7주차 결과보고서

전공: 컴퓨터공학과 학년: 2학년 학번: 20191619 이름: 이동석

**1. 자료구조**

typedef struct node{

int data;

int rank;

char name[NAMELEN];

struct node \*left;

struct node \*right;

}NODE;

NODE \* root=NULL;

이진트리를 만들기 위해 사용한 구조체는 위와 같다. 이진트리는 정렬된 자료를 차례로 삽입하게 될 경우, 한쪽으로 쏠린 구조가 되어 링크드 리스트와 같은 형태가 되어버린다. 따라서 이를 방지하기 위해, 트리를 먼저 만들도록 했다. 다음 그림과 같은 트리가 형성된다.

와이어, 금속, 광장이(가) 표시된 사진

자동 생성된 설명

우선, 파일의 첫줄에는 랭크의 총 개수가 저장되어있다. 편의상 rank의 총 개수를 count라 하겠다. 이때, root 노드에는 중간 랭크가 될 수 있도록, root->rank에 count/2를 먼저 할당한다. 이후 노드들을 1~count 까지 중복되지 않고 랜덤하게 값을 삽입하면, 균형잡힌 이진트리가 생성된다. 생성된 이진트리에 파일을 읽으면서 값을 할당한다. 이때, 공간복잡도는 count만큼의 구조체 배열이라 할 수 있다. 시간복잡도의 경우 노드를 모두 살펴보면서 count개의 값을 할당하므로, O(count\*count) 라 할 수 있다. 이 경우 링크드 리스트보다 처음 파일의 값을 읽는 작업이 훨씬 느려 효율적이지 못하다.

하지만, 게임이 끝나고 새로운 랭킹 정보를 등록할 때, 이진트리는 링크드 리스트보다 효율적이다. 이진트리의 삽입과정은, depth 즉 깊이와 연관이 있는데 한번 비교할 때 마다 비교 대상이 절반으로 계속 줄어든다. 따라서, 랭킹 정보를 등록할때의 시간복잡도는 O(log n)이다. 새로운 랭킹을 위한 공간복잡도는 O(1)이며, 전체 공간복잡도는 O(n)이 되어 연결리스트와 동일하다.

원하는 랭킹 범위를 입력받고 출력하는 과정은, 처음부터 마지막 랭크까지 다 확인하고 조건에 맞게 출력하므로, 시간복잡도는 노드의 개수 O(n) 가 된다. 이때, 공간복잡도 역시 O(n)이 된다. 연결리스트와 비교해서 연결리스트는 처음부터 y번째 까지만 확인한다는 점에서 조금 느리다는 것을 알 수 있다.

연결리스트와 비교하여, 장단점이 분명하게 존재한다. 연결리스트는 삽입과 삭제의 시간복잡도가 노드의 개수인 O(n)이며, 초기 파일의 값을 읽고 연결리스트를 만드는 데 걸리는 시간복잡도 역시 O(n)이라 할 수 있다. 이 자료구조는 삽입의 과정이 연결리스트에 비해 월등히 빠름을 알 수 있다. 하지만, 초기 트리구조를 만드는 데 있어 단순 연결리스트에 비해 느리게 동작한다. 또한, 삭제 연산에서 rank에 맞는 노드를 찾는 과정이 모든 노드를 확인한다는 점에 있어서, 연결리스트보다 이점은 없다. 공간복잡도의 경우 연결리스트와 거의 동일하다.

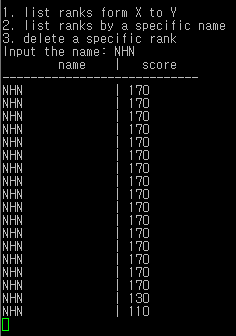
이진트리의 경우 rank가 아니라 data의 삭제였다면 연결리스트 보다 훨씬 효율적이라 말 할수 있었을 것이다. 또한, 정렬된 배열이 아닌 무작위 랭크를 정렬하여 삽입하는 경우였다면 연결리스트보다 효율적인 자료구조가 될 수 있었다. 하지만, 이 자료구조의 경우 이 프로그램을 실행하는데 있어 연결리스트보다 효율적이라 말할 수는 없다. 단지, 새로운 랭킹정보의 저장에 있어서는 확실하게 효율적이라 말할 수 있다. 주어진 두개의 평가기준으로만 보았을 때는 연결리스트보다는 조금 더 나은 자료구조라 할 수 있다.

**2. 실행 결과**

**텍스트이(가) 표시된 사진

자동 생성된 설명**

rank.txt 파일은 제공된 파일을 사용하였다.

