Hash

Data Structures and Algorithms

목차

- 빠른 탐색: 해쉬 테이블
- 예제
- 좋은 해쉬 함수
- 충돌의 해결
- 구현
- 응용: Bloom Filter

빠른 탐색: 해쉬 테이블

테이블: Key-Value의 집합

- 사전 구조 또는 맵(map)

- 데이터는 key와 value가 한 쌍
  - key가 데이터의 저장 및 탐색 기준
  - 테이블 자료구조에서 원하는 데이터를 단번에 찾음

- 복잡도: O(1)

사번 : key	직원 : value
99001	양현석 부장
99002	한상현 차장
99003	이현진 과장
99004	이수진 사원

테이블 자료구조의 예

배열 기반 테이블 (해시 없는 개념 이해 용)

```
typedef struct _empInfo
{
     int empNum;  // Key: 직원의 고유번호
     int age;     // value: 직원의 나이
}EmpInfo;

int main(void)
{
     EmpInfo empInfoArr[1000];
     EmpInfo ei;
     int eNum;

     printf("사번과 나이 입력: ");
     scanf("%d %d", &(ei.empNum), &(ei.age));
     empInfoArr[ei.empNum] = ei;    // 단번에 저장!

     printf("확인하고픈 직원의 사번 입력: ");
     scanf("%d", &eNum);

     ei = empInfoArr[eNum];    // 단번에 탐색!
     printf("사번 %d, 나이 %d \n", ei.empNum, ei.age);
     return 0;
}
```

직원 고유번호(key)를 배열의 인덱스로 활용

해쉬 함수와 충돌

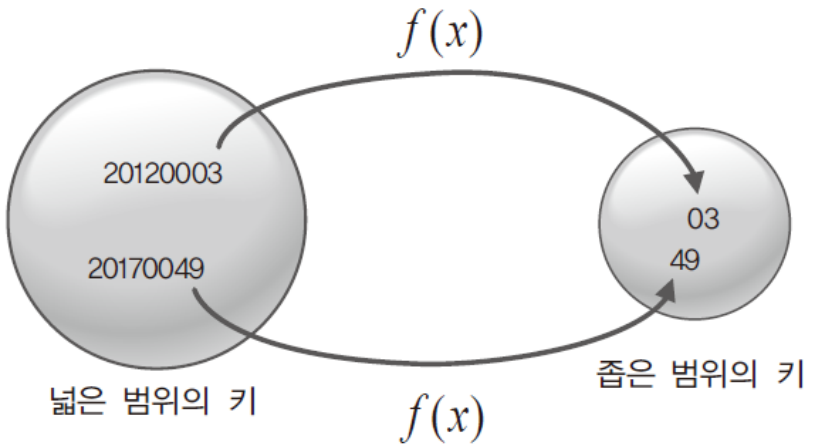
```
int main(void)
{
	EmpInfo empInfoArr[100];

	EmpInfo emp1={20120003, 42};
	EmpInfo emp2={20130012, 33};
	EmpInfo emp3={20170049, 27};

	EmpInfo r1, r2, r3;

	empInfoArr[GetHashValue(emp1.empNum)] = emp1; // 키를 인덱스로 활용
	empInfoArr[GetHashValue(emp2.empNum)] = emp2;
	empInfoArr[GetHashValue(emp3.empNum)] = emp3;

	r1 = empInfoArr[GetHashValue(20120003)]; // 키를 인덱스로 탐색
	r2 = empInfoArr[GetHashValue(20130012)];
	r3 = empInfoArr[GetHashValue(20170049)];

	printf("사번 %d, 나이 %d \n", r1.empNum, r1.age); // 탐색 결과 확인
	printf("사번 %d, 나이 %d \n", r2.empNum, r2.age);
	printf("사번 %d, 나이 %d \n", r3.empNum, r3.age);
	return 0;
}
```

