

#2. 배열, 리스트, 해시테이블

나정휘

<https://justicehui.github.io/>

자료구조

자료구조

- 데이터를 잘 저장하는 방법에 대해 다루는 분야
- 해결하고자 하는 문제에 따라 적합한 방법을 선택해서 저장해야 함
- 사전 vs 단어장
 - 사전: 뜻이 궁금한 단어를 찾아보는 용도
 - 단어장: 단어를 암기하기 위해 보는 용도
- 도서관 이용자가 책을 찾는 상황
 - 원하는 책의 제목을 알고 있어서, 그 제목을 가진 찾아야 하는 사람
 - IT 전공 서적을 구경하고 싶은 사람

자료구조

자료구조

- 데이터를 잘 저장하는 방법에 대해 다루는 분야
- 해결하고자 하는 문제에 따라 적합한 방법을 선택해서 저장해야 함
- 사전 vs 단어장
 - 사전: 뜻이 궁금한 단어를 찾아보는 용도
 - 원하는 단어를 빠르게 찾을 수 있도록 사전 순으로 정렬해서 배치
 - 단어장: 단어를 암기하기 위해 보는 용도
 - 학습의 효율성을 위해 접사/어근/상황 등에 따라 묶어서 제공
- 도서관 이용자가 책을 찾는 상황
 - 원하는 책의 제목을 알고 있어서, 그 제목을 가진 찾아야 하는 사람
 - IT 전공 서적을 구경하고 싶은 사람

자료구조

자료구조

- 데이터를 잘 저장하는 방법에 대해 다루는 분야
- 해결하고자 하는 문제에 따라 적합한 방법을 선택해서 저장해야 함
- 사전 vs 단어장
 - 사전: 뜻이 궁금한 단어를 찾아보는 용도
 - 원하는 단어를 빠르게 찾을 수 있도록 사전 순으로 정렬해서 배치
 - 단어장: 단어를 암기하기 위해 보는 용도
 - 학습의 효율성을 위해 접사/어근/상황 등에 따라 묶어서 제공
- 도서관 이용자가 책을 찾는 상황
 - 원하는 책의 제목을 알고 있어서, 그 제목을 가진 찾아야 하는 사람
 - 책장에 책을 제목의 사전 순으로 배치하면 효율적
 - IT 전공 서적을 구경하고 싶은 사람
 - 십진 분류법에 따라 구역을 나누면 효율적
 - 100 철학, 200 종교, 300 사회과학, 400 자연과학, 500 기술과학, 600 예술, 700 언어, 800 문학, 900 역사

자료구조

자료구조

- 데이터를 잘 저장하는 방법에 대해 다루는 분야
- 해결하고자 하는 문제에 따라 적합한 방법을 선택해서 저장해야 함
- Python에서도...
 - 원소가 맨 뒤에 추가되고, 추가된 순서가 중요하면 List
 - 원소의 순서는 상관 없고, 중복된 원소를 제거해야 한다면 Set
 - key-value 쌍을 관리해야 한다면 Dictionary

자료구조

자료구조

- 데이터를 잘 저장하는 방법에 대해 다루는 분야
- 해결하고자 하는 문제에 따라 적합한 방법을 선택해서 저장해야 함
- 구간의 합을 구하는 문제
 - 단순히 구현하면 매번 $O(n)$ 에 구해야 함
 - 누적 합 배열을 사용하면 $O(n)$ 시간에 배열 한 번 만든 이후로 항상 $O(1)$ 시간에 계산 가능
 - 값이 바뀌는 쿼리도 함께 주어진다면?

- Prefix Sum:	전처리 n	값 변경 n	구간 합 1	매우 간단함
- Sqrt Decomposition:	전처리 n	값 변경 $\sqrt{n}$	구간 합 $\sqrt{n}$	간단함

자료구조

자료구조

- 데이터를 잘 저장하는 방법에 대해 다루는 분야
- 해결하고자 하는 문제에 따라 적합한 방법을 선택해서 저장해야 함
- 구간의 합을 구하는 문제
 - 단순히 구현하면 매번 $O(n)$ 에 구해야 함
 - 누적 합 배열을 사용하면 $O(n)$ 시간에 배열 한 번 만든 이후로 항상 $O(1)$ 시간에 계산 가능
 - 값이 바뀌는 쿼리도 함께 주어진다면?

