 만약 이전의 **recommendY**의 값이 더 크다면 값을 갱신하지

않고, 그 반대라면 `recommendY`의 값을 갱신한다. 이 점을 개선하면 블록이 한자리에 반복해서 쌓이는 경우를 방지하게 되며, 높이가 낮게 블록이 쌓여 게임을 조금 더 오래 지속할 수 있다.

- `mid`



```
if(cScore[i] < mid) s[i] = false;
else s[i] = true;
```

버블 정렬을 이용하여 `score`를 새로운 배열 `tmp`에 삽입하여 순서를 정한다. 정렬된 인덱스들의 중간 인덱스 값에 해당하는 `score`보다 값이 작은 `child`는 탐색하지 않을 수 있도록 표시하는 함수이다.

- `displayTime`

추천 함수가 작동하는데 걸리는 시간을 계산하여 화면상에 보여주는 함수이다.

`duration`이라는 변수로 계산하는데 걸리는 시간, 점수를 `duration`으로 나누어 프로그램의 효율을 화면에 표시할 수 있도록 한다.

## 2.5 시험

- 1) `play()`

1) 게임 플레이	2) 게임 오버	3) 점수 등록

- 1) 실제로 게임을 플레이하는 화면이다. 다음 블록과 그 다음블록을 디스플레이 한다.
- 2) 블록이 쌓이다가 더 이상 쌓일 수 없게 되면서 게임이 종료된다.

3) 게임 종료 후 랭크에 정보를 등록한다.

2) rank

랭크 실행

```
1. list ranks from X to Y
2. list ranks by a specific name
3. delete a specific rank
```

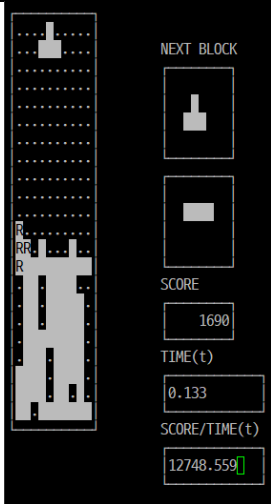
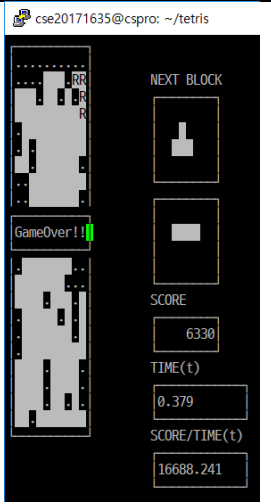
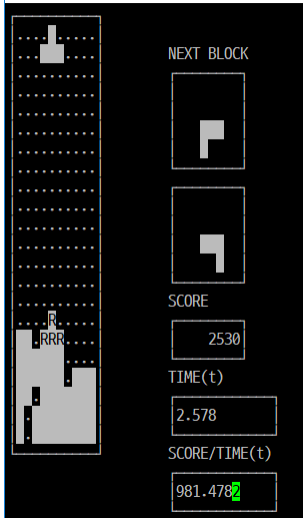
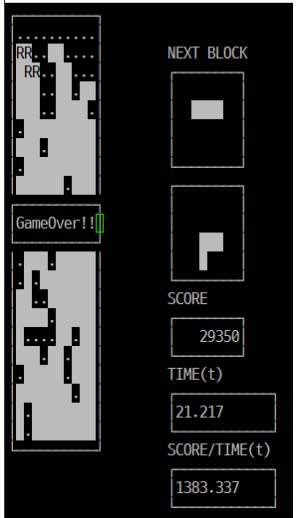
2번 랭크를 눌렀을 때 가장 먼저 뜨는 화면이다.

1) 랭크 나열	2) 랭크 범위 선택	3) 이름 검색
<pre>1. list ranks from X to Y 2. list ranks by a specific name 3. delete a specific rank X: Y:  name        score ----- ----- Jacob        20340 Park         17990 Jacob        620 Flake        500 Smith        300 2.5test      260 Pikachu      150 Raccoon      130 Coffee       110 Pizza        100</pre>	<pre>1. list ranks from X to Y 2. list ranks by a specific name 3. delete a specific rank X: 4 Y: 9  name        score ----- ----- Flake        500 Smith        300 2.5test      260 Pikachu      150 Raccoon      130 Coffee       110</pre>	<pre>1. list ranks from X to Y 2. list ranks by a specific name 3. delete a specific rank Input the name: 2.5test  name        score ----- ----- 2.5test      260</pre>
4) 랭크 삭제	5) 삭제 확인	6) Rank.txt 확인
<pre>1. list ranks from X to Y 2. list ranks by a specific name 3. delete a specific rank Input the rank: 6  name        score ----- ----- 2.5test      260  result: the rank deleted</pre>	<pre>1. list ranks from X to Y 2. list ranks by a specific name 3. delete a specific rank X: Y:  name        score ----- ----- Jacob        20340 Park         17990 Jacob        620 Flake        500 Smith        300 Pikachu      150 Raccoon      130 Coffee       110 Pizza        100</pre>	<pre>1 10 2 Jacob 20340 3 Park 17990 4 Jacob 620 5 Flake 500 6 Smith 300 7 2.5test 260 8 Pikachu 150 9 Raccoon 130 10 Coffee 110 11 Pizza 100</pre>
		<pre>1 9 2 Jacob 20340 3 Park 17990 4 Jacob 620 5 Flake 500 6 Smith 300 7 Pikachu 150 8 Raccoon 130 9 Coffee 110 10 Pizza 100</pre>

