Data Structure #11

hashing

2020년 1학기

Intro.

- **실습주제 소개**
  - hashing
- **실습수업 문제**
  - hashing

해싱(hashing)의 개념

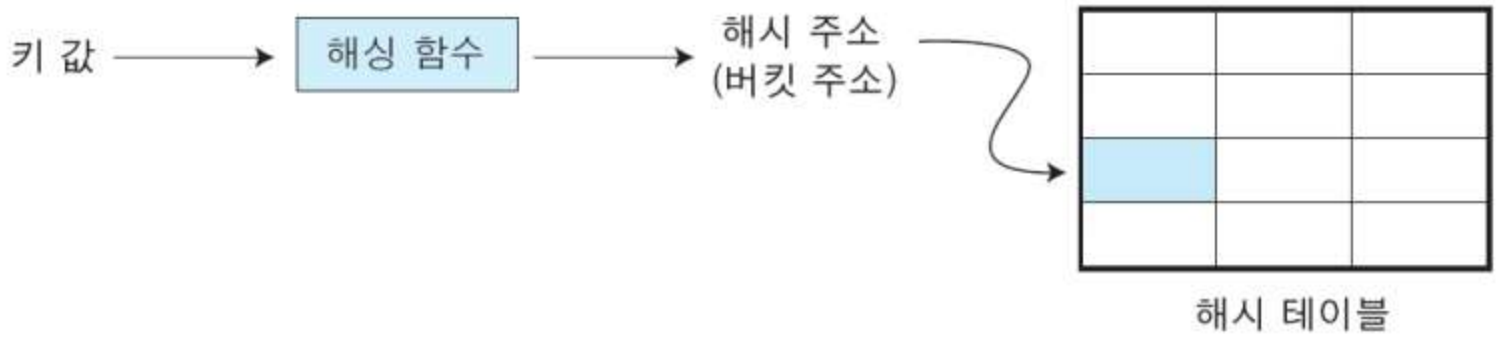
- **해싱(hashing)**
  - 산술적인 연산을 이용하여 키가 있는 위치를 계산하여 찾는 계산 검색 방식
  - 검색 방법
    - 키 값에 대해서 해시 함수를 계산하여 주소를 구하고,
    - 구한 주소에 해당하는 해시 테이블로 이동
      - 해당 주소에 찾는 항목이 있으면 검색 성공, 없으면 검색 실패
  - 해시 함수(hashing function)
    - 키 값을 원소의 위치(주소)로 변환하는 함수
  - 해시 테이블(hash table)
    - 해시 함수에 의해서 계산된 주소의 위치에 항목을 저장한 표

해싱(hashing)의 개념

- **충돌(Collision)**
  - 서로 다른 키 값에 대해서 해시 함수에 의해 주어진 버킷 주소가 같은 경우
  - 충돌이 발생한 경우에 비어있는 슬롯에 키 값 저장
- **동의어(Synonym)**
  - 서로 다른 키 값을 갖고 있으나, 해시 함수에 의해서 같은 버킷에 저장된 키 값들
- **오버플로우(Overflow)**
  - 버킷에 비어있는 슬롯이 없는 포화 버킷 상태에서 충돌이 발생하여 해당 버킷에 키 값을 저장할 수 없는 상태