- Prefix Sum:	전처리 n	값 변경 n	구간 합 1	매우 간단함
- Sqrt Decomposition:	전처리 n	값 변경 $\sqrt{n}$	구간 합 $\sqrt{n}$	간단함
- Segment Tree:	전처리 n	값 변경 $\log n$	구간 합 $\log n$	조금 복잡함
- Sqrt Tree:	전처리 $n \log \log n$	값 변경 $\sqrt{n}$	구간 합 1	복잡함

- 문제 상황에 맞는 자료구조를 잘 선택해야 함

목차

동적 배열

연결 리스트

해시 테이블

동적 배열

동적 배열

정적 배열 vs 동적 배열

- 정적 배열: 컴파일할 때 이미 크기가 정해져 있는 배열
- 동적 배열: 프로그램 실행 중에 원소가 추가/제거될 때마다 동적으로 크기가 바뀌는 배열

우리의 목표

- 배열의 맨 뒤에 원소를 추가하는 함수 `push_back(x)`
- 배열의 맨 뒤에 있는 원소를 삭제하는 함수 `pop_back()`
- 두 함수가 최대한 효율적으로 동작하도록 구현해 보기

동적 배열

방법 1

- push_back: 현재 배열보다 1 만큼 큰 배열 만들기
- pop_back: 현재 배열보다 1 만큼 작은 배열 만들기
- 시간 복잡도
 - 현재 배열에 있는 원소의 개수를 n이라고 했을 때
 - push_back: $O(n)$
 - pop_back: $O(n)$

```
int sz, *arr;

void init_array(){
    arr = NULL; sz = 0;
}

void push_back(int x){
    int *new_arr = (int*)malloc(sizeof(int) * (sz+1));
    for(int i=0; i<sz; i++) new_arr[i] = arr[i];
    new_arr[sz++] = x;
    free(arr); arr = new_arr;
}

void pop_back(){
    int *new_arr = (int*)malloc(sizeof(int) * (sz-1));
    for(int i=0; i<sz-1; i++) new_arr[i] = arr[i];
    free(arr); arr = new_arr; sz--;
}
```

동적 배열

방법 2

- pop_back 할 때는 새로운 배열을 만들 필요 없음
- sz: 현재 원소의 개수
- capacity: 할당 받은 배열의 크기
- push_back: sz = capacity면 capacity 1 증가
- pop_back: sz 1 감소
- 시간 복잡도
 - 현재 배열에 있는 원소의 개수를 n이라고 했을 때
 - push_back: 최악의 경우 $O(n)$
 - pop_back을 수행한 직후에는 $O(1)$
 - pop_back: $O(1)$
 - k번의 push_back할 때의 총 시간: $O(k^2)$

```
int sz, cap, *arr;

void init_array(){
    arr = NULL; sz = 0; cap = 0;
}

void push_back(int x){
    if(sz == cap){
        cap += 1;
        int *new_arr = (int*)malloc(sizeof(int) * cap);
        for(int i=0; i<sz; i++) new_arr[i] = arr[i];
        new_arr[sz++] = x;
        free(arr); arr = new_arr;
    }
    else arr[sz++] = x;
}

void pop_back(){
    sz--;
}
```

동적 배열

방법 3

- 굳이 1칸씩 늘릴 필요가 있을까? B칸씩 늘려보자.
- 시간 복잡도
 - n번의 push_back에서 필요한 총 연산 횟수
 - $k = qB + 1$ 번째 삽입: 재할당으로 인해 $O(k)$
 - 그렇지 않을 때의 삽입: $O(1)$
 - $n = qB + 1$ 꼴이면 n번의 push_back에서
 - $B * (1 + 2 + \dots + q) + n \in O(Bq^2 + n)$
 - $Bq = n-1$ 이므로 $O(nq + n)$ 으로 나타낼 수 있음
 - 이때 B는 상수이므로 $q = n/B$ 는 $O(n)$
 - 따라서 $O(nq + n) \in O(n^2)$
 - B의 값이 클수록 재할당 횟수가 줄어들어서 성능 개선이 되지만
 - n이 충분히 크면 B에 관계 없이 $O(n^2)$

