본 PSet 은 저의 강의 경험과 학생들의 의견 및 Stanford CS106 과 Harvard CS50 같은 강의에서 수집된 자료를 토대로 작성되었습니다. 본 PSet 에 문제가 있거나, 질문 혹은 의견이 있다면, 언제든지 알려 주시면 강의 개선에 많은 도움이 되겠습니다. idebtor@gmail.com

Final: Hashing

목차

해싱

해싱은 **해시 함수**라고 불리는 특별한 함수를 사용하도록 설계된 중요한 데이터 구조입니다. 해시 함수는 요소의 **더 빠른 접근**을 위해 주어진 값을 특정 키와 매핑하는 데에 사용합니다. 해싱은 삽입, 제거, 검색을 평균 케이스 상수 시간 O(1)로 지원합니다. 다음 테이블은 이번 학기에 배운 자료 구조들의 시간 복잡성을 요약한 것입니다.

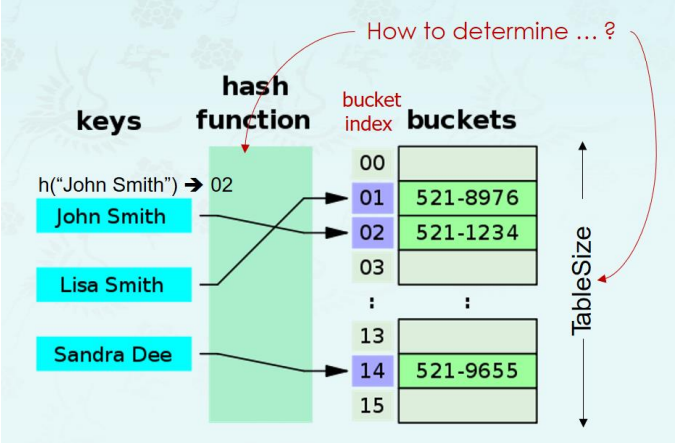
C++ STL Data Structure	Insert	Find	Delete
vector	O(n)	O(n)	O(n)
sorted vector	O(n)	O(log n)	O(n)
linked list (list, stack, queue)	O(1)	O(n)	O(1)

balanced BT (map, set)	O(log n)	O(log n)	O(log n)
Heap (priority_queue)	O(log n)	O(1) for min/max	O(log n)
hash table (unordered_map/set)	O(1)	O(1)	O(1)

해시 테이블 주요 구성 요소

따라서 해싱은 아래에 언급된 2 가지 단계를 통해 구현된다고 할 수 있습니다.

1) 해시 함수를 이용해서 값을 고유 정수 키 또는 해시로 변환합니다. 이 값은 해시 테이블에 속하는 기존 요소를 저장하기 위한 인덱스로 사용합니다.
2) 데이터 배열의 요소를 해시 키를 사용하여 빠르게 검색할 수 있는 해시 테이블에 저장합니다.



위 다이어그램에서 해시 함수를 사용하여 각각의 위치를 계산한 후 해시 테이블에 모든 요소를 저장한 것을 확인할 수 있습니다. 해시 값과 인덱스를 얻기 위해 다음 식을 사용할 수 있습니다.
hash = hash_func(키)
index = hash % table_size

해시 함수

매핑의 효율성은 사용되는 해시 함수의 효율성에 좌우됩니다. 해시 함수는 기본적으로 다음과 같은 요구 사항을 충족해야 합니다.

- 간편한 계산: 해시 함수는 고유 키를 간편하게 계산할 수 있어야 합니다.
- 적은 충돌: 요소가 동일한 키값과 같으면 충돌이 발생합니다. 사용되는 해시 함수에서 충돌은 가능한 최소화해야 합니다. 충돌은 일어나기 마련이므로 충돌을 처리하기 위해 적절한 충돌 해결 기술을 활용해야 합니다.
- 균등 분포: 해시 함수는 해시 테이블 전체에 걸쳐 데이터를 균일하게 분산시켜 군집화를 방지해야 합니다.

해시 테이블

해시 테이블 또는 해시 맵은 원본 데이터 배열의 요소에 대한 포인터를 저장하는 데이터 구조입니다. 해시 테이블에 항목이 있으면 배열에서 특정 요소를 더욱 쉽게 검색할 수 있습니다. 해시 테이블은 해시 함수를 사용하여 원하는 값을 찾을 수 있는 버킷 또는 슬롯의 배열로 인덱스를 계산합니다.

충돌

해싱의 경우, 아주 큰 크기의 테이블을 가지고 있어도 충돌이 불가피합니다. 일반적으로 큰 키에 대한 작은 고윳값을 찾으므로 어느 시점이든 하나 이상의 값이 동일한 해시 코드를 가질 수 있습니다.

해싱에서 충돌이 불가피하다는 점을 고려하여 충돌을 예방하거나 해결할 방법을 항상 찾아보아야 합니다. 해싱 도중에 발생하는 충돌을 해결하기 위해 사용할 수 있는 다양한 충돌 해결 기법이 있습니다.

