9주차 예비보고서

전공: 신문방송학과 학년: 3학년 학번: 20191150 이름: 전현길

1. 2주차 실습에 구현하는 랭킹 시스템에 대한 자료를 읽어보고, 이를 구현하기 위한 자료구조를 2가지 이상 생각한다.

구현해야 할 랭킹 시스템의 특징은 다음과 같다. 데이터는 **이름(char[]과 점수(int)**로 구성된다. rank.txt 파일을 읽어서 랭킹 정보를 불러오며, 게임 종료 시 사용자 이름을 입력받고 랭킹 정보를 자료구조에 추가한다. 프로그램 종료 시 랭킹 정보에 변화가 있다면 랭킹 목록을 탐색하며 rank.txt에 다시 기록한다.

추가적으로 원하는 범위(x위 ~ y위)의 랭킹 정보를 효율적으로 추출할 수 있어야 한다. 따라서 최댓값/최솟값만을 추출하는 힙 자료구조나 가장 먼저 들어왔던 데이터, 가장 나중에 들어왔던 데이터를 출력하는 스택/큐 자료구조 등은 비효율적이다. 그렇다면 **이진 탐색 트리, 배열, 링크드 리스트** 등을 고려할 수 있다.

점수 순으로 정렬된 **배열**의 경우 삽입, 삭제 자체에 O(n)의 시간 복잡도가 발생한다. 또 새로운 데이터가 입력될 위치를 찾기 위해서 선형 탐색을 활용한다면 O(n), 이진 탐색을 활용한다면 O(logn)의 시간 복잡도가 발생한다. 단 특정한 순위의 데이터를 탐색하는 데엔 O(1)의 시간 복잡도가 발생한다.

일반적으로, 배열은 최대 크기가 정해져 있기 때문에 메모리가 꽉 찰 때마다 더 큰 크기의 공간에 메모리를 동적 할당하고, 데이터를 복사하며, 기존 데이터를 해제하는 번거로운 과정을 거쳐야 한다. 단, 랭킹 시스템의 경우 특성상 프로그램이 실행될 때마다 파일에 적힌 데이터의 개수만큼 메모리를 할당하면 되고, 데이터의 삽입 역시 게임이 종료될 때 단 한 번만 이뤄지기 때문에 해당 과정을 수행할 필요 없이 데이터의 개수 + a의 메모리를 게임 실행 시 동적 할당하면 문제 없이 데이터를 삽입할 수 있다.

링크드 리스트의 경우 삽입, 삭제에는 O(1)의 시간 복잡도가 발생하지만, 데이터가 입력될 위치를 찾기 위해 O(n)의 시간 복잡도가 발생한다. 또한, 특정한 순위의 데이터를 탐색하는 데에도 역시 O(n)의 시간 복잡도가 발생한다. 단 배열처럼 최대 메모리 공간을 고려할 필요 없이 새로운 데이터가 추가될 때마다 데이터를 연결하면 된다는 편의성이 있다.

이진 탐색 트리는 부모 노드보다 숫자가 작은 데이터는 왼쪽 자식 노드, 숫자가 큰 데이터는 오른쪽 자식 노드로 저장하는 자료구조이다. 이진 탐색 트리에 대한 데이터 삽입, 삭제, 탐색은 트리의 높이에 비례하므로 평균 시간 복잡도는 O(logn)이다. 단, 기초적인 이진 탐색 트리의 경우 트리가 불균형하게 생성되기 쉬운데, 이 경우 worst-case 시간 복잡도인 O(n)에 가까워질 수 있다.

이 때문에 RB 트리, AVL 트리 등 이진 탐색 트리가 균형을 유지할 수 있도록 구현된 자료구조를 많이 사용하지만, 해당 자료구조에 대한 이해가 부족하기 때문에 배열, 링크드 리스트에 대해서만 의사코드를 작성했다.

