**09. 해시 테이블(Hash Table)**

**09-1. 해시 테이블**

**# 검색트리와 해시 테이블\_시간 복잡도**

- 배열 또는 연결 리스트 : **O(n), 평균 Θ(n)**

- 이진 검색 트리 : 검색, 삽입, 삭제 시 **평균 Θ(log n), 최악의 경우 Θ(n)**

- 균형 이진 검색 트리

\* 검색, 삽입, 삭제 시 **최악의 경우 Θ(log n)**

\* AVL트리, RB트리

- 균형 다진 검색 트리

\* 검색, 삽입, 삭제 시 **최악의 경우 Θ(log n)**

\* 2-3 트리, 2-3-4 트리, B-트리

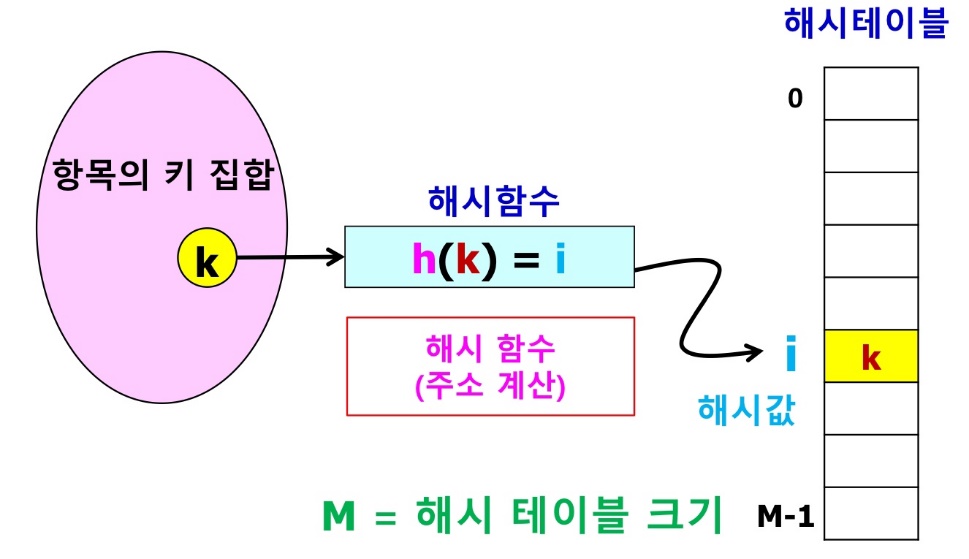
**- 해시 테이블 : 검색, 삽입, 삭제 시 평균 Θ(1)**

**# 해시 테이블(Hash Table)**

: 키를 변환하여 배열의 인덱스로 사용한다.

* **Θ(1) : 아주 빠른 검색, 삽입, 삭제 작업을 제공한다.**
* 자신의 킷값에 의해 위치가 결정된다.
* 다른 킷 값과의 상대적인 크기에 의해 위치가 결정되지 않는다.
* **키를 배열의 인덱스로 그대로 사용하면 메모리 낭비가 심해질 수 있다.**

**\* 주소계산**

****

**# 해싱(Hashing)**

: 키를 간단한 함수를 사용해 변환한 값을 배열의 인덱스로 이용하여 항목을 저장하는 것

* 해시 함수(Hash Function) : 해싱에 사용되는 함수
* 해시 값(Hash Value) 또는 해시 주소 : 해시함수가 계산한 값
* 해시 테이블(Hash Table) : 항목이 해시 값에 따라 저장되는 배열

**# 충돌 테이블\_해시 함수(Hash Functions)**

**\* 가장 이상적인 해시 함수**

**: 키들을 균등하게(Uniformly) 해시 테이블의 인덱스로 변환하는 함수**

- 균등하게 변환한다는 것은 키들을 해시 테이블에 랜덤하게 흩어지도록 저장하는 것을 뜻한다.

- 해시 함수는 키들을 균등하게 해시 테이블의 인덱스로 변환하기 위해 의미가 부여되어 있는 키를 간단한 계산을 통해 ‘뒤죽박죽’ 만든 후 해시 테이블의 크기에 맞도록 해서 해시 값을 계산한다.

- 아무리 균등한 결과를 보장하는 해시 함수라도 함수 계산 자체에 긴 시간이 소요된다면 해싱의 장점인 연산의 신속성을 상실하므로 그 가치를 잃어버린다.

🡪 일반적으로 키들은 부여된 의미나 특성을 가지므로 키의 가장 앞 부분 또는 뒤의 몇 자리 등을 취하여 해시 값으로 사용하는 방식의 해시함수는 많은 충돌을 야기시킨다.

**\* 장난감 수준의 함수**

- 자릿수를 선택 : h(001362825)=35

- 접기(folding) : h(001364825)=1190 🡪 해시테이블의 크기가 3이라면 1190에서 3자리 수만을 해시 값으로 이용한다.

