안녕하세요, 동계 대학생 S/W 알고리즘 특강의 여덟번째 시간인 오늘은 Hash 에 대해 다루어보도록 하겠습니다.

1. 기초 강의

동영상 강의 컨텐츠 확인 > 8. Hash

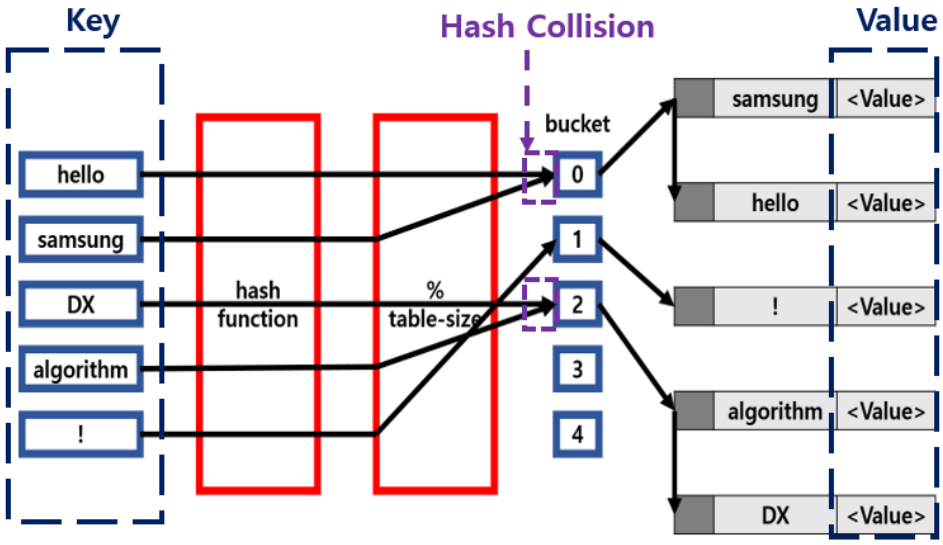
Link :
https://swexpertacademy.com/main/learn/course/subjectDetail.do?courseId=CONTENTS_REVIEW&subjectId=AYVXrOa6QykDFARs

※ 출석은 강의 수강 내역으로 확인합니다.

2. 실전 강의

2.0 해시 테이블이란?

해시함수를 사용하여 키를 해시값으로 매핑하고, 이 해시값을 주소또는 색인 삼아 데이터(value)를 key 와 함께 저장하는 자료구조입니다.



Value : 저장하고자 하는 정보. 최종적으로 저장소(bucket, slot)에 hash 와 매칭되어 저장됩니다.

key : 고유한 값. Hash function 의 input 입니다. Key 값 그대로 최종 저장소에 저장되면 ,다양한 길이의 저장소를 미리 구성해 두어야 하기 때문에 hash function 으로 값을 바꿔 저장합니다.

Hash function : key 를 고정된 길이의 hash 로 변경합니다. 이때 서로다른 key 가 같은 hash 가 되는 경우가 있습니다. 이를 Hash Collison 이라 불리며, Hash Collision 발생확률을 최대한 줄이는 Hash function 을 만들어야 합니다. Hash Collision 을 해결하는 방법에는 chaining 기법 과 open address hash 방식이 있습니다. 이 기법들은 뒤에서 후술하겠습니다.

2.1 Hash Table 의 삽입, 삭제, 검색

일반적으로 해시테이블의 삽입, 삭제, 검색의 시간복잡도는 O(1)입니다. Key 는 고유하며, Hash Function 의 결과로 나온 hash 와 value 를 저장소에 삽입/삭제/검색 하면 되기 때문입니다.

하지만, 이는 Hash Collision 을 고려하지 않았을때의 결과입니다. 최악의 경우는 O(n)으로 Hash Collision 이 발생하여 모든 bucket 의 value 를 찾아 봐야하는 경우가 해당합니다.

2.2 hash function example - djb2

djb2 는 문자열의 hash 함수 중 간략하면서도 무작위 분포를 만드는데 뛰어나다고 알려져 있습니다. magic number 5381 과 33 을 활용하여 hash key 를 생성합니다.

djb2 의 코드는 다음과 같습니다.

```
unsigned long djb2(unsigned char* str) {
        unsigned long hash = 5381;
        int c;
        while (c = *str++) {
                hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
        }
        return hash;
}
```

2.3 Hash Collision(충돌)

입력 값이 달라도 해시 값은 같을 수 있기 때문에 충돌의 가능성은 늘 존재합니다. 충돌 문제를 해결하기 위해 다음과 같은 방법을 주로 사용합니다.

◆ Open addressing
충돌이 발생할 경우, 해시 테이블의 다른 자리에 대신 저장하는 방법입니다.
따라서 Open addressing 방식에서는 하나의 해시에 하나의 값이 매칭됩니다.