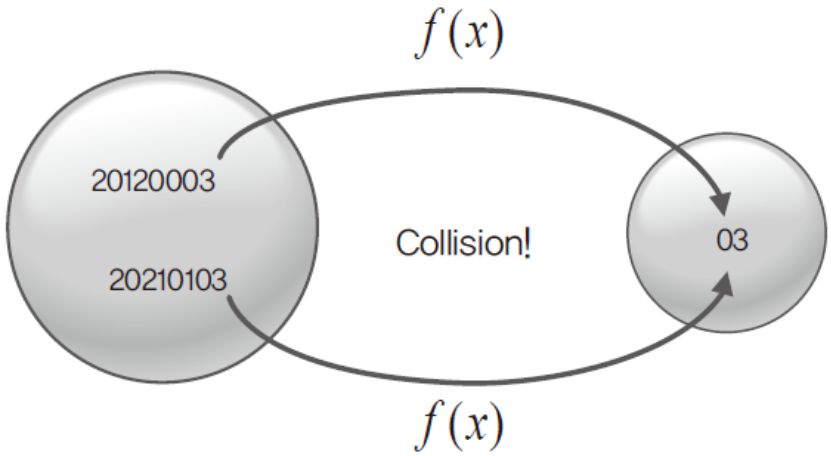
```
// hash 함수 f(x)
int GetHashValue(int empNum)
{
    return empNum % 100;
}
```

해쉬 함수와 충돌

```
EmpInfo emp1={20120003, 42};
EmpInfo emp2={20130012, 33};
EmpInfo emp3={20170049, 27};
r1 = empInfoArr[GetHashValue(20120003)];
r2 = empInfoArr[GetHashValue(20130012)];
r3 = empInfoArr[GetHashValue(20170049)];
```

```
int GetHashValue(int empNum)
{
    return empNum % 100;
}
```





f(x)는 키를 기준 범위 내에 재배치함
단, 기준 범위가 좁으면 다른 key이지만 같은 공간에 저장될 수 있음 (충돌 발생)

예제

Ex: Person

- 테이블 정의
  - 키 : 주민등록 번호
  - 값 : 구조체 변수의 주소 값

```
typedef struct _person
{
    int ssn;    // 주민등록번호
    char name[STR_LEN];    // 이름
    char addr[STR_LEN];    // 주소
} Person;

int GetSSN(Person * p);
void ShowPerInfo(Person * p);
Person * MakePersonData(int ssn,
            char * name, char * addr);
```

```
int GetSSN(Person * p){
    return p->ssn;
}

void ShowPerInfo(Person * p){
    printf("주민등록번호: %d \n", p->ssn);
    printf("이름: %s \n", p->name);
    printf("주소: %s \n\n", p->addr);
}

Person * MakePersonData(int ssn,
                char * name, char * addr){
    Person * newP = \
           (Person*)malloc(sizeof(Person));

    newP->ssn = ssn;
    strcpy(newP->name, name);
    strcpy(newP->addr, addr);
    return newP;
}
```

Ex: Person - Slot

- 슬롯 상태: EMPTY, DELETED, INUSE

```
typedef int Key;
typedef Person * Value;

enum SlotStatus {EMPTY, DELETED, INUSE};

typedef struct _slot
{
    Key key;
    Value val;
    enum SlotStatus status;
}Slot;
```

사번 : key	직원 : value
99001	양현석 부장
99002	한상현 차장
99003	이현진 과장
99004	이수진 사원

슬롯

Ex: Person – 해쉬 함수와 초기화

```
typedef int HashFunc(Key k);

typedef struct _table
{
    Slot tbl[MAX_TBL];
    HashFunc * hf;
}Table;

// 테이블의 초기화
void TBLInit(Table * pt, HashFunc * f);

// 테이블에 키와 값을 저장
void TBLInsert(Table * pt, Key k, Value v);

// 키를 근거로 테이블에서 데이터 삭제
Value TBLDelete(Table * pt, Key k);

// 키를 근거로 테이블에서 데이터 탐색
Value TBLSearch(Table * pt, Key k);
```

```
void TBLInit(Table * pt, HashFunc * f)
{
    int i;
    // 슬롯 초기화
    for(i=0; i<MAX_TBL; i++)
        (pt->tbl[i]).status = EMPTY;

    pt->hf = f; // 해쉬 함수 등록
}
```