해싱(hashing)의 개념

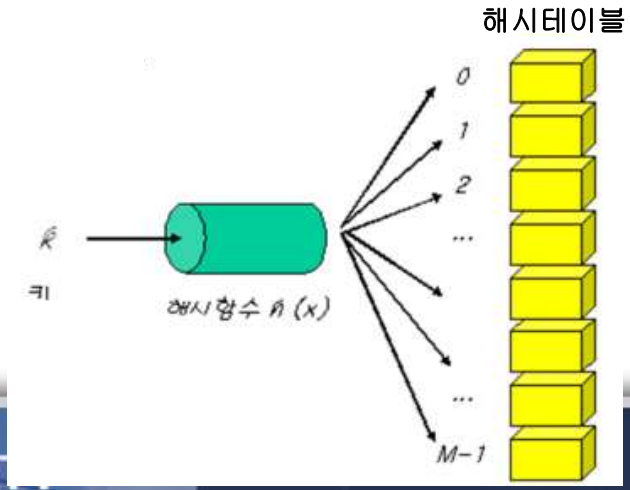
- **해시 검색 수행 방법**



  - 선형검색, 이진검색 등 방법이 사용됨

해시 함수

- **해시 함수의 조건**
  - 해시 함수는 계산이 쉬워야 한다
    - 비교 검색 방법을 사용하여 키 값의 비교연산을 수행하는 시간보다 해싱 함수를 사용하여 계산하는 시간이 빨라야 해시 검색을 사용하는 의미가 있다.
  - 해시 함수는 충돌이 적어야 한다
    - 충돌이 많이 발생한다는 것은 같은 버킷을 할당 받는 키가 많다는 것이므로 비어있는 버킷이 많아도 어떤 버킷은 오버플로우가 발생할 수 있는 상태가 되므로 좋은 해싱 함수가 될 수 없다.



해시 함수의 종류

- **제산**
  - 함수는 나머지 연산자 mod(C에서의 %연산자)를 사용하는 방법
  - 키 값 k를 해시 테이블의 크기 M으로 나눈 나머지를 해시 주소로 사용
- **승산**
  - 곱하기 연산을 사용하는 방법
  - 키 값 k와 정해진 실수 α를 곱한 결과에서 소수점 이하 부분을 테이블의 크기 M과 곱하여 그 값을 주소로 사용
- **위에서 언급한 것 외 여러 가지가 존재**

오버플로우 처리 방법

- **오버플로우 처리 방법**
  - 충돌에 의한 오버플로우 처리 방법에 따라 해시 테이블 구조가 결정됨
    - Chaining
      - 해시 테이블의 구조를 변경하여 각 버킷에 하나 이상의 키 값을 저 장할 수 있도록 하는 방법
        - » 버킷에 슬롯을 동적으로 삽입하고 삭제하기 위해서 연결 리스트 사용
    - Open Addressing(개방 주소법)
      - 한 버킷당 들어갈 수 있는 키가 한 개
        - » **선형 탐사(Linear probing)**
        - » **제곱 탐사(Quadratic probing)**
        - » **이중 해싱(double hashing)**

Chaining

- **Chaining 예시**
  - 해시 테이블의 크기 : 5
  - 해시 함수 : 제산함수 사용. 해시 함수 h(k) = k mod 5
  - 저장할 키 값 : {45, 9, 10, 96, 25}

① 키 값 45 저장 : h(45) = 45 mod 5 = 0
- 해시 테이블 0번에 노드를 삽입하고 45 저장

② 키 값 9 저장 : h(9) = 9 mod 5 = 4
- 해시 테이블 4번에 노드를 삽입하고 9 저장

③ 키 값 10 저장 : h(10) = 10 mod 5 = 0
- 해시 테이블 0번에 노드를 삽입하고 10 저장

④ 키 값 96 저장 : h(96) = 96 mod 5 = 1
- 해시 테이블 1번에 노드를 삽입하고 96 저장

⑤ 키 값 25 저장 : h(25) = 25 mod 5 = 0
- 해시 테이블 0번에 노드를 삽입하고 25 저장

Open Addressing

- **Linear Probing**

$$h_i(x) = (h(x) + i) \bmod m$$

예: 입력 순서 25, 13, 16, 15, 7, 28, 31, 20, 1, 38

0	13
1	
2	15
3	16
4	28
5	
6	
7	7
8	
9	
10	
11	
12	25

0	13
1	
2	15
3	16
4	28
5	31
6	
7	7
8	20
9	
10	
11	
12	25

0	13
1	1
2	15
3	16
4	28
5	31
6	38
7	7
8	20
9	
10	
11	
12	25

$h_i(x) = (h(x) + i) \bmod 13$

Open Addressing

- **Quadratic Probing**

예: 입력 순서 15, 18, 43, 37, 45, 30

0	
1	
2	15
3	
4	43
5	18
6	45
7	
8	30
9	
10	
11	37
12	

$$h_i(x) = (h(x) + i^2) \bmod 13$$

Open Addressing

- **Double Hashing**

예: 입력 순서 15, 19, 28, 41, 67

0	
1	
2	15
3	
4	67
5	
6	19
7	
8	
9	28
10	
11	41
12	

$h_0(15) = h_0(28) = h_0(41) = h_0(67) = 2$

$h(x) = x \bmod 13$

$f(x) = (x \bmod 11) + 1$

$h_i(x) = (h(x)+if(x)) \bmod 13$

해싱 구현(개인 실습#1)

