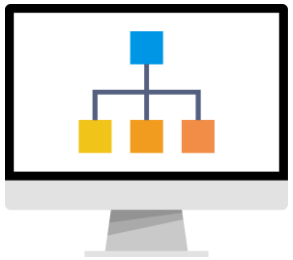


Data Structure & Algorithm
자료구조 및 알고리즘

21. 보간 탐색 (Interpolation Search)

20강 보고서 돌아보기

- 해쉬 테이블에서 Load Factor는 무엇일까?
- SHA(Secure Hash Algorithm) 함수들의 목적은 무엇일까?
- 해쉬 함수는 웹 사이트의 민감한 정보(비밀번호)를 저장하는데 흔히 사용된다고 한다. 어떤 원리일까?

Load Factor

- $load\ factor\ (\tau) = \frac{n}{k}$
- n: 저장된 데이터의 개수
- k: 슬롯의 개수

- 사번의 끝자리를 해쉬 값으로 20명의 데이터를 저장했다.
  - n = 20
  - k = 10
  - Load Factor = 20 / 10 = 2

Secure Hash Algorithm

- 미국 NIST에서 정하는 안전한 해쉬 알고리즘들

- 입력: 임의의 길이의 문자열 (넓은 범위)
- 출력: 정해진 길이의 문자열 (좁은 범위)
  - 128비트 암호화, 256비트 암호화
- 해쉬 알고리즘이 “안전하다”?

SHA1("The quick brown fox jumps over the lazy dog") =
2fd4e1c67a2d28fced849ee1bb76e7391b93eb12

SHA1("The quick brown fox jumps over the lazy dog.") =
408d94384216f890ff7a0c3528e8bed1e0b01621

비밀번호 저장하기

- 비밀번호를 평문(plain text)으로 데이터베이스에 저장하는 것은 위험하다!

- (입력된 비밀번호) == (저장된 비밀번호)
- SHA1(입력된 비밀번호) == SHA1(저장된 비밀번호)

- 비밀번호의 해쉬 값을 데이터베이스에 저장해두자!
  - 아주 낮은 확률로 해쉬 값이 충돌할 수 있다.

과제 1 풀이

```c
typedef struct node_t_ {
    int data;
    struct node_t_* next;
    struct node_t_* prev;
} node_t;

typedef struct list_t_ {
    int count;
    node_t* head;
} list_t;
```

```c
list_t lists[NUM_LISTS];

void initialize() {
    for (int i = 0; i < NUM_LISTS; i++) {
        lists[i].head = NULL;
        lists[i].count = 0;
    }
}
```

```c
int insert(int id, int pos, int data) {
    list_t* list = &lists[id];
    if (pos == -1) pos = list->count;
    if (pos > list->count) return -1;

    node_t* node = create_node(data);

    if (list->head == NULL) {
        list->head = node;
        list->count = 1;
        return 1;
    }

    node_t* p = list->head, * p_prev = NULL;

    while (pos--) {
        p_prev = p;
        p = p->next;
    }

    node->prev = p_prev;
    node->next = p;

    if (p_prev) p_prev->next = node;
    if (p) p->prev = node;

    if (p == list->head) list->head = node;
    list->count++;

    return 1;
}
```

- <pos> = -1 고려
- 빈 리스트에 노드를 처음으로 추가할 때
- 중간 노드를 추가할 때
- 추가하는 노드가 첫 노드일 때
  - p_prev = NULL일 것이다.
- 추가하는 노드가 마지막 노드일 때
  - p = NULL일 것이다.

```c
int insert(int id, int pos, int data) {
    list_t* list = &lists[id];
    if (pos == -1) pos = list->count;
    if (pos > list->count) return -1;

    node_t* node = create_node(data);

    if (list->head == NULL) {
        list->head = node;
        list->count = 1;
        return 1;
    }

    node_t* p = list->head, * p_prev = NULL;

    while (pos--) {
        p_prev = p;
        p = p->next;
    }

    node->prev = p_prev;
    node->next = p;

    if (p_prev) p_prev->next = node;
    if (p) p->prev = node;

    if (p == list->head) list->head = node;
    list->count++;

    return 1;
}
```