**\* 나누기 방법(Division Method)에 의한 함수**

- h(key)=key%m 🡨 m : 해시 테이블 사이즈(m은 소수를 권장), % : 나머지 연산

**\* 곱하기 방법(Multiplication Method)에 의한 함수**

- h(key) = (keyA mod 1)\* m

- A : (0, 1) 범위의 상수

- m이 굳이 소수일 필요 없음. 보통 2^p(p는 양의 정수)

텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

자동 생성된 설명

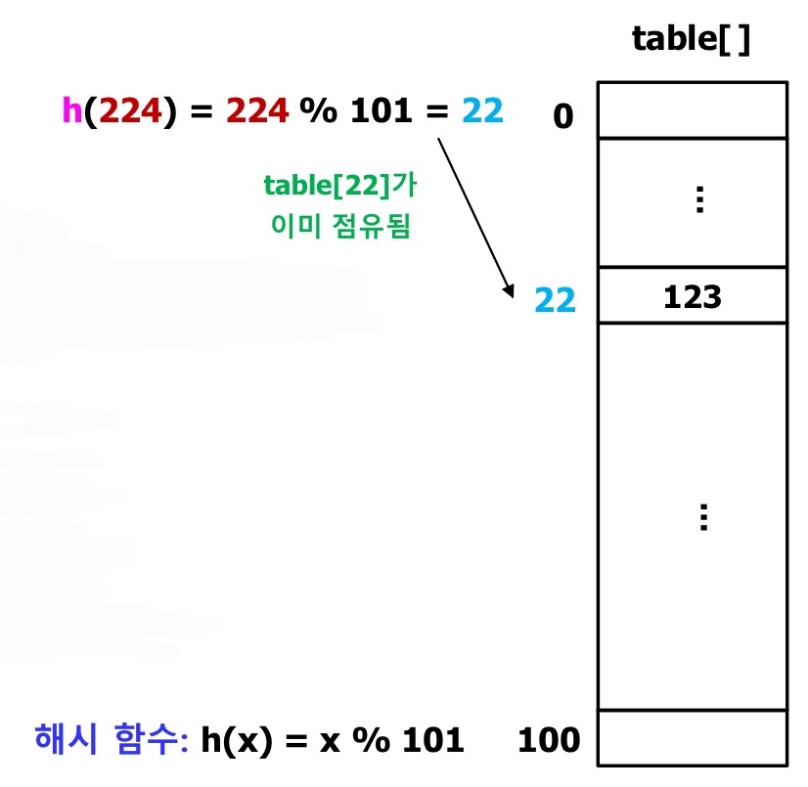
**09-2. 충돌 해결(Collision Resolution)**

**충돌(Collision)**

: 어떤 킷값으로 도출된 주소에 이미 다른 키가 자리함

**충돌 해결**

* 일련의 해시 함수를 생성한다.
* 해시 테이블에서 핵심적인 부분



**# 개방형 주소 방식(Open Addressing Methods)**

: 해시테이블 전체를 열린 공간으로 가정하고 충돌된 키를 일정한 방식에 따라서 찾아낸 empty 원소에 저장한다.

ex) 선형 조사(Linear Probing), 이차원 조사(Quadratic Probing), 무작위 조사(Random Probing), 이중 해싱(Double Hashing)

**\* 개방 주소 방법(Open Addressing)**

: 주어진 배열 안에서 해결

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

**# 선형 조사(Linear Probing)**

: 선형 조사는 충돌이 일어난 원소에서부터 순차적으로 검색하여, 처음 발견한 empty 원소에 충돌이 일어난 키를 저장한다.

1. **h(key) = i** : 해시테이블 **a[i], a[i+1], a[i+2], …, a[i+j]**를 차례로 검색하여 처음으로 찾아낸 **empty** 원소에 **key**를 저장한다.

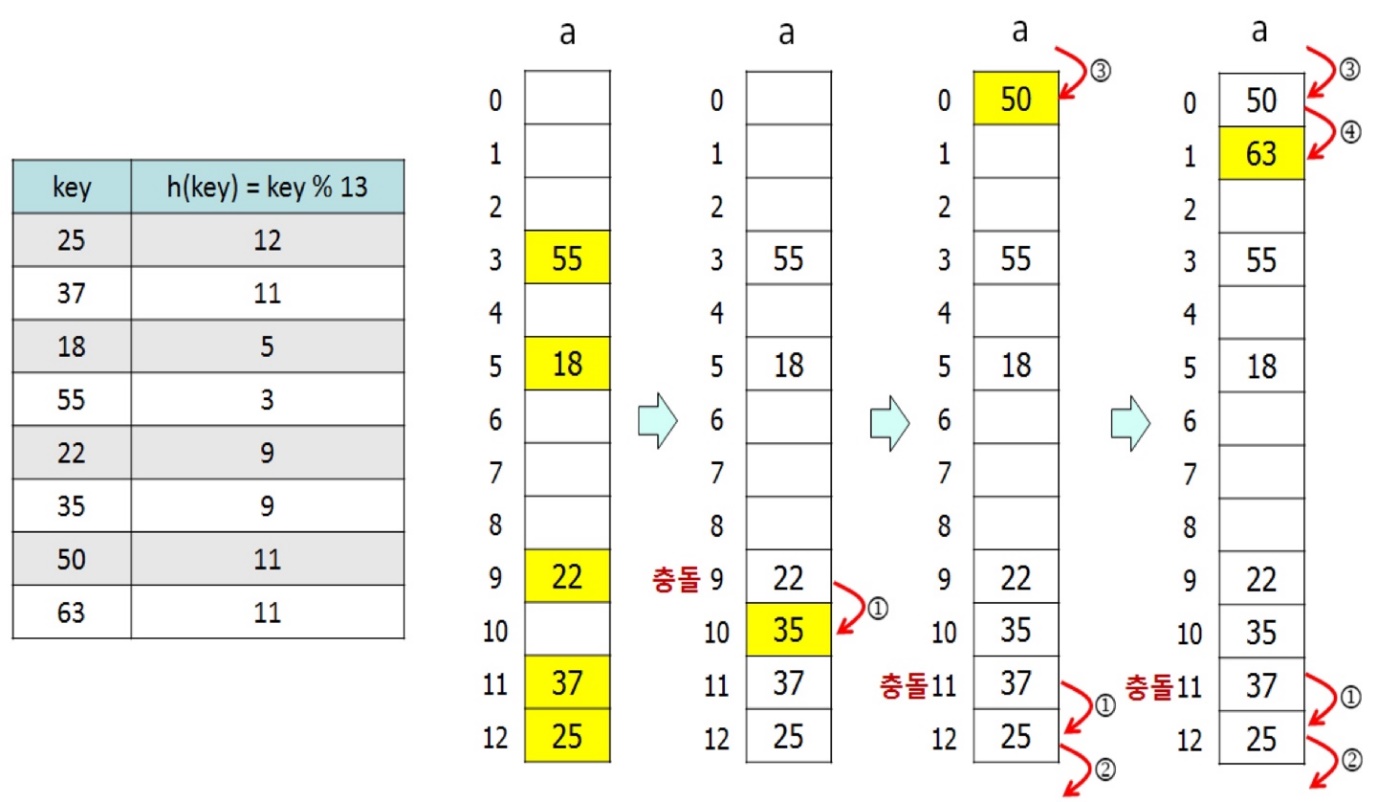
2. 해시테이블은 1차원 리스트이므로, **i+j**가 **m**이 되면 **a[0]**를 검색한다.