동적 배열

방법 4 – array doubling

- size = capacity 일 때 capacity를 2배로 늘리면 n번의 push_back을 $O(n)$ 에 처리할 수 있음
- $n = 2^k + 1$ 번의 삽입에서 필요한 연산 횟수를 계산해 보자.
 - 데이터를 저장하는 연산(`arr[sz++] = v;`)은 총 n번 수행
 - $2^0 + 1, 2^1 + 1, \dots, 2^{k-1} + 1, 2^k + 1$ 번째 삽입에서 각각 $2^0, 2^1, 2^2, \dots, 2^{k-1}, 2^k$ 번의 복사 수행
 - $2^0, 2^1, 2^2, \dots, 2^{k-1}, 2^k = 2^{k+1} - 1$ 이므로 $n = 2^k + 1$ 번의 삽입에서 복사는 총 $2n - 3$ 번 수행
 - 따라서 전체 연산 횟수는 $3n - 3$.
- 최악의 경우에도 n번의 push_back에서 걸리는 시간은 $O(n)$
- 따라서 연산당 $O(1)$ 이라고 생각할 수 있지 않을까?
- amortized $O(1)$

동적 배열

```
template<typename T>
class dynamic_array{
private:
    T *arr;
    size_t sz, cap;
    void set_capacity(size_t new_cap){
        T *new_arr = new T[new_cap];
        for(int i=0; i<min(sz, new_cap); i++) new_arr[i] = arr[i];
        if(arr) delete[] arr;
        arr = new_arr; cap = new_cap;
    }
public:
    dynamic_array(){
        arr = nullptr; sz = 0; set_capacity(0);
    }
    void push_back(const T x){
        if(sz == cap) set_capacity(cap == 0 ? 1 : cap * 2);
        arr[sz++] = x;
    }
    void pop_back(){
        sz--;
    }
    size_t size() const { return sz; }
    size_t capacity() const { return cap; }
    T& operator [] (const size_t index) { return arr[index]; }
    const T& operator [] (const size_t index) const { return arr[index]; }
};
```

동적 배열

std::vector

- 앞에서 다룬 동적 배열이 구현되어 있음
 - 헤더: <vector>
 - 클래스: vector<Type>
 - int형 자료를 관리하는 큐: std::vector<int>
 - 두 번째 파라미터도 있는데 지금을 신경 쓸 필요 없음
 - 멤버 함수
 - push_back(value) , pop_back()
 - front(), back()
 - empty(), size()
 - resize(), reserve()
 - 인덱스 접근 가능

동적 배열



```
void test(){
    vector<int> vec; // int형 변수를 저장하는 빈 동적 배열 생성
    /*
    vector<int> vec = {1, 2, 3};
    vector<int> vec(4);           // 크기가 4인 동적 배열
    vector<int> vec(4, 5);        // 크기가 4고, 모든 원소가 5로 초기화된 동적 배열
    */

    for(int i=0; i<10; i++) vec.push_back(i);
    cout << "size : " << vec.size() << "\n"; // size : 10
    cout << "front : " << vec.front() << "\n"; // front : 0
    cout << "back : " << vec.back() << "\n"; // back : 9
    vec.pop_back();
    cout << "back(after pop_back) : " << vec.back() << "\n"; // back : 8

    vec.resize(20); // 크기 변경, 크기를 더 키운다면 남은 공간은 기본값(0)으로 초기화
    vec.resize(5); // 크기 변경, 크기를 더 줄인다면 앞에 있는 원소를 남김

    // 0 1 2 3 4
    for(int i=0; i<vec.size(); i++) cout << vec[i] << " ";
    cout << "\n";
}
```

동적 배열

아직 남아있는 비효율성

- `pop_back`을 여러 번 수행해서 사용하지 않는 공간이 남아있더라도 할당받은 공간을 해제하지 않음
- 어떻게 하면 시간 복잡도를 amortized $O(1)$ 로 유지하면서 메모리를 절약할 수 있을까?
- 시도 1. `pop_back`으로 인해 `size`가 `capacity`의 절반 이하가 되면 `capacity`를 절반으로 축소
 - `size = capacity`인 상황에서 `push_back`과 `pop_back`을 번갈아 가며 호출하면 매번 $O(n)$ 이라서 안 됨
- 시도 2. `size`가 `capacity/4` 이하가 되면 `capacity`를 절반으로 축소
 - 이 경우 삽입과 삭제를 총 n 번 수행할 때, 최악의 경우에도 전체 연산 횟수가 $O(n)$ 이 됨을 증명할 수 있음
 - 증명 방법을 고민해 보자.

resize

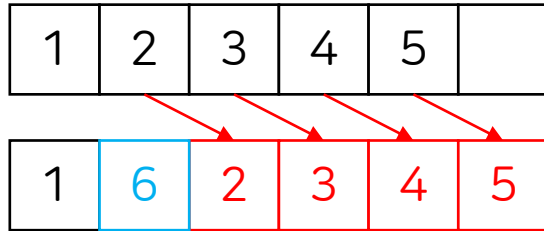
- `resize`를 이용해 크기를 늘리거나 줄일 때, 어떻게 구현해야 amortized $O(1)$ 을 유지할 수 있을까?