1) 범위를 입력하지 않았을 때, 처음부터 끝까지의 순위를 출력하는 화면이다. 위의 게임에서 추가된 이 용자의 데이터를 확인할 수 있다.

2) 랭크의 범위를 지정하여 점수를 출력한다.

- 3) 1번 게임에서 추가된 사용자를 검색한다.
  - 4) 6위인 사용자의 데이터를 삭제한다.
  - 5) 4번의 결과를 확인할 수 있다.
  - 6) 삭제전과 삭제후의 rank.txt의 모습이다.
- 3) recommend play

Depth\Status	실행 중	종료
<b>VISIBLE_BLOCKS = 3</b> <pre>#define VISIBLE_BLOCKS (1 + 2)</pre>		
<b>VISIBLE_BLOCKS = 4</b> <pre>#define VISIBLE_BLOCKS (1 + 3)</pre>		



Recommend play는 프로그램이 자동으로 가장 좋은 점수를 낼 수 있다고 생각 하는 위치에 블록을 쌓아주는 기능이다. 아래 Time은 추천 위치를 계산하는 데에 누적된 시간이고, score/time은 프로그램의 효율을 보여준다.

실제로 추천 기능의 깊이를 변경해서 프로그램을 돌린 결과 최종 점수에서 꽤 큰 차이가 있었다. Depth가 3인 경우에는 6330점을 획득하였고, 4인 경우에는 29350점을 획득하였다.

이후 보고서를 작성하다가 발견한 코드의 오류를 고친 후 VISIBLE\_BLOCKS = 3 인 경우에는 10630점, 4일땐 57610점이 나왔다.

## 2.6 평가

- 게임을 실제로 직접 플레이 해볼 때, 블록이 필드 밖에 출력되거나 점수가 올라가는데 있어서 오류는 없었다. 게임의 기본적인 플레이 기능도 잘 구현되었고 이후 실제 플레이에 recommended play를 추가하는 부분도 잘 구현되었다.
- 랭크 기능을 시험할 때도 게임 종료 후 점수가 추가될 때 올바른 순서로 추가되었고, rank.txt 에 결과 값도 제대로 추가되었다. 랭킹을 보여주는 기능도 오류 없이 잘 작동했고 랭킹을 삭제 한 이후에 랭킹을 보여주는 기능을 실행했을 때 랭크가 잘 삭제된 것을 확인할 수 있었다. recommend play에서는 depth가 3일때와 4일때의 점수의 차이점을 확실하게 확인할 수 있었다. 결과 코드를 제출한 이후 발견한 코드의 오류를 수정한 뒤 depth 3에서의 점수 차이는 그렇게 크지 않았었지만, depth 4에서는 꽤 점수차이가 큰 것을 발견할 수 있었다.
- recommend play가 어느 수준으로 작동하는지를 확인하기 위해서 실제로 게임을 플레이 해본 결과, 내가 플레이한 게임의 최대 점수는 평균적으로 약 2만점이었다. 이를 통해서 depth가 3 인 경우의 recommended 보다는 사람이 플레이 하는 경우가 낮고 수정된 코드의 depth 4로

recommended play를 실행할 경우에는 추천 프로그램의 기능이 나온 걸 발견할 수 있었다.

## 2.7 환경

학생들이 리눅스 서버를 접속하여 프로젝트를 진행하므로 해당 서버에 접속할 수 있는 데스크탑과 ssh 접속 프로그램을 제공한다. 접속하는 리눅스 서버에 각 학생들에게 하위 계정을 발급하여 할당 받는 용량에 한하여 자유롭게 이를 이용하여 프로젝트를 진행할 수 있는 환경을 제공한다.