Ex: Person: 해쉬 함수

```
void TBLInsert(Table * pt, Key k, Value v){
    int hv = pt->hf(k); // 해쉬 값을 얻음
    pt->tbl[hv].val = v;
    pt->tbl[hv].key = k;
    pt->tbl[hv].status = INUSE;
}

Value TBLDelete(Table * pt, Key k){
    int hv = pt->hf(k); // 해쉬 값을 얻음
    if((pt->tbl[hv]).status != INUSE){
        return NULL;
    } else {
        (pt->tbl[hv]).status = DELETED;
        return (pt->tbl[hv]).val; // 삭제된 데이터 리턴
    }
}

Value TBLSearch(Table * pt, Key k){
    int hv = pt->hf(k); // 해쉬 값을 얻음
    if((pt->tbl[hv]).status != INUSE)
        return NULL;
    else
        return (pt->tbl[hv]).val; // 탐색 대상 리턴
}
```

Ex: Person - Main

```
int MyHashFunc(int k){
    // 키를 부분적으로만 사용한
    // 않은 해쉬의 예!!!
    return k % 100;
}

int main(void)
{
   Table myTbl;
   Person * np; Person * sp; Person * rp;

   TBLInit(&myTbl, MyHashFunc);

   // 데이터 입력
   np = MakePersonData(20120003, "Lee", "Seoul");
   TBLInsert(&myTbl, GetSSN(np), np);

   np = MakePersonData(20130012, "KIM", "Jeju");
   TBLInsert(&myTbl, GetSSN(np), np);

   np = MakePersonData(20170049, "HAN", "Kangwon");
   TBLInsert(&myTbl, GetSSN(np), np);

   // 데이터 검색
   sp = TBLSearch(&myTbl, 20120003);
   if(sp != NULL) ShowPerInfo(sp);

   sp = TBLSearch(&myTbl, 20130012);
   if(sp != NULL) ShowPerInfo(sp);

   sp = TBLSearch(&myTbl, 20170049);
   if(sp != NULL) ShowPerInfo(sp);

   // 데이터 삭제
   rp = TBLDelete(&myTbl, 20120003);
   if(rp != NULL) free(rp);

   rp = TBLDelete(&myTbl, 20130012);
   if(rp != NULL) free(rp);

   rp = TBLDelete(&myTbl, 20170049);
   if(rp != NULL) free(rp);

   return 0;
}
```

좋은 해쉬 함수

좋은 해쉬함수

- 키 전체를 참조하여 해쉬 값을 생성
  - 키 스페이스를 모두 조합하여 해쉬 값 생성시 결과가 고르게 분포할 것이다는 기대

특정 위치에 몰림

Vs

분산된 저장 위치

자릿수 선택 방법

- 가정
  - 키 스페이스 분석 가능 및 완료

- 전략
  - 중복 비율이 높거나 공통인 부분을 제외

- 예: 여덟 자리의 수로 이뤄진 키 스페이스
  - 해쉬 값에 영향을 주는 네자리 수를 뽑아 생성

자릿수 폴딩 방법

- 키 스페이스의 연산을 통해 저장할 위치 지정
  - 정해진 규칙은 없음
  - 필요에 따라 다양한 근거로 생성

- 예

27 + 34 + 19 = 80

충돌의 해결

선형 조사법 (Linear Probing)

- 단순함
- 충돌의 횟수 증가에 따라 클러스터 현상 발생
  - 클러스터 현상: 특정 영역에 데이터가 몰리는 현상

충돌 없을 때: f(k) = k % 7

충돌 있을 때: f(k) = k % 7 + n, 이 때 n은.충돌 횟수

$f(k)+1 \rightarrow f(k)+2 \rightarrow f(k)+3 \rightarrow f(k)+4 \cdots$

K = 9

K%7 + 1 = 2

K = 2

K%7 + 1^2 = 3

2차 조사법 + DELETED

- 선형 조사법 보다 멀리서 빈자리 찾음

$$f(k)+1^2 \rightarrow f(k)+2^2 \rightarrow f(k)+3^2 \rightarrow f(k)+4^2 \cdots$$