2. 생각한 각 자료구조에 대해서 새로운 랭킹을 삽입 및 삭제를 구현하기 위한 pseudo-code를 작성하고, 시간 및 공간 복잡도를 계산한다.

a. 배열

구조체 배열 선언

```
typedef struct _Rank {
  int score;
  char name[NAME_LEN + 1];
} Rank;
Rank rank[RANK_NUM];
```

구조체 배열 삽입

적절한 순위를 탐색하고, 반복문을 통해서 순위를 하나씩 조정한다. RANK_NUM은 현재 랭킹에 저장된 점수의 개수를 의미하며, CURRENT_SCORE, CURRENT_NAME은 현재 gameOver 시의 점수, 이름을 의미한다. 점수는 내림차순으로 정렬되어 있다.

선형 탐색으로 자신보다 작거나 같은 score가 처음으로 등장할 때 found_rank flag를 set하고 그 뒤에 있는 기록들의 데이터를 전부 1칸 뒤로 이동한다. 만약 CURRENT_SCORE가 꼴등이거나 현재 랭킹에 저장된 점수가 하나도 없었을 경우 found_rank가 0이므로 마지막에 저장한다.

시간 복잡도는 랭킹의 모든 데이터를 선형 탐색하므로 O(n)이며, 공간 복잡도 역시 O(n)이다.

```
found_rank = 0
for i = 0 to RANK_NUM - 1
  if (CURRENT_SCORE >= rank[i].score && !found_rank)
    found_rank = 1
    for j = RANK_NUM downto i + 1
      rank[j].score = rank[j-1].score
      rank[j].name = rank[j - 1].name
    rank[i].score = CURRENT_SCORE
    rank[i].name = CURRENT_NAME
    break;
if (!found_rank)
  rank[RANK_NUM].score = CURRENT_SCORE;
  rank[RANK_NUM].name = CURRENT_NAME;
RANK_NUM++;
```

구조체 배열 삭제

입력된 순위의 데이터를 삭제한 뒤, 그 뒤의 순위를 전부 앞으로 조정한다. INPUT_RANK는 입력된 순위, CURRENT_SCORE, CURRENT_NAME은 현재 gameOver 시의 점수, 이름을 의미한다. 입력된 순위가 배열 밖의 순위일 경우 예외 처리한다.

시간 복잡도는 적절한 순위를 탐색하는 데엔 O(1)이 걸리지만, 입력된 순위보다 뒤에 있는 데이터를 선형 탐색하므로 **최종적으로 O(n)이며, 공간 복잡도 역시 O(n)**이다.

```
if (INPUT_RANK < 1 || INPUT_RANK > RANK_NUM)
  [print error message]
  return;
for i = INPUT_RANK - 1 to RANK_NUM - 1
  rank[i].score = rank[i + 1].score;
  rank[i].name = rank[i + 1].name;
RANK_NUM--;
```

링크드 리스트

구조체 배열 선언

```
typedef struct _Node{
          int score;
          char name[NAMELEN + 1];
          struct _Node *next;
} Node;
Node *rankRoot = NULL;
```

구조체 링크드 리스트 삽입

입력된 데이터에 대한 노드 newNode, 현재 가리키는 노드 포인터 curNode, 이전 노드 포인터 prevNode를 선언한다. newNode에 대한 데이터(점수, 이름)는 이미 입력했고, newNode->next는 NULL 포인터라고 가정한다.

curNode, prevNode로 링크드 리스트를 이동하며 적절한 순위를 탐색했다면 found_rank flag를 set하고 prevNode->next에 newNode, newNode->next에 curNode를 연결한 후 종료한다. 적절한 순위를 탐색했는데 prevNode == NULL이라면 1등인 경우이므로 rankRoot를 갱신한다.

만약 적절한 순위를 탐색하지 못했다면 꼴등이거나, 기록된 순위가 없는 것이므로 prev가 널 포인터인지 아닌지에 따라 행동을 결정한다.