연결 리스트

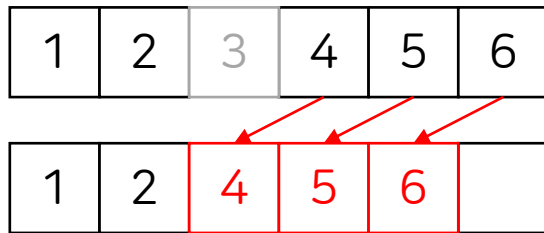
배열의 단점

배열 중간에서 원소 삽입/삭제

- 원소를 삽입하는 경우: 삽입한 위치 이후에 있는 원소를 모두 한 칸씩 뒤로 옮겨야 함



- 원소를 삭제하는 경우: 삭제한 위치 이후에 있는 원소를 모두 한 칸씩 앞으로 옮겨야 함



- 두 연산 모두 최악의 경우 $O(n)$

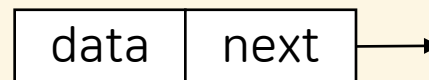
자기 참조 구조체

자기 참조 구조체

- 자기 자신과 같은 자료형을 포함하는 구조체
- 하지만 자기 자신과 같은 타입 그 자체를 포함하면 구조체의 크기가 무한히 커지기 때문에 포인터로 저장



```
typedef struct node{  
    int data;  
    struct node *next;  
} Node;  
Node *front;
```



```
Node* new_node(int v){  
    Node *node = (Node*)malloc(sizeof(Node));  
    node->data = v;  
    node->next = NULL;  
    return node;  
}
```

연결 리스트

연결 리스트

- 각 노드는 값(data)과 바로 다음 원소를 가리키는 포인터(next)로 구성
- 마지막 원소의 next는 NULL



```
void init_list(int n){  
    // 0, 1, 2, ... , n-1로 구성된 리스트 생성  
    Node *next = NULL;  
    for(int i=n-1; i>=0; i--){  
        front = new_node(i);  
        front->next = next; next = front;  
    }  
}
```

연결 리스트

연결 리스트

- 각 노드는 값(data)와 바로 다음 원소를 가리키는 포인터(next)로 구성
- 마지막 원소의 next는 NULL

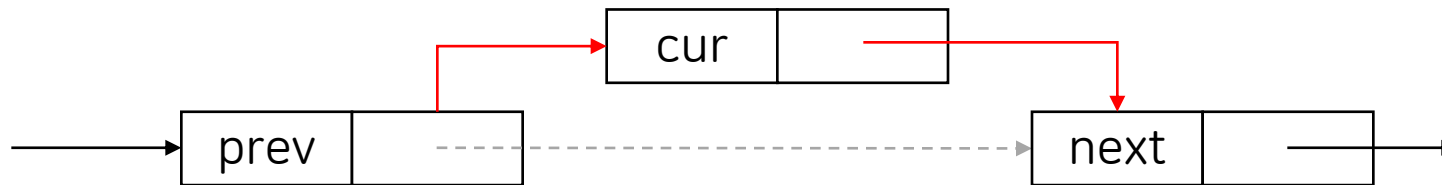


```
void print_list(){  
    // 리스트의 모든 원소를 차례대로 출력  
    for(Node *cur=front; cur!=NULL; cur=cur->next){  
        printf("%d ", cur->data);  
    }  
    printf("\n");  
}  
  
Node* get_by_index(int k){  
    // k번째 원소(0-based) 반환  
    Node *cur = front;  
    for(int i=0; i<k; i++) cur = cur->next;  
    return cur;  
}
```

연결 리스트

연결 리스트

- 원하는 위치에 원소 삽입을 $O(1)$ 시간에 할 수 있음



```
void insert_element(Node *prev, int v){  
    // prev 바로 다음에 원소 삽입  
    Node *cur = new_node(v);  
    cur->next = prev->next;  
    prev->next = cur;  
}
```

연결 리스트

연결 리스트

- 원하는 위치에 있는 원소를 $O(1)$ 시간에 삭제할 수 있음



```
void delete_element(Node *prev){  
    // prev 바로 다음에 있는 원소 제거  
    Node *cur = prev->next;  
    prev->next = cur->next;  
    free(cur);  
}
```