K = 9

K%7 + 1 = 2

K = 2

K%7 + 1^2 = 3

K = 9
삭제

K%7 + 1 = 2

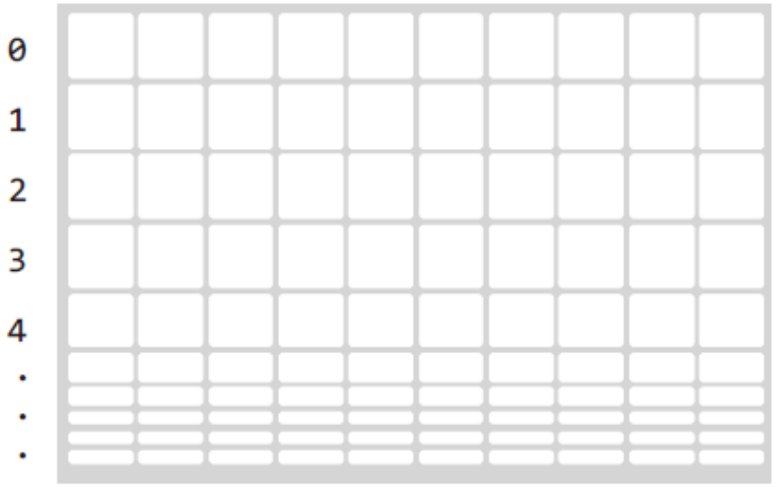
DELETED 표시해야 동일한 해쉬 값의
데이터 저장 검사 가능

이중 해시

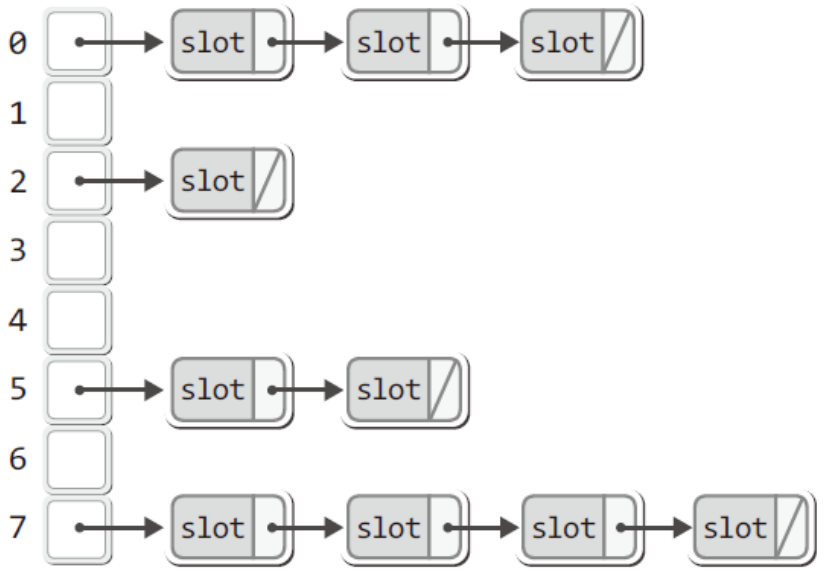
- 빈 슬롯의 위치가 늘 동일한 문제
- 다른 해쉬 함수를 활용하는 방법
  - 1차 해쉬 함수 $h1(k)=k \% 15$ 배열의 길이가 15인 경우의 예
  - 2차 해쉬 함수 $h2(k)=1+(k \% c)$ 15보다 작은 소수로 c를 결정
- 예
  - 1차 해쉬 함수 $h1(k)=k \% 15$
  - 2차 해쉬 함수 $h2(k)=1+(k \% 7)$
  - 1을 더하는 이유: 2차 해쉬 값이 0이 안되도록
  - c를 15보다 작은 값으로 하는 이유: 배열의 길이가 15이므로
  - c를 소수로 결정하는 이유: 클러스터 현상을 낮춘다는 통계를 근거로!