## 2.8 미학

새로운 기능들을 추가한 이후에도 기존에 가장 처음 받았던 필드를 그리는 화면에서 모습이 크게 바뀌지 않았다. 아래의 요소들은 ncurses 라이브러리를 이용해 구현되었다.

- 1주차: 그림자를 추가하고, score와 nextblock과 그 다음 블록을 그려주는 기능이 추가되었다.
- 2주차: 1주차에 구현한 플레이에서 게임이 종료되면 이름을 입력 받아 시스템에 추가되도록 하는 기능을 구현했다. 게임 종료 시 your name: 이라는 화면이 뜨게 되어있다.
- 3주차: 자동 플레이를 실행할 때에도 다음 블록, 그 다음 블록 그리고 점수를 시간으로 나누어서 시간적 효율을 보여주는 것을 일관성 있게 박스를 그려서 표시하였다.

## 2.9 보건 및 안정

rank 기능의 경우 rank.txt에 있는 데이터의 수가 증가하여도 오류없이 프로그램이 잘 출력되는 것을 확인할 수 있었다. 게임을 평가하면서 실제로 플레이할 때 q나 방향키를 제외한 다른 키들을 입력했을 때에도 오류 없이 게임은 잘 작동하였다.

그 이외에 오류로 발생할 수 있는 경우는 depth가 깊어질 경우이다. 프로그램 컴파일시 **VISIBLE\_BLOCKS**의 값이 너무 커질 경우, 계산에 소요되는 시간이 너무 오래 걸려 플레이 하는데 지장을 줄 수 있다. 실제로 **VISIBLE\_BLOCKS**를 4로 지정했을 때까지는 값이 3일때와 체감상 크게 차이가 없었지만 5로 지정할 경우, 확실히 플레이하는 데에 추천 위치를 계산하는 동안 시간이 걸리기 때문에 영향을 주는 것을 확인할 수 있었다.

## 3. 기 타

### 3.1 환경 구성

gcc



```
cse20171635@cspro:~/tetris$ gcc --version
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.10) 5.4.0 20160609
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

유닉스 환경에서 많이 이용되는 c언어 컴파일러이다.

vi

```
cse20171635@cspro:~/tetris$ vi --version
VIM - Vi IMproved 7.4 (2013 Aug 10, compiled Nov 24 2016 16:44:48)
```

version 7.4  
April 1st, 06

## vi / vim 단축키 모음

**Esc**  
명령 모드

~ 대소문자 전환	! 외부 명령	@ 매크로 실행	# 백업 파일 열기	\$ 줄번호 입력	% 일치하는 줄 전체를	^ 줄의 첫 줄로	& * 반복	* 다음 줄로	( 문장 시작	) 문장 끝	아래줄로 이동	+ 다음 줄
~ 매크로 기록	1	2	3	4	5	6	7	8	9	0	출력 지움	= 자동 들여쓰기

**q** 실행 모드

q 대소문자 기록	W word	E word	R 수동으로	T 원본 텍스트	Y 줄 단위로 복사	U 줄 단위로 실행 취소	I 줄 시작에서 삽입	O 행 위에 삽입	P 현재 커서 위치에 붙여넣기	{ 문단 시작	} 문단 끝
A 줄의 끝에서 새 줄로 시작하여 편집	S 줄의 시작에서 새 줄로 시작하여 편집	D 줄의 끝에서 삭제	F 뒤로 검색	G 줄의 끝까지	H 화면 상단으로	J 한 줄 합치기	K 다음 줄로	L 화면 하단으로	. 행을 복사하여 다음 행에 붙이기	" 레지스터 지정	' 커서 이동

**Z** 줄을 종료

Z 줄을 종료	X 백스페이스	C 줄의 끝까지 삭제	V 줄 단위로 선택	B 이하의 단위로	N (다음) 단위로	M 화면 끝까지	< 맨 앞까지	> 맨 뒤까지	? 찾기 (뒤로)
Z 줄을 종료	X 백스페이스	C 바꿈	V 블록 모드	b 이하의 단위로	n (다음) 단위로	m 마크 설정	< 맨 앞까지	> 맨 뒤까지	? 찾기

**주요 명령행 명령 ('ex'):**

- w (저장), q (종료), q! (저장하지 않고 종료)
- se! (파일명 지정)
- :%s/x/y/g (파일 전체에서 'x'를 'y'로 교체), :b (vim 도움말), :new (새 파일)

**그외 중요한 명령들:**

- CTRL-R: 재실행 (vim)
- CTRL-F/R: 페이지 위로/아래로
- CTRL-E/R: 종 스크롤 위로/아래로
- CTRL-W: 윈도우를 이동 (vim 전용)

**비주얼 모드:**

- 커서를 움직여 지정된 범위에 연신자를 적용합니다. (vim 전용)

**참고:**

- (1) 복사/붙여넣기/지우기 명령어를 사용하기 전에 ">"를 입력하여 레지스터(클립보드)를 지정하세요. (x)에서 z 또는 + 을 사용할 수 있음 (예: "ay\$ 을 입력하면 현재 커서에서 라인 끝까지의 내용을 레지스터 'a'에 저장합니다.)
- (2) 어떤 명령을 입력하기 전에 횡수를 지정하면, 횡수만큼 번역하게 됩니다. (예: 2b, 4z, 5, 4d)
- (3) 연속으로 입력하는 명령은 현재의 라인에 변경됩니다. 예: :dd(현재 라인 지우기), >>(들여쓰기)
- (4) ZZ는 저장후 종료, ZZ는 저장하지 않고 종료.
- (5) zt: 커서가 위치한 곳을 제일위로 올리거나, zb: 바닥으로, zz: 가운데로
- (6) gg: 파일의 처음으로 (vim 전용), g\$ : 커서가 위치한 곳의 파일 끝(Vim 전용)

vi/vim 에 대한 더 많은 강좌나 팁을 얻으려면 [www.viemu.com](http://www.viemu.com) (ViEmu, MS 비주얼 스튜디오를 위한 vi/vim 에뮬레이션)을 방문하십시오.

유닉스 환경에서 많이 이용되는 텍스트 에디터로 다양한 플러그인을 이용하여 커스텀이 가능하다. 위의 단축키에 설명되어 있듯 vi [파일명]을 입력하고 i를 누르면 텍스트 편집이 가능하다.

gdb