적절한 순위를 탐색하기 위해서 링크드 리스트를 순서대로 탐색해야 하므로 시간 복잡도는 O(n)이지만, 삽입 자체에는 O(1)의 시간 복잡도가 발생한다. 따라서 **최종적인 시간 복잡도는 O(n)이며, 공간 복잡도 역시 O(n)**이다.

```
found_rank = 0
RANK_NUM++
curNode = rankRoot
prevNode = NULL
while (cur != NULL)
  if (newNode->score >= curNode->score)
    found_rank = 1
    newNode->next = curNode
    if (prevNode == NULL)
      rankRoot = newNode
    else
      prevNode->next = newNode
    break
  prevNode = curNode
  curNode = curNode->next
if (!found_rank)
  if (prevNode == NULL)
    rankRoot = newNode
  else
    prevNode->next = newNode
    newNode->next = curNode
```

구조체 링크드 리스트 삭제

입력된 데이터에 대한 노드 newNode, 현재 가리키는 노드 포인터 curNode, 이전 노드 포인터 prevNode를 선언한다. newNode에 대한 데이터(점수, 이름)는 이미 입력했고, newNode->next는 NULL 포인터라고 가정한다. INPUT_RANK는 삭제할 데이터의 rank를 가리킨다.

입력된 데이터가 적절한 순위가 아닐 경우 에러 메시지를 출력한다. rank를 1씩 증가시키면서 curNode 포인터가 NULL일 때까지 탐색하고, rank == INPUT_RANK일 때 prevNode가 NULL(순위가 1위)이라면 링크드 리스트의 head(rankRoot)를 초기화하고, 그렇지 않다면 이전 노드에 현재 노드의 다음 노드를 연결한다. 이후에 curNode의 동적 할당을 해제하고 break로 반복문을 종료한다.

적절한 순위를 탐색하기 위해서 링크드 리스트를 순서대로 탐색해야 하므로 시간 복잡도는 O(n)이지만, 삭제 자체에는 O(1)의 시간 복잡도가 발생한다. 따라서 **최종적인 시간 복잡도는 O(n)이며, 공간 복잡도 역시 O(n)**이다.

```
RANK_NUM--
curRank = 1
curNode = rankRoot
prevNode = NULL

if (INPUT_RANK < 1 || INPUT_RANK > RANK_NUM)
  [print error message]
  return;

else
  while (curNode != NULL)
    if (rank == INPUT_RANK)
      if (prevNode == NULL)
        rankRoot = NULL
      else
        prevNode->next = curNode->next
      free(curNode)
      break
    curRank++
    prevNode = curNode
    curNode = curNode->next
```

2. 생각한 각 자료구조에 대해서 사용자가 부분적으로 확인하길 원하는 정렬된 랭킹(x~y위, x≤y, x, y는 정수)의 정보를 얻는 방법을 간략히 요약해서 pseudo code로 작성하고, 시간 및 공간 복잡도를 계산한다.

배열에서의 순위 탐색

배열의 경우 단순히 X, Y값을 범위에 맞게 적절히 조정하고 X - 1 부터 Y - 1에 있는 데이터를 적절히 출력하면 된다. **시간 복잡도는 최악의 경우 rank의 모든 데이터를 탐색하므로 O(n)이고, 공간 복잡도 역시 O(n)**이다.

```
if (X < 1) X = 1
if (Y > RANK_NUM) Y = RANK_NUM
if (X > Y || RANK_NUM == 0)
  [print search failure message]
  return;
for i = X - 1 to Y - 1
  print(rank[i].name, rank[i].score)
```

링크드 리스트에서의 순위 탐색

X, Y를 범위에 맞게 적절히 조정하는 것까지는 같으나, 링크드 리스트의 경우 무조건 curNode 노드 포인터를 이용해 rankRoot에서부터 순위를 탐색해야 한다. 따라서 평균적으로 링크드 리스트가 배열에 비해 순위 탐색이 더 비효율적이다. 하지만 big-O notation은 worst-case를 계산하므로 **시간 복잡도는 동일하게 O(n)이며, 공간 복잡도 역시 O(n)**이다.

```
if (X < 1) X = 1
if (Y > RANK_NUM) Y = RANK_NUM
if (X > Y || RANK_NUM == 0)
  [print search failure message]
  return;

int curRank = 1
curNode = rankRoot
while (curNode != NULL)
  if (rank >= X && rank <= Y)
    print(rank[i].name, rank[i].score)
  curRank++
  curNode = curNode->next
```