체이닝(닫힌 어드레싱 모델)

- 한 해쉬 값에 다수의 데이터를 저장할 수 있도록 배열을 2차원의 형태로 선언하는 모델
- 한 해쉬 값에 다수의 데이터를 저장할 수 있도록 각 해쉬 값 별로 연결 리스트를 구성하는 모델



2차원 배열 모델



연결리스트 모델

구현

구현: 체이닝

- 슬롯에 저장할 데이터 관련 헤더 및 소스파일
  - Person.h, Person.c
- 테이블 관련 헤더 및 소스 파일
  - Slot.h, Table.h, Table.c

- 확장 대상
  - Slot.h, Table.[ch]

- 활용 할 개념: 해쉬 값 별 연결 리스트 구성 용
  - 이중 연결 리스트 (DLinkedList.[ch])

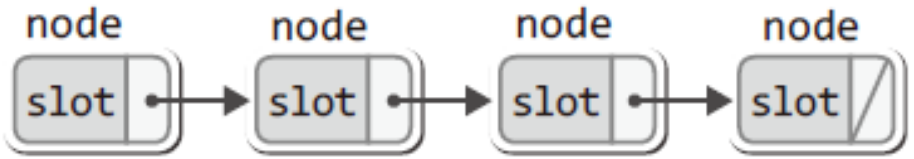
확장: 슬롯 (Slot.h)

- 체이닝 기반: 슬롯 상태 비유지

```
typedef int Key;
typedef Person * Value;

enum SlotStatus {EMPTY, DELETED, INUSE};

typedef struct _slot
{
    Key key;
    Value val;
    enum SlotStatus status;
}Slot;
```

확장: 테이블

```
typedef int HashFunc(Key k);

typedef struct _table
{
    // 해쉬 값 별로 리스트 구성
    Slot List tbl[MAX_TBL];
    HashFunc * hf;
}Table;

// 테이블의 초기화
void TBLInit(Table * pt, HashFunc * f);

// 테이블에 키와 값을 저장
void TBLInsert(Table * pt, Key k, Value v);

// 키를 근거로 테이블에서 데이터 삭제
Value TBLDelete(Table * pt, Key k);

// 키를 근거로 테이블에서 데이터 탐색
Value TBLSearch(Table * pt, Key k);
```



노드의 데이터가 슬롯
리스트의 활용

확장: 연결 리스트

```
#include "Slot2.h" // 헤더파일 추가

// typedef int LData;
typedef Slot LData; // 변경된 LData 타입

typedef struct _node
{
	LData data;
	struct _node * next;
} Node;

typedef struct _linkedList
{
	Node * head;
	Node * cur;
	Node * before;
	int numOfData;
	int (*comp)(LData d1, LData d2);
} LinkedList;
```

확장: 초기화와 삽입 연산

```
void TBLInit(Table * pt, HashFunc * f)
{
     int i;
     for(i=0; i<MAX_TBL; i++)
         ListInit(&(pt->tbl[i])); // 연결리스트 각각에 대해서 초기화
     pt->hf = f;
}

void TBLInsert(Table * pt, Key k, Value v)
{
     int hv = pt->hf(k);
     Slot ns = {k, v};
     if(TBLSearch(pt, k) != NULL) {        // 키가 중복되었다면
           printf("키 중복 오류 발생 \n");
           return;
     } else {
           LInsert(&(pt->tbl[hv]), ns); // 해쉬 값 기반 삽입
     }
}
```

확장: 삭제

```
Value TBLDelete(Table * pt, Key k)
{
    int hv = pt->hf(k);
    Slot cSlot;

    if(LFirst(&(pt->tbl[hv]), &cSlot)){
        if(cSlot.key == k){
            LRemove(&(pt->tbl[hv]));
            return cSlot.val;
        } else {
            while(LNext(&(pt->tbl[hv]), &cSlot)){
                if(cSlot.key == k){
                    LRemove(&(pt->tbl[hv]));
                    return cSlot.val;
                }
            }
        }
    }
    return NULL;
}
```

확장: 탐색

```
Value TBLSearch(Table * pt, Key k)
{
     int hv = pt->hf(k);
     Slot cSlot;

     if(LFirst(&(pt->tbl[hv]), &cSlot)){
          if(cSlot.key == k){
          return cSlot.val;
          } else {
               while(LNext(&(pt->tbl[hv]), &cSlot)){
                    if(cSlot.key == k)
                    return cSlot.val;
               }
          }
     }

     return NULL;
}
```