3. **(h(key)+j)%m, j=0, 1, 2, 3, …**

텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

자동 생성된 설명

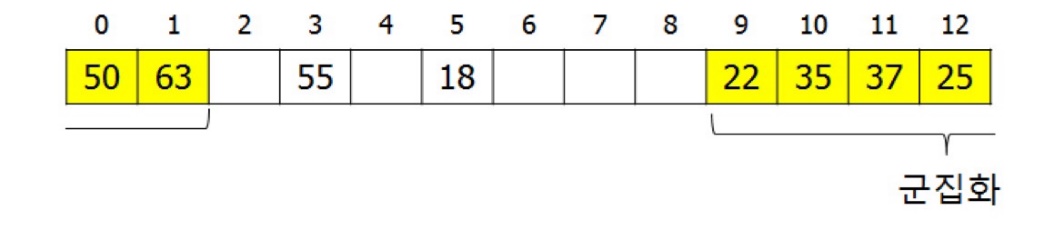
**\* 선형 조사 방식의 키 저장 과정**

****

**\* 1차 군집화(Primary Clustering)**

: 선형조사는 순차 검색으로 empty 원소를 찾아 충돌된 키를 저장하므로 해시 테이블의 키들이 빈틈없이 뭉쳐지는 현상이 발생한다.

* 이러한 군집화는 검색, 삽입, 삭제 연산 시 군집된 키들을 순차적으로 방문해야 하는 문제점을 야기한다.
* 군집화는 해시 테이블에 empty 원소 수가 적을수록 더 심화되면 해시 성능을 극단적으로 저하시킨다.

****

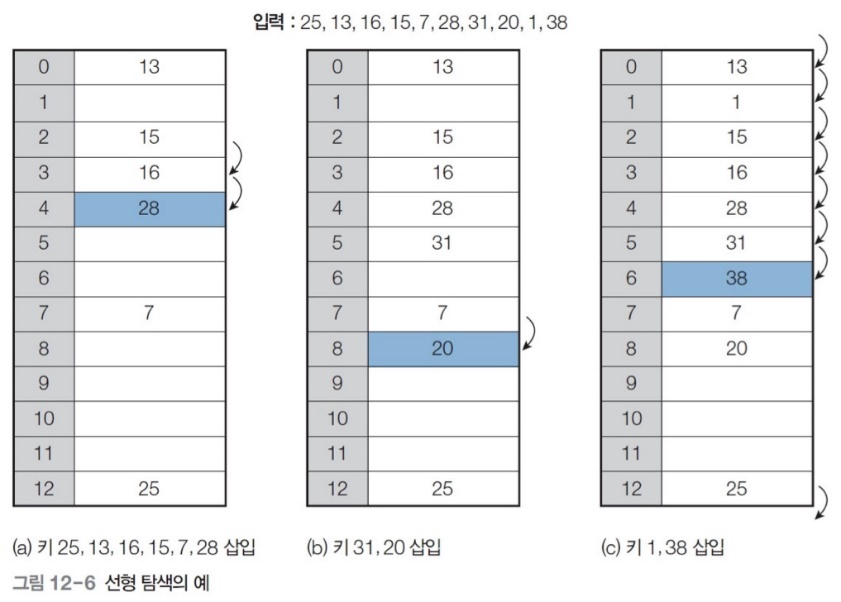
**\* 선형조사 : 알고리즘(검색, 삽입, 삭제)**

**텍스트, 스크린샷, 소프트웨어, 번호이(가) 표시된 사진

자동 생성된 설명**

**1. 삽입**

**ex1) 삽입의 예**

****

**ex2) 1차 군집화**

**텍스트, 스크린샷, 번호이(가) 표시된 사진

자동 생성된 설명**

**ex3) 1차 군집화와 2차 군집화**

**텍스트, 스크린샷, 도표, 평행이(가) 표시된 사진

자동 생성된 설명**

**2. 삭제(삭제 시 주의할 점)**

삭제된 자리를 null값으로 채우면, 검색 시 존재하는 원소를 없다고 대답할 수 있다.