- <pos> = -1 고려
- 빈 리스트에 노드를 처음으로 추가할 때
- 중간 노드를 추가할 때
- 추가하는 노드가 첫 노드일 때
  - p_prev = NULL일 것이다.
- 추가하는 노드가 마지막 노드일 때
  - p = NULL일 것이다.

```c
int insert(int id, int pos, int data) {
    list_t* list = &lists[id];
    if (pos == -1) pos = list->count;
    if (pos > list->count) return -1;

    node_t* node = create_node(data);

    if (list->head == NULL) {
        list->head = node;
        list->count = 1;
        return 1;
    }

    node_t* p = list->head, * p_prev = NULL;

    while (pos--) {
        p_prev = p;
        p = p->next;
    }

    node->prev = p_prev;
    node->next = p;

    if (p_prev) p_prev->next = node;
    if (p) p->prev = node;

    if (p == list->head) list->head = node;
    list->count++;

    return 1;
}
```

- <pos> = -1 고려
- 빈 리스트에 노드를 처음으로 추가할 때
- 중간 노드를 추가할 때
- 추가하는 노드가 첫 노드일 때
  - p_prev = NULL일 것이다.
- 추가하는 노드가 마지막 노드일 때
  - p = NULL일 것이다.

```c
int find(int id, int data) {
    list_t* list = &lists[id];
    node_t* p = list->head;
    int i = 0;

    while (p) {
        if (p->data == data) return i;
        i++;
        p = p->next;
    }

    return -1;
}
```

```c
int delete(int id, int pos) {
    list_t* list = &lists[id];
    if (pos >= list->count) return -1;
    if (pos == -1) pos = list->count - 1;

    node_t* p = list->head;

    while (pos--) {
        p = p->next;
    }

    //delete p
    node_t* p_prev = p->prev,
        * p_next = p->next;

    if (p_prev) p_prev->next = p_next;
    if (p_next) p_next->prev = p_prev;

    if (p == list->head) list->head = p->next;
    free(p);

    list->count--;
    return 1;
}
```

- <pos> = -1 고려
- 중간 노드를 삭제할 때
- 삭제하는 노드가 첫 노드일 때
  - p_prev = NULL일 것이다.
- 삭제하는 노드가 마지막 노드일 때
  - p_next = NULL일 것이다.

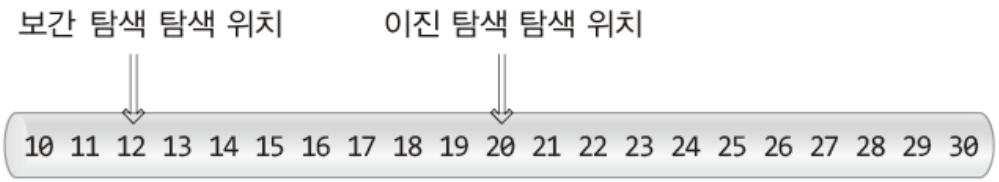
```c
int count(int id) {
    return lists[id].count;
}

void reset(int id) {
    list_t* list = &lists[id];
    node_t* p = list->head, * p_next;

    while (p) {
        p_next = p->next;
        free(p);
        p = p_next;
    }

    list->head = NULL;
    list->count = 0;
}
```

```c
void print(int id) {
    list_t* list = &lists[id];
    node_t* p = list->head;

    while (p) {
        printf("%d ", p->data);
        p = p->next;
    }
    printf("\n");
}

void print_reverse(int id) {
    list_t* list = &lists[id];
    node_t* p = list->head;

    while (p && p->next) p = p->next;

    while (p) {
        printf("%d ", p->data);
        p = p->prev;
    }
    printf("\n");
}
```

보간 탐색

- 이진 탐색과 보간 탐색 모두 정렬이 완료된 데이터를 대상으로 탐색을 진행하는 알고리즘이다.

아래의 배열을 대상으로 정수 12를 찾는다고 가정하면?