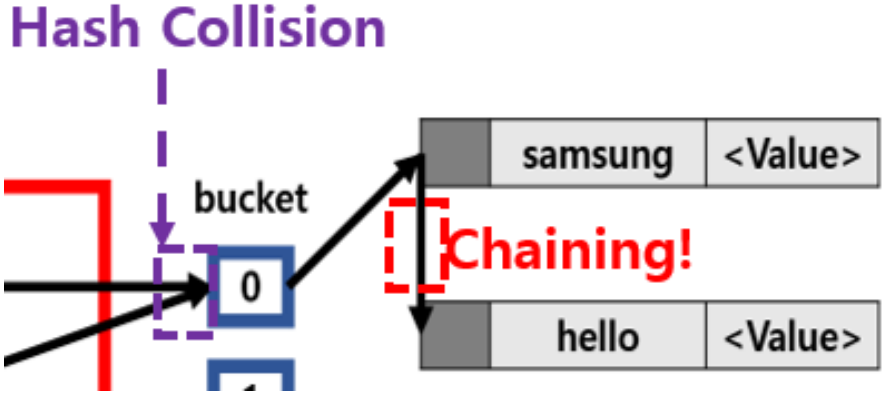
Open Addressing 는 위에서 언급한 비어있는 해시를 찾는 규칙에 따라 다음과 같이 구분할 수 있습니다.

- 선형 탐색(Linear Probing): 다음 해시(+1)나 n 개(+n)를 건너뛰어 비어있는 해시에 데이터를 저장
- 제곱 탐색(Quadratic Probing): 충돌이 일어난 해시의 제곱을 한 해시에 데이터를 저장
- 이중 해시(Double Hashing): 다른 해시함수를 한 번 더 적용한 해시에 데이터를 저장

◆ Chaining

Linked list 를 이용해서 충돌을 해결하는 방법입니다. (주로 사용할 방법)

자료 저장시, 저장소(bucket)에서 충돌이 일어나면 해당 값을 기존 값과 연결시키는 기법입니다.



위 이미지의 경우, hello 를 저장할 때 충돌이 일어나 기존의 samsung 에 연결시켰습니다.

이때 Linked List 자료구조를 이용해 다음에 저장할 자료를 기존 자료 다음에 위치시킵니다.

장점 :

1) 한정된 저장소(Bucket)를 효율적으로 사용할 수 있습니다.
2) 상대적으로 적은 메모리를 사용합니다

단점 :

1) 한 Hash 에 자료들이 계속 연결될 수 있으며, 이때 검색 효율이 떨어집니다.

2.4 적절한 hash table size

보통 소수나 2 의 거듭제곱(2^m)을 사용합니다.

테이블의 크기로 소수를 사용 할 경우, 충돌이 적지만 느린 % 연산자를 사용해야 합니다.

해시 함수가 아주 고르게 분포하는(무작위에 가까운) 값들을 출력해준다면 테이블의 크기가 소수인지는 별로 중요하지 않습니다. 하지만 출력 값이 고르지 않다면 어떨까요?

예를 들어 해시 값이 15, 30, 45, 60, 75, 90, ...처럼 15 의 배수가 굉장히 많이 나오는 상황에서 해시 테이블의 크기가 30 이라면 0 번 버킷과 15 번 버킷에만 많은 데이터가 저장됩니다.

테이블의 크기가 s 이고 해시 함수가 x 의 배수만 출력한다면, 실질적인 테이블의 크기는 s / gcd(s, x)과 같습니다. 따라서 이런 편향된 해시 값이 많은 경우에 효율적인 테이블의 크기는

자신을 제외한 모든 수와 서로소인 소수입니다.
C++의 Unordered Associative Containers 는 소수를 사용합니다.

연관배열 구조(associative array) : 키(key) 1 개와 값(value) 1 개가 1:1 로 연관되어 있는 자료구조이다. 따라서 키(key)를 이용하여 값(value)을 도출할 수 있다.

반면에 2 의 거듭제곱을 활용 할 경우, bit masking 을 통해 나머지를 빠르게 계산을 할 수 있지만 m 개의 하위 bit 를 그대로 사용하기 때문에 hash function 이 하위 bit 을 고르게 분포 시키지 못한다면 많은 충돌이 발생하게 됩니다.

Java 에선 hashMap 의 size 를 2 의 거듭제곱을 사용하고 있습니다.

다만, 소수나 2 의 거듭제곱이나 당락에 영향을 줄 만큼 큰 영향은 없습니다.
편하신 방법을 이용하시면 됩니다.
10007, 20011, 30011, 40009, 100003, 200003 이 흔히 쓰이는 소수로 소수를 사용하실 분들은 참고해 주세요.

2.5 생일 문제 예제

2.2 와 2.3 에서 해시 충돌에 대비하는 방법을 배웠습니다. 하지만 이것보다 훨씬 간단한 해결책이 있어 보입니다. 해시 테이블의 크기를 충분히 크게 만들면 되지 않을까요?
이 방법이 얼마나 비효율적인지 알려주는 "생일 문제"가 있습니다.