* **지운 자리에 표식을 해둔다. ex) 상수 DELETED 할당**

**텍스트, 스크린샷, 번호, 평행이(가) 표시된 사진

자동 생성된 설명**

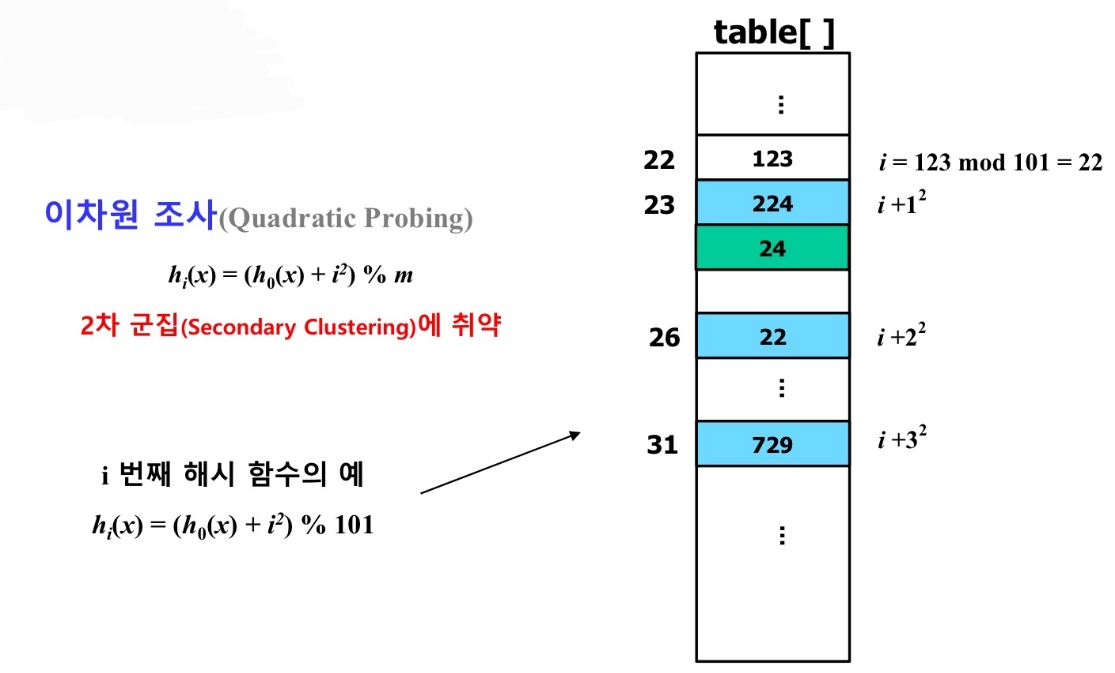
**# 이차조사(Quadratic probing)**

: 선형조사와 근본적으로 동일한 충돌해결 방법

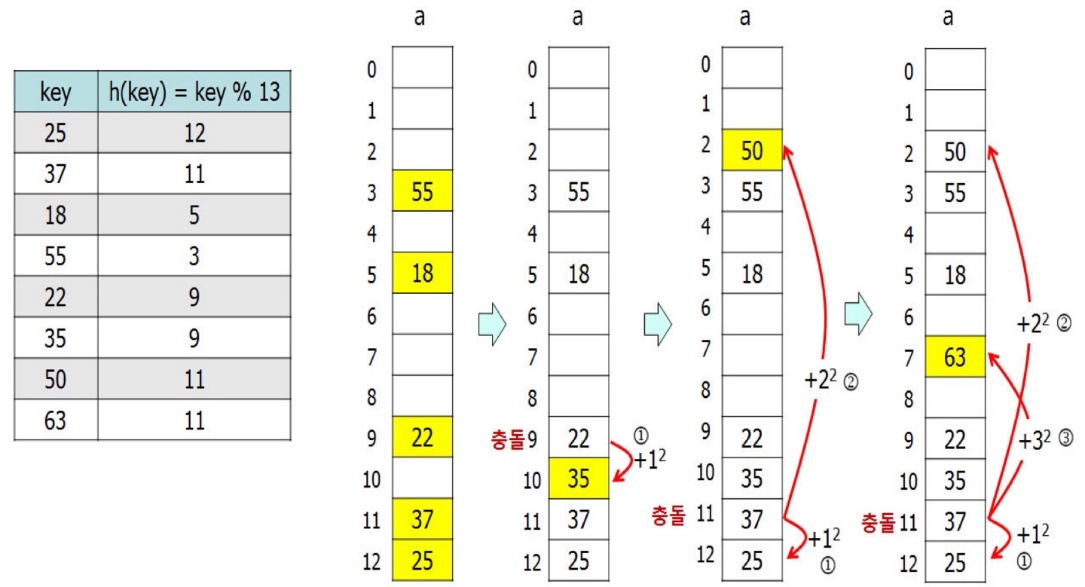
- 충돌 후 배열 a에서 **(h(key)+j^2)%M = 0, 1, 2, 3…** 으로 선형조사보다 더 멀리 떨어진 곳에서 empty 원소를 찾는다.