구조체

```
#define KEY_SIZE  10          // 탐색키의 최대길이
#define TABLE_SIZE            13          // 해시 테이블의 크기=소수
typedef struct
{
    char key[KEY_SIZE];
    // 다른 필요한 필드들
} element;

struct list
{
    element item;
    struct list *link;
};
struct list *hash_table[TABLE_SIZE];
```

해싱 구현

- 출력 함수

```
void hash_chain_print(struct list *ht[])
{
   struct list *node;
   int i;
   for(i=0;i<TABLE_SIZE;i++){

   }
}
```

해싱 구현

– 문자로 된 탐색키를 숫자로 변환 / 해싱 함수

```
// 문자로 된 탐색키를 숫자로 변환
int transform(char *key)
{
        // 간단한 덧셈 방식 사용 자연수 생성
}

// 제산 함수를 사용한 해시 함수
int hash_function(char *key)
{
        // 키를 자연수로 변환한 다음 테이블의 크기로 나누어 나머지를 반환
}
```

해싱 구현

– 테이블에 키를 삽입

```
void hash_chain_add(element item, struct list *ht[])
{



}
```

persucode :

```
Compute the hash key
If slot at hash key is null
   Insert as first node of chain
Else
   Search the chain for a duplicate key
   If duplicate key
      Don' t insert
   Else
      Insert into chain
   Endif
Endif
```

구현된 코드로 설명:

0. ht[] 해시테이블, item에는 key를 지니고 있다.

1. 체인을 따라서 duplicate key를 찾는다.
   만일 dublicate key라면 return;

2. duplicate key가 아니라면 체인에 insert한다.
   * insert 방법은 아래와 같다.

```
ptr = (struct list *)malloc(sizeof(struct list));
ptr->item = item;
ptr->link = NULL;

if(node_before) node_before->link = ptr;
else ht[hash_value]=ptr;
```

해싱 구현

- 테이블에 저장된 키를 탐색

```
void hash_chain_find(element item, struct list *ht[])
{
   struct list *node;

   int hash_value = hash_function(item.key);

            //
}
```

해싱 구현

- 메인함수

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
void main()
{
        element tmp;
        strcpy(tmp.key, "a");
        hash_chain_add(tmp, hash_table);
        strcpy(tmp.key, "b");
        hash_chain_add(tmp, hash_table);
        strcpy(tmp.key, "c");
        hash_chain_add(tmp, hash_table);
        strcpy(tmp.key, "d");
        hash_chain_add(tmp, hash_table);
        strcpy(tmp.key, “a”);
        hash_chain_add(tmp, hash_table);
        strcpy(tmp.key, “ai”);
        hash_chain_add(tmp, hash_table);

        hash_chain_print(hash_table);

}
```

```
already exists
[0]->
[1]->
[2]->
[3]->
[4]->
[5]->
[6]->a->
[7]->b->ai->
[8]->c->
[9]->d->
[10]->
[11]->
[12]->
```

제출

- **제출**
  - 개인 실습 (#1)
    - 오늘 자정까지 제출 (~ 2020/5/29 23:59)
    - Hash (chaining)
  - 과제
    - Lab11.docx
    - 다음주 목요일 자정까지 제출 (~2020/6/4 23:59)
    - Hash (Open Addressing – Linear probing)

제출

- **Lab11.docx**
  - 구조체

```
typedef int ElementType;

struct HashTable{
        int TableSize;
        ElementType*  TheLists;
};
```

제출

- **Lab11.docx**
  - input

```
30
3  5  35  2  7 18  19  22  5  100  26  8  4 16
5 27 45 67 2
```

```
Collision for the Key [35] has occured.
Duplicated value [5]
The key [5] exists in the list
The key [27] doesn't exist in the list
The key [45] doesn't exist in the list
The key [67] doesn't exist in the list
The key [2] exists in the list

☆Hash table Print out☆

<0> =>
<1> =>
<2> => [2]
<3> => [3]
<4> => [4]
<5> => [5]
<6> => [35]
<7> => [7]
<8> => [8]
<9> =>
<10> => [100]
<11> =>
<12> =>
<13> =>
<14> =>
<15> =>
<16> => [16]
<17> =>
<18> => [18]
<19> => [19]
<20> =>
<21> =>
<22> => [22]
<23> =>
<24> =>
<25> =>
<26> => [26]
<27> =>
<28> =>
<29> =>
```