연결 리스트

배열 vs 연결 리스트

- 배열의 장점
 - 특정 인덱스에 있는 원소를 $O(1)$ 시간에 참조할 수 있음
 - 포인터를 사용하지 않으므로 구현이 간단함
- 연결 리스트의 장점
 - 원하는 위치에 원소 삽입/삭제를 $O(1)$ 시간에 할 수 있음
 - 실제로 필요한 만큼만 메모리를 할당 받아서 사용하므로, 낭비되는 메모리가 없음
- 주의 사항
 - 원하는 위치에서 삽입/삭제하기 위해서는 $O(n)$ 시간에 해당 위치에 있는 원소를 찾아야 함
 - 즉, 삽입/삭제하고자 하는 원소의 위치를 알고 있을 때 $O(1)$ 에 할 수 있다는 뜻

연결 리스트

이중 연결 리스트

- 양방향 탐색이 가능하도록 현재 노드의 이전 노드를 가리키는 포인터를 추가한 자료구조



```
typedef struct node{
    int data;
    struct node *prev;
    struct node *next;
} Node;

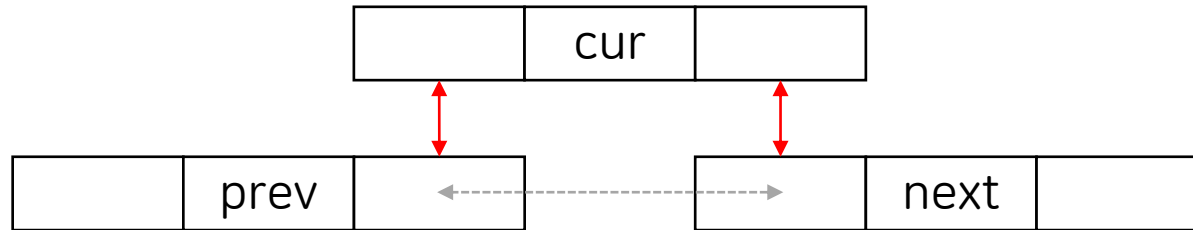
Node *front, *back;

Node* new_node(int v){
    Node *node = (Node*)malloc(sizeof(Node));
    node->data = v;
    node->prev = NULL;
    node->next = NULL;
    return node;
}
```

연결 리스트

이중 연결 리스트

- 원소 삽입



```
void insert_element(Node *prev, int v){  
    // prev 바로 다음에 원소 삽입  
    Node *next = prev->next;  
    Node *cur = new_node(v);  
    cur->prev = prev;  
    prev->next = cur;  
    cur->next = next;  
    next->prev = cur;  
}
```

연결 리스트

이중 연결 리스트

- 원소 삭제



```
void delete_element(Node *cur){  
    // cur 제거  
    cur->prev->next = cur->next;  
    cur->next->prev = cur->prev;  
    free(cur);  
}
```

연결 리스트

std::list

- 이중 연결 리스트로 구현되어 있음
 - 헤더: <list>
 - 클래스: list<Type>
 - int형 자료를 관리하는 큐: std::list<int>
 - 두 번째 파라미터도 있는데 지금을 신경 쓸 필요 없음
 - 멤버 함수
 - push_front(value), push_back(value)
 - pop_front(), pop_back()
 - front(), back()
 - empty(), size()

해시 테이블

해시 테이블

배열의 일반화

- 배열의 인덱스로 다른 걸 넣을 수는 없을까?
- 예를 들면 아주 큰 수(10^9 , 10^{18} 등), 문자열, 구조체 등

해싱

- 해시 함수: 임의의 길이를 갖는 임의의 데이터를 고정된 길이의 데이터로 매핑하는 함수
 - x 가 정수일 때 $f(x) = x \% M$ 은 x 의 크기에 관계 없이 항상 $[0, M)$ 사이의 정수를 반환함
 - s 가 문자열일 때 $f(s) = s[0] * 128 + s[1]$ 은 s 의 길이에 관계 없이 항상 $[0, 128^2)$ 사이의 정수를 반환함
- 해시 함수의 조건
 - input이 같으면 여러 번 실행해도 항상 같은 결과가 나와야 함
 - input이 다르면 output이 다를 확률이 높을수록 좋음
 - 임의 길이 데이터를 고정된 길이의 데이터로 매칭하므로 일대일 대응은 안 될 수 있음
 - $x \neq y$ 이면서 $f(x) = f(y)$ 인 것을 "해시 충돌"이라고 부름