- 이차조사는 이웃하는 빈 곳이 채워져 만들어지는 1차 군집화 문제를 해결하지만, 같은 해시 값을 갖는 서로 다른 키들인 동의어(Synonym)들이 똑 같은 점프 시퀀스(jump Sequence)를 따라 empty 원소를 찾아 저장하므로 결국 또 다른 형태의 군집화 인 **2차 군집화(Secondary Clustering)**을 야기한다.

- 점프 크기가 제곱만큼 씩 커지므로 배열에 empty원소가 있는데도 empty 원소를 건너뛰어 검색에 실패하는 경우도 피할 수 없다.

****

**\* 이차 조사 방식의 키 저장 과정**

****

**\* 알고리즘 구현(Linear probing, Quadratic probing)**

#include <iostream>

#include <vector>

using namespace std;

// HashOpenAddressed class

template <typename E>

class HashOpenAddressing {

public:

HashOpenAddressing(int num);

int insert(int num);

int remove(int num);

int search(int num);

void output(void);

private:

vector<E> \_\_table;

int \_\_count;

const int DELETED = -54321;

int \_\_Hash(int i, int num) { // 해시 함수

return (num + i) % \_\_table.size(); // 선형 조사

//return (num + i\*i) % \_\_table.size(); // 이차 조사

}

};

template <typename E>

HashOpenAddressing<E>::HashOpenAddressing(int num) : \_\_count(0) {

\_\_table = vector<int>(num, NULL);

}

// 데이터 삽입

template <typename E>

int HashOpenAddressing<E>::insert(int num) {

if (\_\_count < \_\_table.size()) {

for (int i = 0; i < \_\_table.size(); i++) {

int slot = \_\_Hash(i, num);

if (\_\_table[slot] == NULL || \_\_table[slot] == DELETED) {

\_\_table[slot] = num;

\_\_count++;

return num;

}

}

}

return NULL;

}

// 데이터 삭제

template <typename E>

int HashOpenAddressing<E>::remove(int num) {

for (int i = 0; i < \_\_table.size(); i++) {

int slot = \_\_Hash(i, num);

if (\_\_table[slot] == NULL)

return NULL;

else if (\_\_table[slot] == num) {

\_\_table[slot] = DELETED;

\_\_count--;

return num;

}

}

return NULL;

}

// 데이터 검색

template <typename E>

int HashOpenAddressing<E>::search(int num) {

for (int i = 0; i < \_\_table.size(); i++) {

int slot = \_\_Hash(i, num);

if (\_\_table[slot] == NULL) return NULL;

else if (\_\_table[slot] == num) return \_\_table[slot];

}

return NULL;

}

// 전체 원소 출력

template <typename E>

void HashOpenAddressing<E>::output(void) {

cout << "count(" << \_\_count << "): ";

for (int i = 0; i < \_\_table.size(); i++) {

if (\_\_table[i] == DELETED) cout << ' ' << "DELETED";

else cout << ' ' << \_\_table[i];

}

cout << endl;

}

int main(void) {

HashOpenAddressing<int> h(13);

while (true) {

cout << "\n ### 해시 테이블 ###" << endl;

cout << "1) 데이터 삽입" << endl;

cout << "2) 데이터 삭제" << endl;

cout << "3) 데이터 검색" << endl;

cout << "4) 전체 출력" << endl;

cout << "5) 프로그램 종료" << endl;

cout << "메뉴 선택: ";

int choice;

cin >> choice;

int num;

if (choice == 1) {

cout << "삽입 할 데이터 입력: ";

cin >> num;

int inserted = h.insert(num);

if (inserted != NULL)

cout << "삽입 된 데이터: " << inserted << std::endl;

else

cout << "데이터 삽입에 실패하였습니다." << std::endl;

}

else if (choice == 2) {

cout << "삭제 할 데이터 입력: ";

cin >> num;

int removed = h.remove(num);

if (removed != NULL)

cout << "삭제 된 데이터: " << removed << endl;

else

cout << "데이터가 존재하지 않습니다." << endl;

}

else if (choice == 3) {

cout << "검색 할 데이터 입력: ";

cin >> num;

int searched = h.search(num);

if (searched != NULL)

cout << "검색된 데이터: " << searched << endl;

else

cout << "데이터가 존재하지 않습니다." << endl;

}

else if (choice == 4) h.output();

else if (choice == 5) {

cout << "프로그램 종료!!!" << endl;

break;

}

else cout << "잘못 선택 하셨습니다." << endl;

}

return 0;

}

**# 무작위 조사(Random probing)**

: 선형조사와 이차 조사의 규칙적인 점프 시퀀스와는 달리 점프 시퀀스를 무작위화하여 empty원소를 찾는다.

- 무작위 조사는 **의사 난수 생성기**를 사용하여 다음 위치를 찾는다.

- 무작위 조사 방식도 동의어들이 똑 같은 점프 시퀀스에 따라 empty 원소를 찾아 키를 저장하게 되고, 이 때문에 **3차 군집화(Tertiary Clustering)**가 발생한다.

**# 이중해싱(Double Hashing)**

: 2개의 해시 함수를 사용한다.

