알고리즘 과제 보고서 7

2022110151 이주연

문제 1. 순차 탐색

1) 최악의 상황 고려해 배열 생성한 후, 순차탐색 실행하여 비교횟수 출력하기

(출력 결과)

```
2022110151 이주연

1. [순차 탐색] - Sequential search

찾고자 하는 대상(searchSubject)이 10일 때

비교 횟수(count)는 9번! (10번 Index에서 찾았습니다)
```

(설명 + 이해를 돕기 위한 핵심 코드 첨부)

```
// 순차 탐색
int sequential(int k, int* count) {
    int i = 1;

    // 만약 i가 배열 안에 있고, k 즉, 찾는 대상을 아직 못 찾았다면 (key와 k값이 다를 때)
    while ((i < N + 1) && (a[i].key != k)) {
        i++; // i 하나 증가시켜 다시 찾고
        (*count)++; // 비교횟수도 하나 증가시키기
    }

    // i를 반환
    return i;
}
```

최악의 상황을 고려하기 위해 배열의 크기가 10인 a[] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]로 설정하였고, 순차 탐색 함수에서 찾는 대상(searchSubject=10)과 기존 배열 a[]에 있는 값 (a.key = 10)을 비교하며 순차적으로 탐색하기 위해 구조체 배열을 선언했습니다(이때, a[N+1]인 이유는 인덱스를 0~N-1대신 1부터 N까지로 설정해 더욱 보기 쉽도록 하기 위해 이와 같이 설정했습니다.)

메인 함수에서 찾고자 하는 대상을 설정하고 (10), 비교횟수를 0으로 초기화해준 다음, sequential 함수에 searchSubject인 10과, 초기화된 count의 주소를 인자로 전달해 만약 배열 안에서(i < N+1), 찾는 대상(k 즉, searchSubject) 10이 a배열의 key값과 다를 경우(a[i].key != k) 아직 10을 못 찾았다고 판단해, i를 하나 증가시키고, count(비교횟수)를 하나 증가시켜 배열의 맨 왼쪽 (첫 key값)부터 탐색하며 최악의 경우인 맨 오른쪽까지 탐색하도록 설정했습니다. 그럼에도 불구하고 키 값을 못 찾았을 경우에는 이미 탐색하는 배열을 벗어났으므로 while문을 빠져나오고 탐색 실패가 출력되도록 설정했습니다.

따라서, 배열의 크기를 10으로 설정하고, 1부터 10까지 수에서 찾는 대상을 10이라고 가정했을 때 (최악의 경우를 의미), 비교횟수 (count)가 10에서 하나 적은 수인

알고리즘 과제 보고서 7

2022110151 이주연

9 가 나옵니다.

2) 해당 과정을 통해 점근적으로 나아가 길이가 N일 경우에 대해 순차탐색의 최악의 시간복잡도 서술하기

배열의 크기가 N일 때, 최악의 경우 배열에 있는 값을 전부 하나하나 확인하며 비교해야 하기에 최악의 경우 시간 복잡도는 배열의 크기에 비례합니다. 비교횟수는 N-1이며, N-1에서 -1은 N에 비해 비교적 영향을 덜 주므로 이를 무시했을 때 최악의 시간 복잡도는 O(N)이 나옵니다. (결론 : T(n) = O(N))

(다음장에 문제 2번 계속됩니다!)

알고리즘 과제 보고서 7

2022110151 이주연

문제 2. 이진 탐색

1) 크기가 16이상인 이미 정렬된 배열 생성하기

```
#define N 32  // 배열 크기 >= 16 (이상) 설정 (2씩 증가한 정렬된 값 저장용)
//구조체 선언하기 ( key 값을 저장하기 위한 용도!)
typedef struct {
    int key;
} Element;

// 구조체 배열 선언하기 : 인덱스 1~N까지 a[]
Element a[N + 1];

int n = N;  //  전역변수로 사용하기 위해 단순 복사

int main() {
    int i;

    // 0. 배열 초기화하기 ( 2, 4, 6, ..., 64 ) 오름차순 정렬된 상태로
    for (i = 1; i <= N; i++) {
        a[i].key = i * 2;  // key값을 2씩 증가하여 저장
    }
```

배열의 크기를 16이상인 32로 설정했습니다. 이번에는 정렬된 다른 배열을 보여드리기 위해 짝수 형태로 증가하는 형식의 오름차순 정렬 배열을 선언했습니다. key값을 저장하기 위해 구조체를 선언하고, 구조체 배열인 a를 선언한 뒤(여기서도0~N-1이 아닌 1~N으로 표현하기 위해 +1을 하였습니다), 메인 함수에서 for문을 통해 key값을 짝수 형태로 저장했습니다.