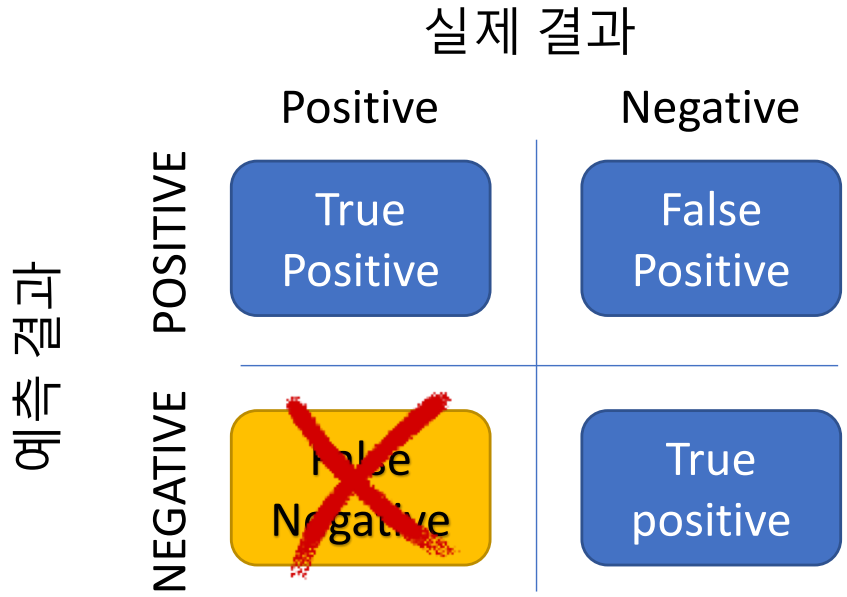
응용: Bloom Filter

블룸 필터란?

- 집합을 표현한 자료구조
  - 집합에서 찾는 값의 존재 여부를 알려줌
- 확률적 자료구조

(False Negative)

- 집합에 속한 원소를 속하지 않았다고 하는 일은 절대 없음
  - 단, 속하지 않은 원소를 가끔 속했다고 함 (False Positive)



핵심 ADT

- `AddKey`: 집합에 키/원소 추가
- `IsMember`: 집합에 키가 존재하는 지 검사(with False Positives)
- `DeleteKey`: Not supported