 텍스트이(가) 표시된 사진

자동 생성된 설명

실행 결과는 다음과 같다.

**3. 시간 및 공간 복잡도**

위 함수를 구현하는 데 필요한 시간복잡도는 모든 노드를 탐색하므로 노드의 개수인 O(n)이라 말할 수 있다. 각 노드에서 strcmp 함수를 이용해 입력받은 이름과 정보가 같은지 비교하고, 맞으면 출력한다. 이에, 공간복잡도 역시 O(n)이라 말할 수 있다. 코드는 재귀함수로 작성하였다.

void print\_list\_2(NODE \*root, char \*name, int \*check)

{

if (root==NULL) return ;

if (root !=NULL) {

print\_list\_2(root->left, name,check);

if (!strcmp(name, root->name)) {

printw("%-16s| %-d\n", root->name, root->data);

\*check=1; // 같으면 출력

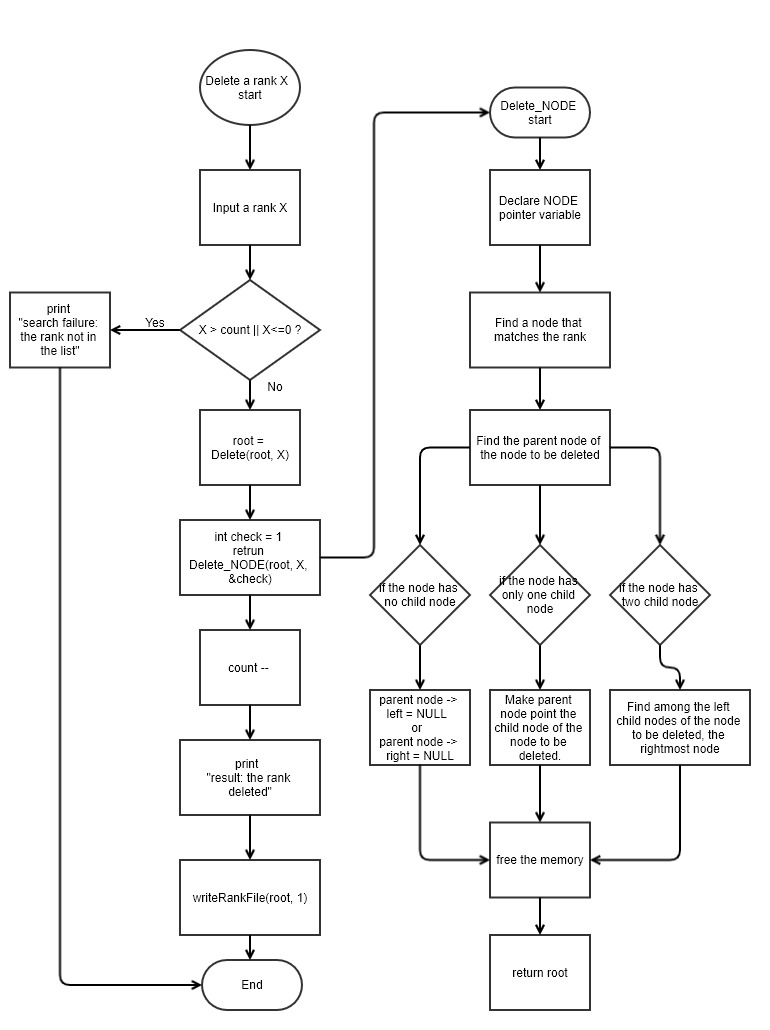
}

print\_list\_2(root->right,name, check);

}

}

**4. 삭제 알고리즘 순서도**



**5. 습득한 내용**

본 실험 및 숙제를 하면서 이진트리의 삽입과 삭제연산에 대해 많이 배웠다. 이진트리의 경우 처음 접해보고 한번도 사용해 본적이 없던 자료구조였기 때문에 직접 구현하고 스스로 어떤식으로 구현해야하는지 생각해 볼 수 있었다. 구현은 어찌 하였지만, 아직은 미숙하여 코드 자체가 보기가 불편하고, 쓸데 없는 변수들이 조금 섞여있음을 알 수 있다. 또한 위에서 설명하였듯이, 효율적일 것이라 생각하고 만들었지만 사실상 연결리스트와 비슷한 성능을 보여준다. 분명히, 이진트리 구조를 조금 더 적극활용한 알고리즘이 존재할 것이라 생각되어 조금 아쉬운 코드라 생각한다.