2) 실행 결과

```
1. 순차 탐색: 30번째 위치에서 찾았습니다! 비교횟수: 29회
2. 이진 탐색: 30번째 위치에서 찾았습니다! 비교횟수: 4회
```

찾고자 하는 값을 2~ 64 값 중에서 60(searchSubject)으로 설정한 후, 순차 탐색과 이진 탐색 함수를 각각 호출했을 때 다음과 같은 결과가 도출됐습니다. 순차 탐색의 경우 60값을 찾기 위해 N-1회, 즉 29회 비교를 하였고, 이진 탐색의 경우는 4회로 더 많이 줄어든 모습을 볼 수 있었습니다.

2022110151 이주연

3) 이진 탐색과 순차 탐색 결과 비교를 통한 성능 비교 + 성능 차이가 발생한 이유 서술

```
// 2. 이진 탐색 ------------------------------------------------------------------------
// 기본 알고리즘 : mid값을 기준으로 탐색 범위를 반으로 나누기
//                  mid
//                /     \
//   low ~ mid-1(왼)    mid+1 ~ high (오)
int binary(int k, int* count) {
    // 트리 형태로 분할 하기 위해 설정한 인덱스
    // 탐색 대상되는 데이터의 시작 위치와 끝 위치 가리킴
    int low, high, mid;
    low = 1;
    high = n;

    // low <= high 동안 : 즉, 탐색 가능한 범위일 때 (low>high면 탐색키가 존재하지 않으므로)
    while (low <= high) {
        mid = (low + high) / 2;  // 중간 위치를 계산하고,
        (*count)++;              // 비교 횟수를 증가한 다음

        //mid를 기준으로
        if (k == a[mid].key) // 1) 만약 값이 mid값이면 바로 인덱스를 리턴하고
            return mid;
        if (k < a[mid].key)  // 2) 찾는 값이 mid 값보다 작으면
            high = mid - 1;  //    왼쪽 부분으로 가서 탐색하고 ( 탐색 범위 축소 )
        if (k > a[mid].key)  // 3) 찾는 값이 mid 값보다 크면
            low = mid + 1;   //    오른쪽 부분으로 가서 탐색하기 (탐색 범위 축소)
    }

    return 0;  // 못 찾았을 경우에는 0리턴하기
}
```

60을 찾고자 할 때 순차탐색은 29회, 이진 탐색은 4회로 확연한 차이가 남을 알 수 있었습니다. 순차 탐색의 경우에는 n-1번의 비교가 일어나 최악의 시간 복잡도 모두 O(n)임을 알 수 있었고, 이진 탐색의 경우, O(logn)로 성능이 순차 탐색보다 더욱 좋다는 사실을 발견할 수 있었습니다. (순차탐색은 N-1번 비교를 통해 60을 찾기 위해 29번의 비교횟수 발생, 이진탐색은 (4 <) log30 (< 5) 즉, 4번의 비교횟수 발생)

이와 같이 성능 차이가 발생한 이유는 이진 탐색의 경우 이진 트리로 그려보았을 때, 순차적으로 하나하나 일일이 비교하는 순차탐색과 달리 트리를 나누어 탐색하기 때문입니다. mid를 기준으로, mid 왼쪽에 mid값보다 더 작은 값들을 배치하고, mid의 오른쪽 서브트리에 mid값보다 더 큰 값들을 배치하기 때문에 값을 찾을 때 mid를 기준으로 작다면 왼쪽 서브트리만을, mid보다 크다면 오른쪽 서브트리만을 탐색하면 됩니다. (즉 탐색 범위가 반으로 줄어들게 됩니다. 따라서 logn번 비교)

이를 알고리즘으로 본다면, low, high, mid인덱스로 나누어 볼 수 있습니다. 1~N까지 주어졌을 때, 1을 low로 두고, high를 n으로 둔 뒤, 이를 2로 나눠 mid로 설정하고, 만약 값이 mid값이라면 바로 인덱스를 반환하고, 만약 찾는 값 k가 mid보다 작다면 왼쪽 서브트리로 가도록 high= mid-1로 설정해주고, 찾는 값 k가 mid값보다 크다면 low 인덱스를 mid+1로 업데이트해 오른쪽 서브 트리로 가서 값을 찾을 수 있습니다. 이를 바탕으로 2)번과 같이 이진 탐색이 더욱 좋은 성능을 냄을 알 수 있었습니다.