- 하나는 기본적인 해시 함수 **h(key)**로 키를 해시 테이블의 인덱스로 변환하고, 제 2의 함수 **d(key)**는 충돌 발생 시 다음 위치를 위한 점프 크기를 다음의 규칙에 따라 정한다.

**- (h(key)+j\*d(key)) mod M, j = 0, 1, 2, …**

- 제 2의 함수 **d(key)**는 점프 크기를 정하는 함수이므로, 0을 반환해서는 안된다.

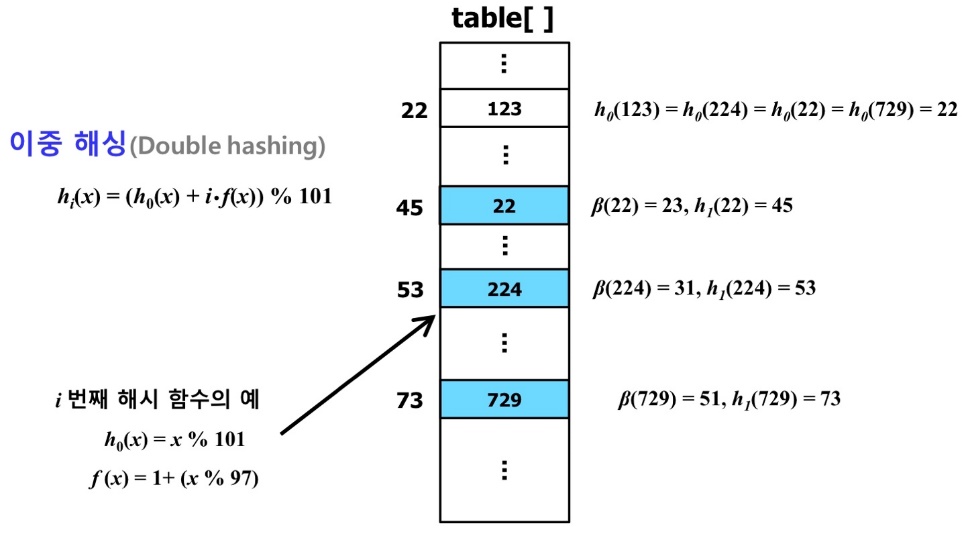
- 그 외의 조건으로 **d(key)**의 값과 해시 테이블의 크기 **M**과 서로소(Relatively Prime) 관계일 때 좋은 성능을 보인다.

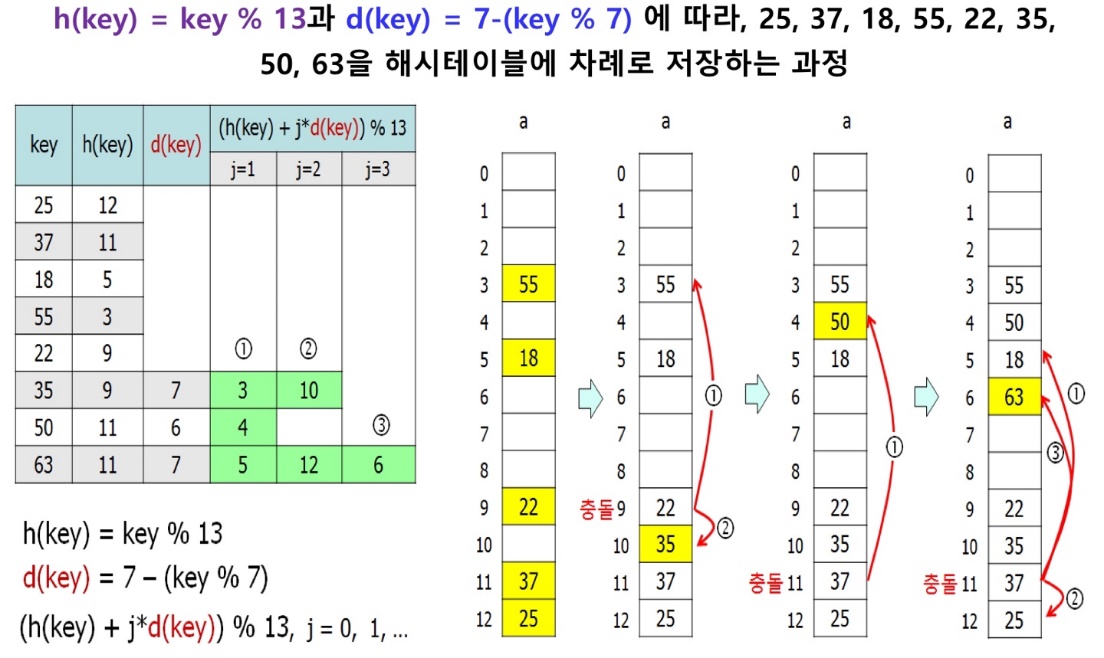
- 하지만 해시 테이블 크기 M을 소수로 선택하면, 이 제약 조건을 만족한다.

**\* 이중해싱의 장점**

1. 빈 곳을 찾기 위한 점프 시퀀스가 일정하지 않으며(저마다 제2의 해시 함수를 갖기 때문에), 모든 군집화 현상을 발생시키지 않는다.**(모든 군집화 문제를 해결)**

2. 또한 해시 성능을 저하시키지 않는 동시에 해시테이블에 많은 키들을 저장할 수 있다는 장점을 가지고 있다.





