9주차 예비보고서

전공 : 경영학과 학년 : 4학년 학번 : 20190808 이름 : 방지혁

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

1) Sorted Linked List

Sorted linked list는 말 그대로 정렬된 linked list로 각 노드의 포인터가 다음 노드를 가리킵니다. 이를 높은 점수가 앞쪽에 위치하도록 하기 위해서 리스트에 새로운 노드를 삽입할 때 점수를 비교하여 삽입해서 정렬된 상태를 유지합니다.

```
Typedef struct Node {
        Char name[NAMELEN];
        Int score;
        Struct Node *next; // 다음 위치를 가리키는 포인터
} Node:

Node *head = NULL; // 리스트의 첫 번째 노드를 가리키는 포인터
```

2) 동적배열

동적 배열은 contiguous한 메모리 공간에 저장하는 자료구조입니다. 필요에 따라 크기를 늘리고, 삽입이 이뤄지고 나서 정렬작업을 수행함으로서 점수를 기준으로 내림차순 정렬을 유지합니다.

```
Typedef struct Node {
        Char name[NAMELEN];
        Int score;
} Node:

Node *rankArray = NULL; // 동적 배열 포인터
Int arraySize = 0; // 데이터 개수
Int capacity = 10; // 현재 최대용량
```

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

1) Sorted Linked List

삽입연산

```
Insert Node(...):
        // 새로운 노드 생성
        NewNode = malloc(...);
        NewNode.name= name;
        NewNode.score = score;
        NewNode.next = NULL;

        // CASE 1: 리스트가 비어있거나, 처음 위치에 삽입 하는 경우
        If head == NULL || head.. score < score:
                newNode.next = head
                head = newNode
                return

        // CASE 2: 그 외의 경우 위치 찾기
        Curr = head
        While curr.next != NULL && curr.next.score >= score:
                Curr = curr.next

        // 노드 삽입
        newNode.next = curr.next
        curr.next = newNode

        return
```

삭제연산

```
Delete Node(...):
        // CASE 1: 비어 있음
        if head is NULL: return FAILURE
        // CASE 2: 첫번째 노드 삭제
        If head.name == name:
                Temp = head
                Temp = head.next
                Free(temp)
                Return SUCCESS
        // CASE 3: 리스트 탐색
        Curr = head
        While curr.next != NULL:
                If curr.next.name == name
```

```
                    Temp = curr.next
                    Curr.next = curr.next.next
                    Free(temp)
                    Return success
            Curr = curr.next
    // CASE 4: 없음
    Return failure
```

삽입 연산의 시간 복잡도를 분석하자면 최선의 경우는 새로 삽입되는 점수가 가장 높아서 맨 앞에 삽입되는 경우로 O(1)이지만, 최악의 경우를 보자면, 결국에는 n개의 노드(rank)가 있을 경우 리스트의 모든 노드를 순회해야 해서 O(n)의 시간 복잡도가 걸릴 것입니다. 공간 복잡도의 경우 삽입 연산 1개만 보자면 O(1)이지만 전체적인 삽입을 보면 O(n)일 것입니다.

삭제 연산의 시간 복잡도를 분석하자면 마찬가지로 첫번째 노드를 삭제한다면 O(1)이겠지만 최악의 경우를 따져야하기 때문에 O(n)일 것입니다. 해당 경우 리스트의 모든 노드를 탐색하는 경우입니다. 공간복잡도는 추가적인 메모리 할당이 필요 없기에 O(1)이 될 것입니다.

2) 동적배열

삽입연산

```
Function InsertRank(...):
    // 1. 배열이 가득 찬 경우 크기 확장
    if arraySize >= arrayCapacity:
        arrayCapacity += 10
        tempArray = alloc (arrayCapacity * sizeof(Node))
    // 기존 데이터 복사
    for i from 0 to arraySize - 1:
        tempArray[i] = rankArray[i]
    free(rankArray)
    rankArray = tempArray
    // 2. 배열 끝에 새 데이터 추가
    rankArray[arraySize].name = name
    rankArray[arraySize].score = score
    arraySize += 1

    // 3. 정렬 수행 (Quick Sort - 내림차순)
    QuickSort(rankArray, 0, arraySize - 1)
    return SUCCESS
```

```
QuickSort(...):
        If low < high:
                Index = partition(array, low, high)
                quicksort(array, low, index - 1)
                quicksort(array, index + 1, high)

partition(...):
        pivot = array[high].score
        i = low – 1
        for j from low ~ high – 1:
                if array[j].score >= pivot:
                        i++;
                        array[i]와 array[j]의 값을 바꾼다
        array[i+1]과 array[high]의 값을 바꾼다
        Return I + 1
삭제연산
DeleteNode:
        deleteIndex = -1

        for i = 0 to arraySize – 1:
                if rankArray[i].name == name:
                        deleteIndex = i // 찾았음!!
                        break
        // 못 찾은 경우
        if deleteIndex == -1
                return FAILURE
        // 찾았으니 한 칸씩 앞으로 이동
        for I = delteIndex to arraysize – 2:
                rankArray[i] = rankArray[i+1]

        // 크기 감소
        arraySize -= 1
        return Success
```

삽입 연산의 시간 복잡도를 분석하자면 quicksort의 평균 시간복잡도인 O(NlogN)의 시간 복잡도가 걸릴 것입니다. 최악의 경우 O(N^2)이 될 것입니다. 공간 복잡도의 경우 배열 전체를 따지자면 O(n)이 소요될 것이고, 재할당시 O(2N)이 될 것입니다.

삭제 연산의 시간 복잡도를 분석하자면 이름으로 검색하기에 탐색에서 O(N), 이동에서 O(N)으로 전체적으로 O(N)의 시간 복잡도가 소요될 것입니다. 공간 복잡도의 경우 추가적인 메모리 사용이 없기에 O(1)일 것입니다.

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

Sorted linked list의 경우

```
GetRank:
        // X위까지 이동
        Curr = head
        Pos = 1
        While curr != NULL && pos < x:
                Curr = curr.next
                Pos += 1
        // x 위 못 찾음
        If curr == NULL
                Return FAIL
        // x부터 Y까지 출력
        While curr != NULL && pos <= y
                Print(....)
                Curr == curr.next
                Pos += 1
        Return success
```

동적 배열의 경우

```
GetRank:
        For pos = x to y:
                Index = pos – 1
                Printf(...)
        Return SUCCESS
```

Sorted linked list의 시간 복잡도의 경우 O(x + (y – x + 1)이 될 것이고 즉 O(y)가 전체 시간 복잡도가 될 것입니다. 최악의 경우 O(N)이라고 볼 수도 있습니다. 공간 복잡도는 추가적인 메모리 할당이 없기에 O(1)이라고 볼 수 있습니다.

동적 배열의 시간복잡도의 경우 O(y – x + 1)이 될 것이며, 즉 이것은 조회 범위 크기입니다. 인덱스를 통한 랜덤 액세스가 O(1)이기에 sorted linked list의 O(y)보다 효율적이라고 말할 수 있겠습니다. 공간 복잡도는 마찬가지로 추가적인 메모리 할당이 없기에 O(1)이라고 볼 수 있습니다.