몇 명의 사람들이 모여야 그중에서 생일이 같은 쌍이 100% 존재할까요?
윤년을 제외하고 생일의 가짓수를 365 개라 하겠습니다. 비둘기 집의 원리에 의해 366 명의 사람이 모이면 생일이 같은 쌍이 무조건 생깁니다. 생일이 같은 두 사람이 100% 생기기 위해서는 366 명의 사람이 필요합니다.

그렇다면 생일이 같은 쌍이 존재할 확률이 50%가 되기 위해서는 몇 명의 사람이 필요할까요?
답은 훨씬 적은 23 명입니다. 23 명의 생일이 모두 겹치지 않을 확률은 50%가 되지 않습니다.

$$\frac{365}{365}\times\frac{364}{365}\times\frac{363}{365}\times\cdots\times\frac{343}{365}\approx 0.49$$

생일의 가짓수를 해시 테이블 크기, 사람 수를 해시 테이블에 삽입되는 데이터의 개수라고 생각해봅시다. 크기가 365 인 해시 테이블에 23 개(테이블 크기의 겨우 6.3%입니다)의 데이터만 삽입해도 충돌이 일어날 확률이 50%가 넘습니다.

2.6 Test Code

길이가 10 이하인 문자열을 key 로, int 를 value 로 가지는 해시 테이블 구현입니다.

```
#include <cstring>
#include <iostream>

size_t djb2(const char* str) {
	size_t hash = 5381;
	for (; *str; ++str) {
		hash = ((hash << 5) + hash) + *str;
	}
	return hash;
}

constexpr size_t MAX_N = 10000;
constexpr size_t MAX_LEN = 10;

struct Node {
	char str[MAX_LEN + 1];
	int data;
	Node* next;
};

int node_count = 0;
Node nodes[MAX_N];

Node* new_node(const char str[MAX_LEN + 1], int data) {
	std::strcpy(nodes[node_count].str, str);
	nodes[node_count].data = data;
	nodes[node_count].next = nullptr;

	return &nodes[node_count++];
}

class HashMap {
	static constexpr size_t TABLE_SIZE = 1 << 12;
	static constexpr size_t DIV = TABLE_SIZE - 1;

	Node hash_table[TABLE_SIZE];
```

```
public:
	HashMap() = default;

	void init() {
		std::memset(hash_table, 0, sizeof hash_table);
		node_count = 0;
	}

	void insert(const char str[MAX_LEN + 1], int data) {
		Node* const prev_node = get_prev_node(str);
		if (prev_node->next == nullptr) {
			prev_node->next = new_node(str, data);
		} else {
			prev_node->next->data = data;
		}
	}

	void remove(const char str[MAX_LEN + 1]) {
		Node* const prev_node = get_prev_node(str);
		if (prev_node->next != nullptr) {
			prev_node->next = prev_node->next->next;
		}
	}

	Node* get(const char str[MAX_LEN + 1]) {
		return get_prev_node(str)->next;
	}

private:
	Node* get_prev_node(const char str[MAX_LEN + 1]) {
		Node* prev_ptr = &hash_table[djb2(str) & DIV];
		while (prev_ptr->next != nullptr && std::strcmp(prev_ptr->next->str, str) != 0) {
			prev_ptr = prev_ptr->next;
		}
		return prev_ptr;
	}
};

int main() {
```

```
	HashMap hash_map {};
	// 0    : 초기화
	// 1 str x : (str, x) 삽입 (이미 str 이 있는 경우 data 를 x 로 교체)
	// 2 str    : str 삭제 (str 이 없는 경우 무시)
	// 3 str    : str 검색
	int cmd, x;
	char str[MAX_LEN + 1];
	Node* ptr;
	for (;;) {
		std::cin >> cmd;
		switch (cmd) {
		case 0:
			hash_map.init();
			break;
		case 1:
			std::cin >> str >> x;
			if (std::strlen(str) > MAX_LEN) return std::cout << "invalid key length", 0;
			hash_map.insert(str, x);
			break;
		case 2:
			std::cin >> str;
			if (std::strlen(str) > MAX_LEN) return std::cout << "invalid key length", 0;
			hash_map.remove(str);
			break;
		case 3:
			std::cin >> str;
			if (std::strlen(str) > MAX_LEN) return std::cout << "invalid key length", 0;
			ptr = hash_map.get(str);
			if (ptr == nullptr) {
				std::cout << "not found₩n";
			} else {
				std::cout << str << ": " << ptr->data << '₩n';
			}
			break;
		default:
			return std::cout << "invalid command₩n", 0;
		}
	}
}
```

1. Chaining 을 linked list 로 구현했습니다.
2. Single linked list 이기 때문에 작업을 위해 이전 노드를 사용하고 있습니다.
3. 노드에 원본 문자열을 복사하는 이유는 충돌 때문입니다. 해시값이 같다면 원본 문자열을 비교합니다.

3. 기본 문제

- 문자열 교집합
- 단어가 등장하는 횟수
- 은기의 아주 큰 그림