**\* 알고리즘 구현(Double hashing)**

#include <iostream>

#include <vector>

using namespace std;

// HashOpenAddressed class

template <typename E>

class HashOpenAddressing {

public:

HashOpenAddressing(int num);

int insert(int num);

int remove(int num);

int search(int num);

void output(void);

private:

vector<E> \_\_table;

int \_\_count;

const int DELETED = -54321;

int \_\_Hash(int i, int num); // 첫 번째 해시 함수

int \_\_Hash2(int num); // 두 번째 해시 함수 (이중 해싱에 사용)

};

template <typename E>

HashOpenAddressing<E>::HashOpenAddressing(int num) : \_\_count(0) {

\_\_table = vector<int>(num, NULL);

}

// 데이터 삽입

template <typename E>

int HashOpenAddressing<E>::insert(int num) {

if (\_\_count < \_\_table.size()) {

for (int i = 0; i < \_\_table.size(); i++) {

int slot = (\_\_Hash(i, num) + i \* \_\_Hash2(num)) % \_\_table.size();

if (\_\_table[slot] == NULL || \_\_table[slot] == DELETED) {

\_\_table[slot] = num;

\_\_count++;

return num;

}

}

}

return 0;

}

// 데이터 삭제

template <typename E>

int HashOpenAddressing<E>::remove(int num) {

for (int i = 0; i < \_\_table.size(); i++) {

int slot = (\_\_Hash(i, num) + i \* \_\_Hash2(num)) % \_\_table.size();

if (\_\_table[slot] == NULL)

return 0;

else if (\_\_table[slot] == num) {

\_\_table[slot] = DELETED;

\_\_count--;

return num;

}

}

return 0;

}

// 데이터 검색

template <typename E>

int HashOpenAddressing<E>::search(int num) {

for (int i = 0; i < \_\_table.size(); i++) {

int slot = (\_\_Hash(i, num) + i \* \_\_Hash2(num)) % \_\_table.size();

if (\_\_table[slot] == NULL) return 0;

else if (\_\_table[slot] == num) return \_\_table[slot];

}

return 0;

}

// 전체 원소 출력

template <typename E>

void HashOpenAddressing<E>::output(void) {

cout << "count(" << \_\_count << "): ";

for (int i = 0; i < \_\_table.size(); i++) {

if (\_\_table[i] == DELETED) cout << ' ' << "DELETED";

else cout << ' ' << \_\_table[i];

}

cout << endl;

}

// 첫 번째 해시 함수 (선형 조사)

template <typename E>

int HashOpenAddressing<E>::\_\_Hash(int i, int num) {

return (num + i) % \_\_table.size();

}

// 두 번째 해시 함수 (이중 해싱)

template <typename E>

int HashOpenAddressing<E>::\_\_Hash2(int num) {

return 7 - (num % 7); // 예시로 7로 나눈 나머지를 사용

}

int main(void) {

HashOpenAddressing<int> h(4);

while (true) {

cout << "\n ### 해시 테이블 ###" << endl;

cout << "1) 데이터 삽입" << endl;

cout << "2) 데이터 삭제" << endl;

cout << "3) 데이터 검색" << endl;

cout << "4) 전체 출력" << endl;

cout << "5) 프로그램 종료" << endl;

cout << "메뉴 선택: ";

int choice;

cin >> choice;

int num;

if (choice == 1) {

cout << "삽입 할 데이터 입력: ";

cin >> num;

int inserted = h.insert(num);

if (inserted != 0)

cout << "삽입 된 데이터: " << inserted << std::endl;

else

cout << "데이터 삽입에 실패하였습니다." << std::endl;

}

else if (choice == 2) {

cout << "삭제 할 데이터 입력: ";

cin >> num;

int removed = h.remove(num);

if (removed != 0)

cout << "삭제 된 데이터: " << removed << endl;

else

cout << "데이터가 존재하지 않습니다." << endl;

}

else if (choice == 3) {

cout << "검색 할 데이터 입력: ";

cin >> num;

int searched = h.search(num);

if (searched != 0)

cout << "검색된 데이터: " << searched << endl;

else

cout << "데이터가 존재하지 않습니다." << endl;

}

else if (choice == 4) h.output();

else if (choice == 5) {

cout << "프로그램 종료!!!" << endl;

break;

}

else cout << "잘못 선택 하셨습니다." << endl;