해시 테이블

해시 테이블

- 해시 테이블로 사용할 배열의 크기를 M 이라고 하자.
- 키(key)들의 집합을 U 라고 할 때, $h: U \rightarrow \{0, 1, \dots, M-1\}$ 인 해시 함수 h 를 하나 준비하자.
- 해시 테이블은 키가 k 인 원소를 배열의 $h(k)$ 번 인덱스에 저장한다.
- 예시
 - $M = 5$
 - $h(x) = 13x + 2 \bmod 5$
 - 1, 2, 4, 8 저장

0	1
1	8
2	/
3	2
4	4

해시 테이블

해시 테이블

- 해시 테이블로 사용할 배열의 크기를 M 이라고 하자.
- 키(key)들의 집합을 U 라고 할 때, $h: U \rightarrow \{0, 1, \dots, M-1\}$ 인 해시 함수 h 를 하나 준비하자.
- 해시 테이블은 키가 k 인 원소를 배열의 $h(k)$ 번 인덱스에 저장한다.

- 예시

- $M = 5$
- $h(x) = 13x + 2 \bmod 5$
- 1, 2, 4, 8 저장
- 7도 저장하려고 했는데...?
- $f(2) = f(7) = 3$ 해시 충돌 발생

0	1
1	8
2	/
3	2
4	4

7

해시 테이블

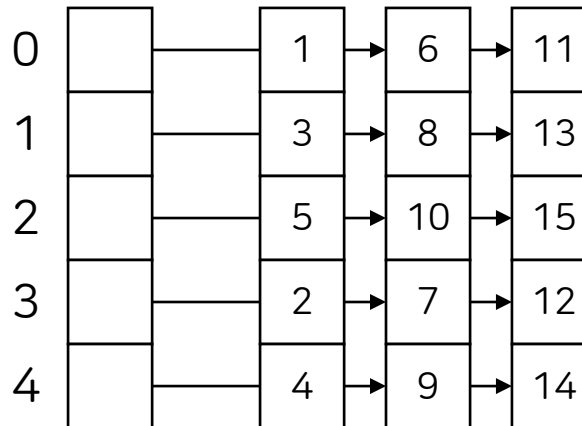
해시 충돌 해결 전략

- 임의 길이의 데이터를 고정된 길이로 매칭하기 때문에 해시 충돌은 불가피함
- 따라서 해시 충돌이 발생했을 때 데이터를 저장하는 전략을 세워야 함
- Chaining
 - 단순 체이닝
 - Java HashMap의 방식
- Open addressing
 - Linear probing
 - Quadratic probing
 - Double hash

해시 테이블

해시 충돌 해결 전략 - Chaining

- 해시 테이블의 각 칸마다 연결 리스트(또는 동적 배열) 관리
- $M = 5$
- $h(x) = 13x + 2 \bmod 5$
- $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$
- 모든 키의 해시 값이 같으면 탐색에 $O(n)$ 소요
- 해시 함수가 충분히 균등하다고 가정하면 n 개의 원소가 있을 때 $O(n/M)$



해시 테이블

해시 충돌 해결 전략 - Chaining



```
constexpr int M = 5;
vector<int> H[M];
int f(int x){ return (13 * x + 2) % 5; }

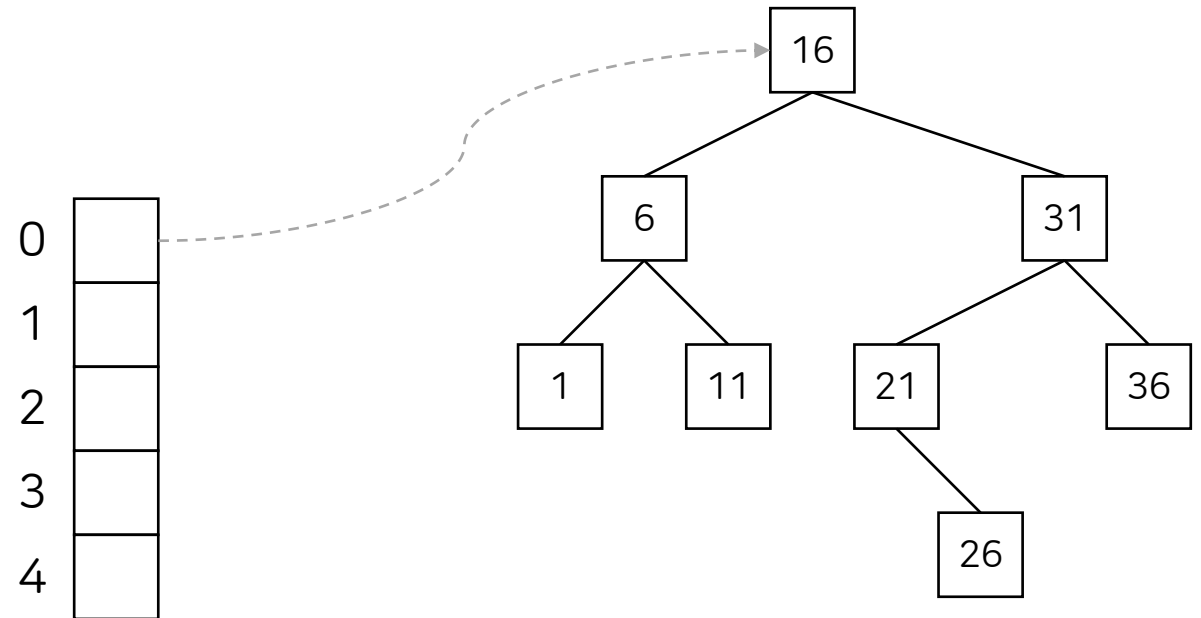
void insert(int x){
    H[f(x)].push_back(x);
}

bool exist(int x){
    for(auto i : H[f(x)]) if(i == x) return true;
    return false;
}
```