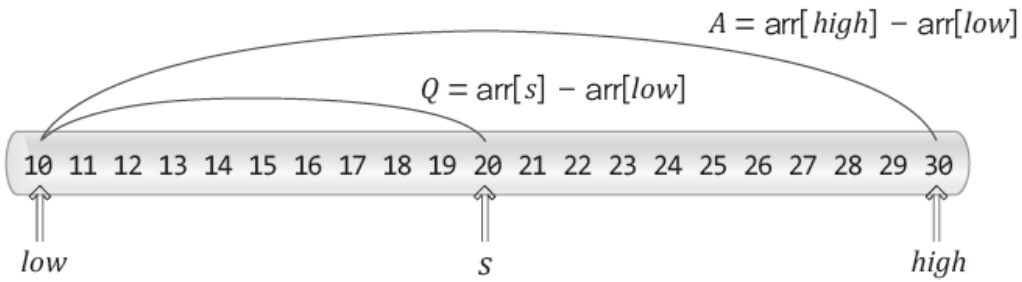


알고리즘 별 탐색 위치의 선택 방법

- 이진 탐색 : 무조건 중간에 위치한 데이터를 탐색의 위치로 결정!
- 보간 탐색 : 대상에 비례하여 탐색의 위치를 결정!

보간 탐색은 단번에 탐색 대상을 찾을 확률이 어느 정도 존재한다.

보간 탐색: 비례식 구성



low, high는 시작과 끝의 인덱스 값

탐색 대상이 저장된 인덱스 값 s

$$A : Q = (high-low) : (s-low)$$

비례식 구성

$$s = \frac{Q}{A}(high-low)+low$$

$$s = \frac{x-arr[low]}{arr[high]-arr[low]}(high-low)+low$$

탐색 위치의 인덱스 값 계산식

보간 탐색의 구현

이진 탐색

```
int BSearchRecur(int ar[], int first, int last, int target)
{
    int mid;
    if(first > last)
        return -1;

    mid = (first+last) / 2;

    if(ar[mid] == target)
        return mid;
    else if(target < ar[mid])
        return BSearchRecur(ar, first, mid-1, target);
    else
        return BSearchRecur(ar, mid+1, last, target);
}
```

교체하면 보간 탐색이 된다!

```
mid = ((double)(target-ar[first]) / (ar[last]-ar[first]) *
       (last-first)) + first;
```

$$s = \frac{x\text{-}arr[low]}{arr[high]\text{-}arr[low]}(high\text{-}low)+low$$

보간 탐색의 구현: 오류의 수정

```
int ISearch(int ar[], int first, int last, int target)
{
    int mid;

    if(first > last)
        return -1;          // -1의 반환은 탐색의 실패를 의미

    mid = ((double)(target-ar[first]) / (ar[last]-ar[first]) *
     0       (last-first)) + first;

    if(ar[mid] == target)
        return mid;         // 탐색된 타겟의 인덱스 값 반환
    else if(target < ar[mid])
        return ISearch(ar, first, mid-1, target);
    else
        return ISearch(ar, mid+1, last, target);
}
```

```
if(ar[first]>target || ar[last]<target)
    return -1;
```

ISearch(arr, 1, 4, 2); 이전 호출과 동일한 인자 전달!

탐색 대상이 존재 않는 경우

```
int main(void)
{
    int arr[] = {1, 3, 5, 7, 9};
    . . . .
    ISearch(arr, 1, 4, 2);
    . . . .
}
```

위의 함수 호출로 mid는 0이 된다.

그래서 탈출 조건은 이진 탐색의 경우와 달리해야 한다. 보간 탐색의 탈출 조건은 다음의 특성을 기반으로 구성해야 한다.

"탐색대상이 존재하지 않는 경우, ISearch 함수가 재귀적으로 호출됨에 따라 target에 저장된 값은 first와 last가 가리키는 값의 범위를 넘어서게 된다."

요약

- 해쉬함수는 임의의 길이의 데이터를 고정된 길이의 해쉬 값으로 변환할 수 있다.
  - 좋은 해쉬함수는 입력이 조금만 달라져도 해쉬 값에 큰 변화가 일어난다.
  - 해쉬 값을 보고 원본 데이터를 복구할 수 없다!

- 보간 탐색은 이진 탐색에서 양 끝의 인덱스뿐만 아니라 양 끝 값을 확인하여 찾고자 하는 값이 존재할 인덱스를 추정한다.

출석 인정을 위한 보고서 작성

- 아래 질문에 대해 답한 후 포털에 제출

- 보간 탐색에서 최악의 경우 시간복잡도는 얼마일까? 또, 어떤 형태의 입력데이터가 주어졌을 때 최악의 경우가 될까?