}

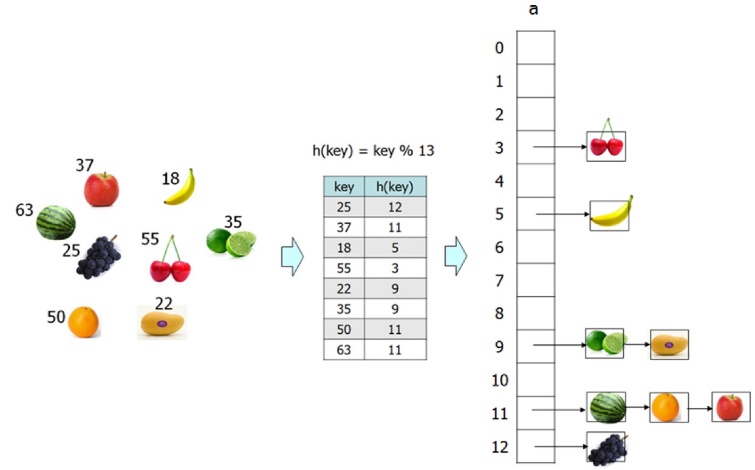
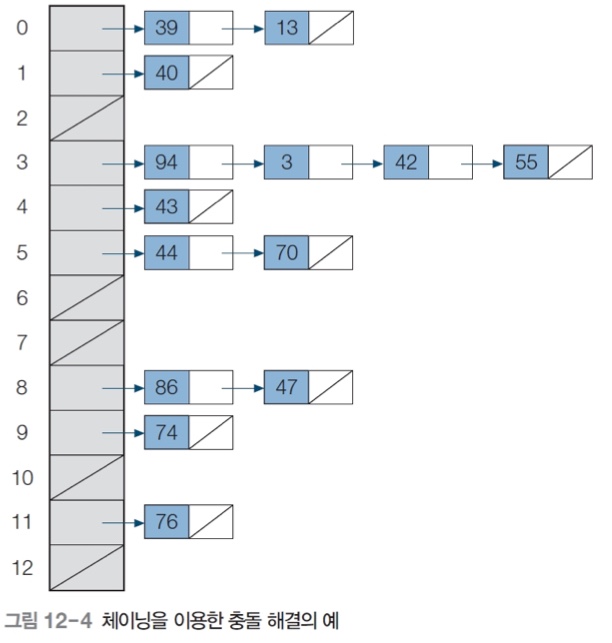
return 0;

}

**# 폐쇄 주소 방식(Closed Addressing Methods)**

: 키에 대한 해시 값에 대응되는 곳에만 키를 저장한다.(충돌이 발생한 키들은 한 위치에 모여 저장한다.)

* **가장 대표적인 방법 : 체이닝(Chaining)**
* 해시 테이블은 연결 리스트의 헤더 노드를 참조한다.
* 직접 충돌이 일어나지 않은 키들끼리는 간섭하지 않는다.

****

**\* 알고리즘 구현(Chaining\_ver.Seo)**

#include <iostream>

#include <vector>

#include <list>

using namespace std;

// HashChaining class

template <typename E>

class HashChaining {

public:

HashChaining(int num);

int insert(int num);

int remove(int num);

int search(int num);

void output(void);

private:

vector<list<E>> \_\_table; // 각 슬롯에 대한 연결 리스트

int \_\_count;

const int DELETED = -54321;

int \_\_Hash(int num); // 해시 함수

};

template <typename E>

HashChaining<E>::HashChaining(int num) : \_\_count(0) {

\_\_table = vector<list<E>>(num);

}

// 데이터 삽입

template <typename E>

int HashChaining<E>::insert(int num) {

int slot = \_\_Hash(num);

\_\_table[slot].push\_back(num);

\_\_count++;

return num;

}

// 데이터 삭제

template <typename E>

int HashChaining<E>::remove(int num) {

int slot = \_\_Hash(num);

auto& chain = \_\_table[slot];

for (auto it = chain.begin(); it != chain.end(); ++it) {

if (\*it == num) {

chain.erase(it);

\_\_count--;

return num;

}

}

return 0;

}

// 데이터 검색

template <typename E>

int HashChaining<E>::search(int num) {

int slot = \_\_Hash(num);

for (auto& item : \_\_table[slot]) {

if (item == num) {

return item;

}

}

return 0;

}

// 전체 원소 출력

template <typename E>

void HashChaining<E>::output(void) {

cout << "count(" << \_\_count << "): " << endl;

for (int i = 0; i < \_\_table.size(); i++) {

cout << i << ": ";

if (\_\_table[i].empty()) {

cout << "EMPTY";

}

else {

for (auto& item : \_\_table[i]) {

cout << item << " ";

}

}

cout << endl;

}

}

// 해시 함수(선형 조사)

template <typename E>

int HashChaining<E>::\_\_Hash(int num) {

return num % \_\_table.size();

}

int main(void) {

HashChaining<int> h(13);

while (true) {

cout << "\n ### 해시 테이블 ###" << endl;

cout << "1) 데이터 삽입" << endl;

cout << "2) 데이터 삭제" << endl;

cout << "3) 데이터 검색" << endl;

cout << "4) 전체 출력" << endl;

cout << "5) 프로그램 종료" << endl;

cout << "메뉴 선택: ";

int choice;

cin >> choice;

int num;

if (choice == 1) {

cout << "삽입 할 데이터 입력: ";

cin >> num;

int inserted = h.insert(num);

cout << "삽입 된 데이터: " << inserted << endl;

}

else if (choice == 2) {

cout << "삭제 할 데이터 입력: ";

cin >> num;

int removed = h.remove(num);

cout << "삭제 된 데이터: " << removed << endl;

}

else if (choice == 3) {

cout << "검색 할 데이터 입력: ";

cin >> num;

int searched = h.search(num);

if (searched != 0)

cout << "검색된 데이터: " << searched << endl;

else

cout << "데이터가 존재하지 않습니다." << endl;

}

else if (choice == 4) h.output();

else if (choice == 5) {

cout << "프로그램 종료!!!" << endl;

break;

}

else {

cout << "잘못 선택 하셨습니다." << endl;

break;

}

}

return 0;

}

**\* 알고리즘 구현(Chaining\_ver.Seo)**