해시 테이블

해시 충돌 해결 전략 - Chaining(Java의 HashMap 구현)

- 해시 테이블의 각 칸마다 연결 리스트 관리
- 연결 리스트의 크기가 특정 값(주로 8) 이상이 되면 연결 리스트 대신 다른 자료구조를 사용
 - 삽입과 삭제의 시간 복잡도가 $O(\log n)$ 이 보장되는 자료구조(BBST) 사용
- $M = 5$
- $h(x) = 13x + 2 \pmod{5}$
- $\{1, 6, 11, 16, 21, 26, 31, 36\}$
- 이상적인 상황에서 $O(n/M)$
- 최악의 경우에도 $O(\log n)$



해시 테이블

해시 충돌 해결 전략 - Open addressing

- 해시 테이블의 크기가 삽입하고자 하는 원소의 개수보다 클 때 사용할 수 있는 방법
- 체이닝과 다르게 모든 원소가 테이블에 직접 저장됨
- 즉, 포인터를 사용하지 않음
- 해시 함수 $h: U \times \{0, 1, \dots, M-1\} \rightarrow \{0, 1, \dots, M-1\}$ 을 생각하자.
 - k 를 삽입할 때, 가장 먼저 $h(k, 0)$ 번 인덱스를 확인
 - $h(k, 0)$ 번 인덱스가 비어있다면 그 위치에 삽입
 - 그렇지 않으면 $h(k, 1)$ 번 인덱스 확인 후 삽입
 - $h(k, 2), h(k, 3), \dots$ 차례대로 확인
- $h(k, 0), h(k, 1), \dots, h(k, M-1)$ 에 중복이 없을수록 유리함
 - 중복이 있으면 해시 테이블이 가득 차지 않은 상황에서도 삽입에 실패할 수 있음

해시 테이블

해시 충돌 해결 전략 - Open addressing



```
constexpr int M = 37;
int f(int k, int i){ return (k * 3 + i * 2) % M; }
int H[M];

void insert(int x){
    for(int i=0; i<M; i++){
        int h = f(x, i);
        if(H[h] == 0){ H[h] = x; return; }
    }
    assert(false); // full
}

bool exist(int x){
    for(int i=0; i<M; i++){
        int h = f(x, i);
        if(H[h] == 0) break;
        if(H[h] == x) return true;
    }
    return false;
}
```

해시 테이블

해시 충돌 해결 전략 - Open addressing

- 삭제할 때 단순히 원소를 지우면(0으로 바꾸면) 안 됨
- 왜?
- DELETED 상태를 만들어서, 지워진 원소가 있던 위치에는 DELETED를 저장해야 함