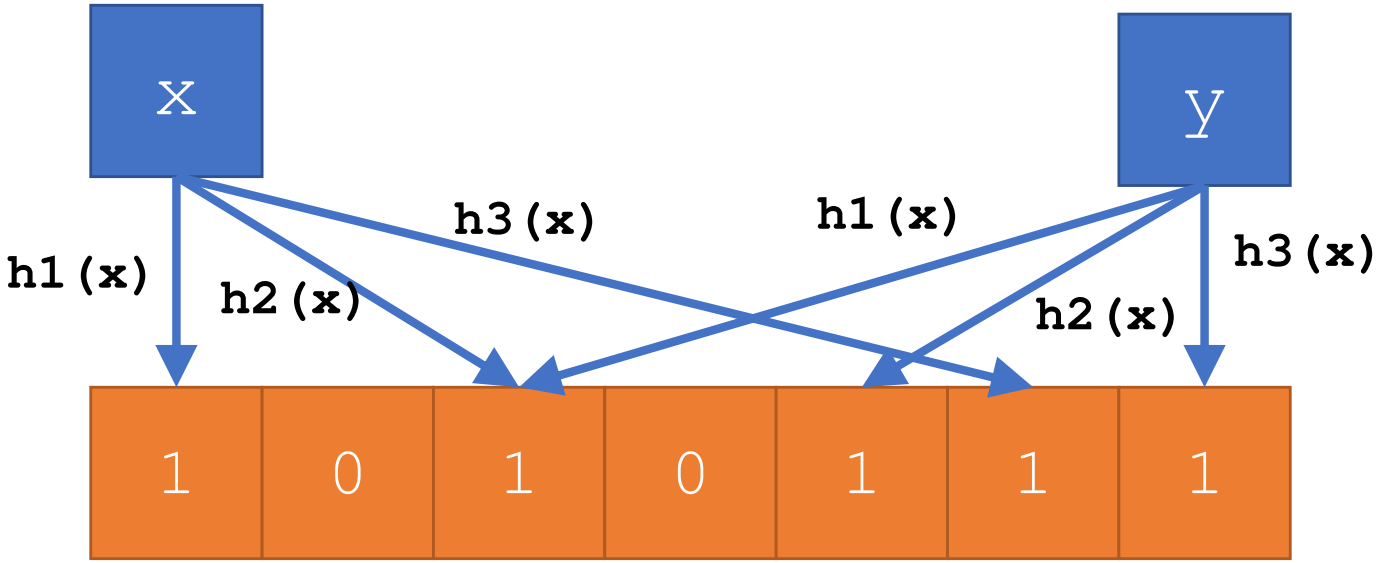
블룸필터의 활용처

- 구글 크롬: 악성 URL 목록
- Medium 뉴스 사이트: 사용자가 이미 읽은 기사 관리
- 그 외에도 응용처가 많음

블룸 필터에 키 삽입

- 가정
  - 집합은 최초에 비었음 (0으로 초기화)
  - 해쉬함수는 모두 독립
  - 집합의 크기: m
  - 해쉬함수의 갯수: k

AddKey(X) // h1(x), h2(x), …, hx(x) 결과의 위치를 set



False Positive Rate

- N 번의 AddKey 연산 후, 특정 비트가 set인 확률

$$P(Bit = 1) = 1 - \left(1 - \frac{1}{m}\right)^{kn}$$

- False positive이려면, k개의 해쉬의 결과가 이미 set이어야 함

$$P(False\ \ Positive) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{\frac{-kn}{m}}\right)^{k}$$

- False positive가 최소가 되려면

$$k = \left(\frac{m}{n}\right) \times \ln 2$$

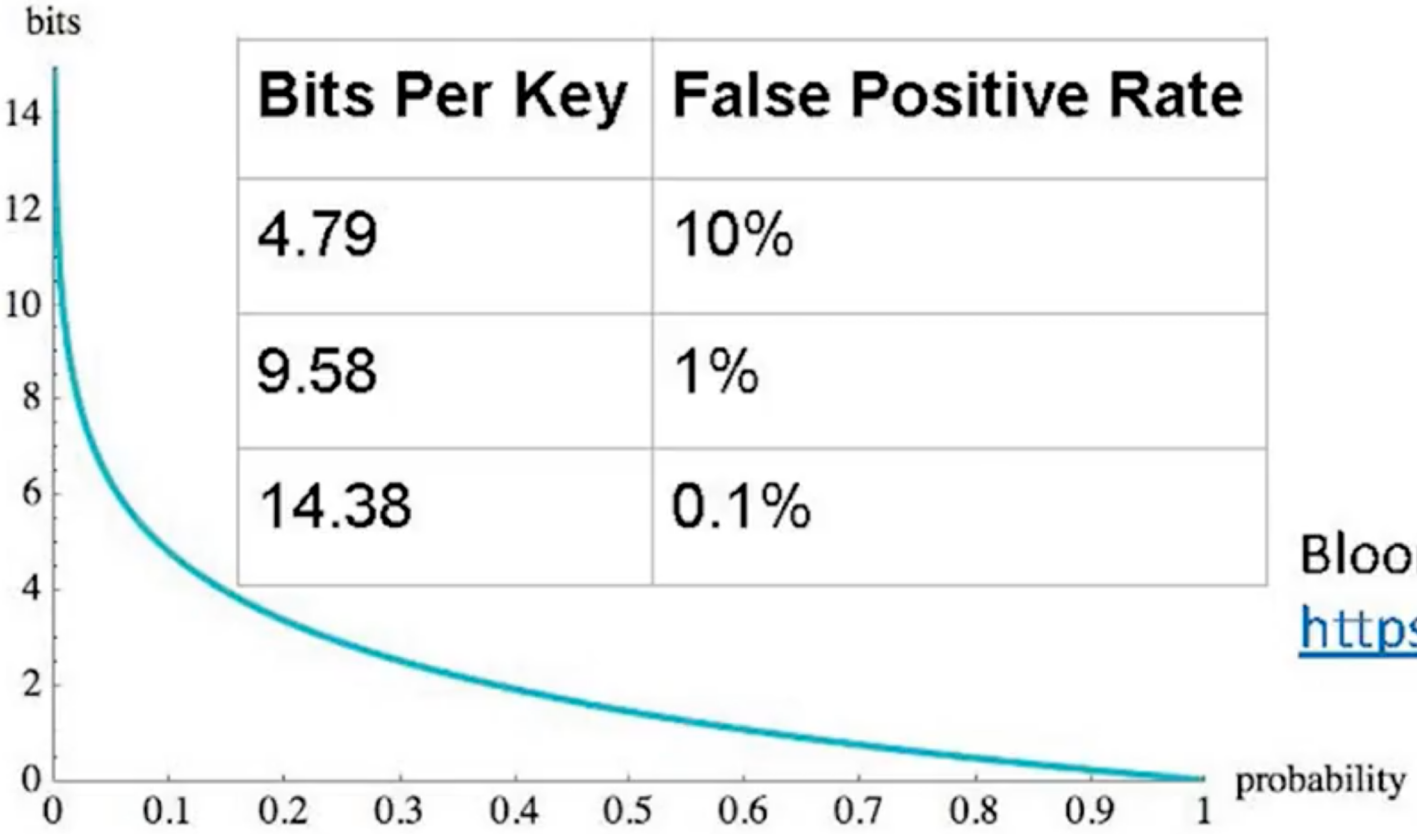
$$no\ \ of\ \ hashses = bits\ \ per\ \ key \times \ln 2$$