```
cse20171635@cspro:~/tetris$ gdb --version
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
```

GNU 소프트웨어 시스템을 위한 기본 디버거로 gdb 실행을 위해서는 gcc -g -o [프로그램명] [소스파일명] 를 커맨드라인에 입력해야한다. 그 이후 gdb [프로그램명]을 통해서 gdb를 실행할 수 있다. b [line number]를 이용해 중단점을 설정할 수 있고 n을 입력하여 현재 행을 실행할 수 있다.

ncurses library:

텍스트모드에서 Window, Panel, Menu, Mouse, Color등을 쉽게 사용할 수 있도록 도와주는 라이브러리다. gcc 설치 시 자동으로 같이 설치된다. 이용하기 위해서는 <ncurses.h> 를 소스파일에 추가해야 하며 컴파일시 -lncurses 라는 옵션을 추가해야 한다. 윈도우에서 ncurses 라이브러리를 이용하여 컴파일할 수 없다.

<b>initscr()</b>	Curses 모드를 시작하며, curses를 사용하기 위해 반드시 사용해야 한다.
<b>endwin()</b>	Curses 모드를 종료하며 사용하지 않을 시에 텍스트모드에 이상이 생길 수 있다.
<b>echo(), noecho()</b>	사용자에게 입력 받은 문자의 출력 여부를 결정하는 함수이다. echo()는 출력, noecho()는 그 반대이다.
<b>getch()</b>	사용자로부터 키를 입력 받는다.
<b>move()</b>	해당 좌표의 위치로 커서를 이동한다. 화면의 범위를 벗어날 경우 segmentation fault가 발생할 수 있다.
<b>clear()</b>	화면 정보를 갱신한다.
<b>printw()</b>	윈도우에 화면을 출력한다.
<b>keypad()</b>	F1, F2, 방향키 등과 같은 특수한 키들을 사용할 수 있다. 표현은 KEY_F(1)~KEY_F(12), KEY_LEFT, KEY_RIGHT, KEY_UP, KEY_DOWN로 된다.
<b>attron(), attrset()</b>	글자에 특수한 효과를 준다. attron() 중첩 사용하면 중첩효과를 attrset()은 그 밑으로 모두 설정을 attrset()로 해준다.
<b>attroff()</b>	attron()이나 attrset()으로 준 효과를 끈다. 인자는 attron(), attrset()과 동일하다.
<b>color_start()</b>	color모드를 시작한다. 색깔을 넣고자 할 때 전에 반드시 써주어야 하는 함수이다. (has_colors()의 리턴값이 TRUE일때만 사용 가능하다.)
<b>refresh()</b>	화면에 찍은 내용을 갱신한다. curses 모드에서는 출력함수를 이용해 출력해도 refresh()를 쓰기 전까진 화면에 나타나지 않는다.

### 3.2 참고 사항

참고 사항 없음.

### 3.3 팀 구성

박상리 100%

### 3.4 수행기간

2018/11/10 ~ 2018/11/30