해시 테이블

해시 충돌 해결 전략 - Open addressing

- 해시 함수 $h: U \times \{0, 1, \dots, M-1\} \rightarrow \{0, 1, \dots, M-1\}$
- Linear probing (선형 조사)
 - $h(k, *)$ 에 중복이 없음을 보장하는 가장 쉬운 방법
 - 보조 해시 함수 $h': U \rightarrow \{0, 1, 2, \dots, M-1\}$ 을 하나 만들자.
 - $h(k, i) = (h'(k) + i) \bmod M$
 - 장점: $h(k, *)$ 의 값이 서로 다름이 보장됨
 - 단점: 데이터가 연속으로 길게 저장되어 성능이 저하될 수 있음
 - 현재 x 개의 공간이 연속으로 채워져 있다면, 원소 삽입할 때 $(x+1)/M$ 의 확률로 연속 공간 다음 자리에 저장됨
 - 1차 군집 문제

해시 테이블

해시 충돌 해결 전략 - Open addressing

- 해시 함수 $h: U \times \{0, 1, \dots, M-1\} \rightarrow \{0, 1, \dots, M-1\}$
- Quadratic probing (2차 조사)
 - 보조 해시 함수 $h' : U \rightarrow \{0, 1, 2, \dots, M-1\}$ 와 보조 상수 c_1, c_2
 - $h(k, i) = (h'(k) + c_1i + c_2i^2) \bmod M$
 - 장점: 1차 군집 문제가 비교적 덜 나타남
 - $h(k_1, 0) = h(k_2, 0)$ 이면 모든 i 에 대해 $h(k_1, i) = h(k_2, i)$ 성립 $\rightarrow$ 2차 군집 문제
 - 초기 조사 위치가 전체 조사 순서를 결정 $\rightarrow$ 조사 순서의 종류는 최대 M 가지
 - c_1, c_2, M 의 값을 잘 정해야 함

해시 테이블

해시 충돌 해결 전략 - Open addressing

- 해시 함수 $h: U \times \{0, 1, \dots, M-1\} \rightarrow \{0, 1, \dots, M-1\}$
- Double hash (2중 해싱)
 - 보조 함수 2개 사용
 - $h(k, i) = (h_1(k) + i * h_2(k)) \bmod M$
 - $h_2(k)$ 는 M과 서로소인 값이어야 함
 - M은 소수, $h_2(k)$ 는 항상 M보다 작은 값을 반환하도록 하면 쉽게 조건을 만족할 수 있음
 - $h_2(k)$ 와 M이 서로소가 아니면 $h(k, i)$ 가 M개의 칸을 모두 보지 않게 됨
 - 선형, 이차 조사와 다르게 초기 조사 위치가 같더라도 이후 조사 순서가 다를 수 있음

해시 테이블

`std::unordered_set`

- `#include <unordered_set>`
- 집합을 관리하는 컨테이너
- 원소를 중복 없이 저장함
- `insert(x)`, `erase(x)`, `find(x)`, `count(x)`
 - 평균 $O(1)$, 최악 $O(n)$
- `size()`, `empty()`
- `unordered_set`에서 `count()`의 값은 항상 0 또는 1
- `unordered_multiset`는 중복 허용, `count()`의 값이 1 초과일 수 있음

해시 테이블

std::unordered_set



```
void test(){
    unordered_set<int> st;
    for(int i=0; i<5; i++) st.insert(i*2);

    cout << st.count(0) << " " << st.count(1) << "\n"; // 1 0
    assert(st.find(1) == st.end()); // 없으면 st.end() 반환

    st.erase(0);
    cout << st.count(0) << " " << st.size() << "\n"; // 0 4

    st.erase(5); // 없는 원소를 지우려고 하면 무시됨
    for(auto i : st) cout << i << " "; // 0 2 4 6 8이 임의의 순서대로 출력
}
```

해시 테이블

std::unordered_map

- `#include <unordered_map>`
- key-value 쌍을 관리하는 컨테이너
 - key의 중복과 수정을 허용하지 않음
 - value는 자유롭게 변경 가능
 - key에 대해서 std::unordered_set의 모든 연산 수행 가능
- 배열과 비슷한 인터페이스를 갖고 있음
 - 원소 삽입/수정: `mp[key] = value; mp[key] += 1; 등`
 - value 구하기: `mp[key]`
- `mp[key]`를 이용해 존재하지 않는 key에 접근하는 경우 기본값으로 초기화
 - int와 같은 자료형은 0으로 초기화

해시 테이블

std::unordered_map



```
void test(){
    unordered_map<int, int> mp;
    for(int i=0; i<5; i++) mp[i*2] = i*2+1;
    mp.insert({10, 11});
    mp.erase(4);
    mp[0] += 100;

    // (0, 100) (2, 3) (6, 7) (8, 9) (10, 11)
    for(auto [key,value] : mp) cout << key << " " << value << "\n";
}
```