전공 : 컴퓨터공학과 학년 : 4 학번 : 20212020 이름 : 박민준

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

1) Linked List (연결 리스트)

- 연결 리스트는 동적으로 메모리를 할당하여 노드를 추가하고 삭제하는 과정이 간단하며, 특히 순서대로 랭킹을 유지하면서 새로운 랭킹 정보를 삽입할 때 매우 유용하다.

- 각 노드는 사용자의 이름과 점수를 저장하며, 다음 노드를 가리키는 포인터를 가진다. 랭킹은 점수가 높은 순으로 정렬되어야 하기 때문에 새로운 점수를 삽입할 때는 적절한 위치를 찾아 노드를 삽입한다. 이러한 연산은 O(N) 시간 복잡도를 가지며, 새 랭킹 정보를 추가하거나 특정 범위의 랭킹을 조회할 때 유리하다.

2) Array (배열)

- 랭킹 정보를 저장하기 위한 구조체를 정의하고, 이를 배열에 저장하여 관리한다.

- 새로운 랭킹 정보를 배열에 삽입할 때는 점수를 기준으로 내림차순으로 정렬하여 삽입한다. 이를 위해 적절한 위치를 찾고, 필요하다면 기존의 데이터를 뒤로 밀어내야 한다.

- 특정 위치의 랭킹 정보를 삭제할 때는 해당 위치의 데이터를 제거하고 배열을 앞으로 당겨서 공간을 채워야 한다.

- 랭킹 정보를 출력하는 함수는 배열을 순회하면서 저장된 랭킹 정보를 출력하면 된다.

- 배열을 사용한 랭킹 시스템은 구현이 간단하고, 작은 데이터 set에 대해서는 효율적으로 작동할 수 있다. 하지만 랭킹 정보가 많아질 경우, 삽입과 삭제 연산의 비용이 크게 증가할 수 있다는 단점이 있다.

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

1) Linked List (연결 리스트)

(1) 삽입

void Insert(int score) {

Node\* newNode = new Node(score);

Node\* current = first;

while(current->next != NULL && current->next->score >= score) {

current = current->next;

}

newNode->next = current->next;

current->next = newNode;

}

- Node 삽입 시, 리스트를 순회하며 삽입할 위치를 찾아야 한다. 최악의 경우 모든 Node를 순회해야 하기 때문에 O(N)의 시간 복잡도를 갖는다. 여기서 N은 Linked List의 길이를 의미한다.

- Linked List는 새 Node를 만드는 데 필요한 공간만 고려하기 때문에 O(1)의 공간 복잡도를 갖는다.

(2) 삭제

void Delete(int score) {

if (head == 0) return;

Node\* current = first;

Node\* previous = 0;

while(current != 0) {

if (current->score = score) {

if (previous) {

previous->link = current->link;

current->link = 0;

}

else first = first->link;

break;

}

previous = current;

current = current->link;

}

delete current;

}

- Node 삭제 시, 특정 랭킹의 노드를 찾아 삭제한다. 순위에 따라 삭제할 노드를 찾아야 한다. 최악의 경우 모든 Node를 순회해야 하기 때문에 O(N)의 시간 복잡도를 갖는다.

- 추가 공간을 사용하지 않기 때문에 O(1)의 공간 복잡도를 갖는다.

2) Array (배열)

(1) 삽입

void insertRanking(char\* name, int score) {

RankingInfo newInfo; // 랭킹 정보를 저장하기 위한 구조체 선언

strcpy(newInfo.name, name);

newInfo.score = score;

int pos = rankingSize;

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

if (rankings[i].score < score) {

pos = i;

break;

}

}

for (int i = rankingSize; i > pos; i--) {

rankings[i] = rankings[i - 1];

}

rankings[pos] = newInfo;

}

(2) 삭제

void deleteRanking (int pos) {

if (pos < 0 || pos >= rankingSize) return;

for (int i = pos; i < rankingSize - 1; i++) {

rankings[i] = rankings[i + 1];

}

rankingSize--;

}

- Array 자료구조에서 삽입 연산의 경우, 적절한 위치를 찾는 데 최악의 경우 O(N) 시간이 걸리고, 배열을 밀어내는 데도 O(N)의 시간이 걸린다. 따라서 삽입 연산의 시간 복잡도는 O(N)이다.

- 삭제 연산의 경우, 배열을 앞으로 당기는 데 최악의 경우 O(N) 시간이 걸린다. 따라서 삭제 연산의 시간 복잡도 또한 O(N)이 된다.

- 배열을 사용한 랭킹 시스템의 공간 복잡도는 O(M)이다. 여기서 M은 배열의 최대 크기를 의미한다. 실제 사용하는 공간은 O(N)이지만, 배열의 최대 크기를 미리 할당해야 하므로 O(M)이 된다.

3. 생각한 각 자료구조에서 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.

1) Linked List (연결 리스트)

// x번째 위치까지 이동한다.

while (current != NULL && currentRank < x) {

current = current->next;

currentRank++;

}

// x부터 y까지 랭킹 정보를 출력한다.

while (current != NULL && currentRank <= y) {

printf(“%s: %d”, current->name, current->score);

current = current->next;

currentRank++;

}

- 해당 연산은 최악의 경우 O(N)의 시간 복잡도를 갖는다. 여기서 N은 연결 리스트의 전체 노드 수이다. x = 1이고 y = N일 때 전체 리스트를 순회하게 된다.

- 추가적인 공간 사용 없이 Linked List의 노드를 순회하기 때문에 O(1)의 공간 복잡도를 갖는다.

2) Array (배열)

for (i = x~y) {

printf(“%s: %d”, current->name, current->score);

}

- 배열에서는 랭킹 정보가 배열 인덱스에 따라 이미 정렬되어 저장되어 있기 때문에 x부터 y까지의 랭킹 정보를 직접적으로 인덱스로 접근하여 빠르게 얻을 수 있다. 따라서 해당 연산의 시간 복잡도는 Linked List와 마찬가지로 최악의 경우 O(N)이 된다.

★ 배열 인덱스는 0부터 시작하므로, 실제 접근은 x-1부터 y-1까지이다.

- 또한, 주어진 배열 내에서 인덱스로 접근하므로 추가 공간을 사용하지 않는다. 따라서 공간 복잡도는 O(1)이다.