- 최소 false positive rate

$$P(FalsePositive) \approx 0.619^{\frac{m}{n}} = 0.619^{bits\ \ per\ \ key}$$

블룸필터의 크기 Vs False Positive 확률

- 블룸필터의 크기가 커질 수록 오차 확률은 매우 적어짐



Bits Per Key	False Positive Rate
4.79	10%
9.58	1%
14.38	0.1%

Bloom Filter Calculator:
https://hur.st/bloomfilter

Image Source: http://corte.si

해쉬 함수 개수를 줄이면서 성능 유지하기

- 두 개의 해쉬 함수로 K개의 해쉬 함수 대신하기
  - 해쉬 함수 g1(x)와 g2(x)로 전혀 다른 해쉬 함수 h(x)만들기

$$h_i(x) = g_1(x) + i \cdot g_2(x)$$

  - 점근적 오차 확률에 영향없음
  - 그러나 계산 비용은 매우 줄어 듬

해쉬 함수의 선택

- 결과는 균등하게 분포되어야 함
- 해쉬의 함수의 결과 크기는 전체 비트의 길이보다 커야 함
- 연산 복잡도는 가능한 적어야 함
- 암호용 해쉬 함수를 쓰는 것은 과함
- 사용 예, murmur3, cityhash, fnv 등

Addkey 코드

```
#define FILTER_SIZE 20
#define NUM_HASHES 7
#define FILTER_BITMASK ((1 << FILTER_SIZE) - 1)
unsigned char filter[FILTER_SIZE_BYTES];

void AddKey(unsigned char filter[], char *str)
{
      unsigned int hash[NUM_HASHES];
      int i;
      get_hashes(hash, str); // 주어진 str의 해쉬결과를 배열 hash에 저장
      for (i = 0; i < NUM_HASHES; i++) {
            /* 해쉬 결과를 필터의 크기만큼 변환(xor활용) */
            hash[i] = (hash[i] >> FILTER_SIZE) ^
                      (hash[i] & FILTER_BITMASK);
            /* 필터에 해쉬 결과 저장 */
            filter[hash[i] >> 3] |= 1 << (hash[i] & 7);
      }
}
```

블룸필터의 문제

- 한 번 삽입된 키는 필터에서 삭제 되지 않음 (DeleteKey 없음)
- 더 이상 존재하지 않는 키로 인해 false positive rate 증가함
- 이 문제를 해결하는 것은 응용프로그램이 담당해야 함
  - 방법 1: 오차를 허용할 수 있는 한 아무 것도 하지 않음
  - 방법 2: 현재 사용 중인 키로 새로운 필터를 생성함
  - 방법 3: 접근된 시간을 정보를 캐쉬로 두어 1차, 2차 필터를 생성함