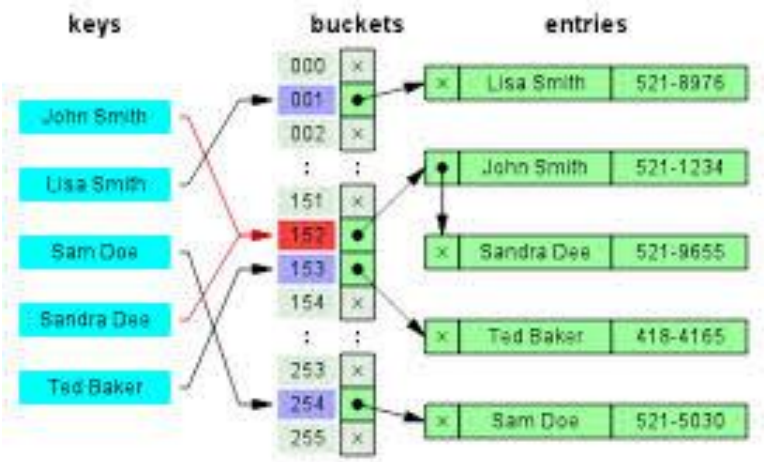
충돌 해결 기술

충돌 해결 기법은 크게 2 가지가 있습니다. 하나는 개방 해싱 또는 분리 연결법(separate chaining)입니다. 다른 하나는 폐쇄 해싱 또는 개방 주소법(open addressing)이라고 불립니다.

- **분리 연결법**(또는 개방 해싱)
- **개방 주소법**(또는 폐쇄 해싱)
  - 선형 탐사(Linear Probing)
  - 제곱 탐사(Quadratic probing)
  - 이중 해싱(Double hashing)

개방 해싱 또는 분리 연결법

개방 해싱(Open Hashing)은 해시 테이블의 해시 키 인덱스(k)에 데이터가 직접 저장되지 않는 기법입니다. 오히려 해시 테이블의 키 인덱스(k)에 있는 데이터는 데이터가 실제로 저장되는 데이터 구조의 헤드를 가리키는 포인터입니다. 가장 단순하고 보편적인 구현에서 요소를 저장하기 위해 채택된 데이터 구조는 링크된 구조입니다.



해시 테이블의 각 엔트리(entry)는 연결 리스트입니다. 키가 해시 코드와 일치하면, 해당 해시 코드와 일치하는 리스트에 입력됩니다. 따라서 두 키가 동일한 해시 코드를 가지면, 두 항목이 연결 리스트에 입력됩니다. 분리 연결법의 최악의 경우는 모든 키가 동일한 해시 코드를 가져서 하나의 연결 리스트에만 삽입되는 경우입니다. 따라서 해시 테이블의 모든 항목과 테이블의 키의 개수에 비례하는 비용을 확인해야 합니다.

폐쇄 해싱 또는 개방 주소법

이 기법에서는 미리 식별된 크기의 해시 테이블을 다룹니다. 모든 항목은 해시 테이블 자체에 저장됩니다. 데이터 외에도 각 해시 버킷은 3 가지 상태(EMPTY, OCCUPIED, DELETED)를 유지합니다. 삽입 기능을 수행하는 동안 충돌이 발생하면 빈 버킷이 발견될 때까지 대체 셀에 삽입을 시도합니다. 다음 기술 중 하나가 채택됩니다.

선형 탐사

개방 주소법 또는 선형 탐사 기법에서는 모든 엔트리의 레코드를 해시 테이블 자체에 저장합니다. 키값이 해시 코드에 매핑되고 해시 코드가 가리킨 위치가 비어 있을 때, 키값을 그 위치에 삽입합니다. 위치가 이미 점유(occupied)된 경우, 탐사 시퀀스를 사용하여 해시 테이블에서 비어 있는 다음 위치에 키값을 삽입합니다.

선형 탐사의 경우 해시 함수가 아래와 같이 변경될 수 있습니다.

hash = hash % hashTableSize
hash = (hash + 1) % hashTableSize
hash = (hash + 2) % hashTableSize
hash = (hash + 3) % hashTableSize

선형 탐사는 연속적인 셀을 점유할 가능성이 있고 새로운 요소를 삽입할 확률이 감소하는 "일차 군집화"(primary clustering) 문제를 겪을 수 있습니다.

제곱 탐사

제곱 탐사는 선형 탐사와 동일하며 탐사에 사용하는 간격만 다릅니다. 이름에서 알 수 있듯이 이 기법은 충돌이 발생할 때 선형 거리 대신 비선형 또는 제곱 거리를 사용하여 슬롯을 점유합니다.

제곱 탐사에서 슬롯 사이의 간격은 이미 해시된 인덱스에 임의의 다항식 값을 추가하여 계산합니다. 이 기법은 일차 군집화를 크게 감소시키지만 이차 군집화를 개선하지는 않습니다.

hash = hash % hashTableSize
hash = (hash + 1) % hashTableSize
hash = (hash + 4) % hashTableSize
hash = (hash + 9) % hashTableSize

이중 해싱

이중 해싱 기술은 선형 탐사와 유사합니다. 이중 해싱과 선형 탐사의 유일한 차이점은 이중 해싱 기법에서 탐사에 사용되는 간격은 두 개의 해시 함수를 사용하여 계산된다는 점입니다. 키에 해시 함수를 차례대로 적용하므로 일차 군집화 뿐만 아니라 이차 군집화도 제거합니다.

hash(x) = hash % hashTableSize
hash(x) = (hash + 1 * (R - (x % R)) % hashTableSize, R is prime number less than hashTableSize
hash(x) = (hash + 2 * (R - (x % R)) % hashTableSize
hash(x) = (hash + 3 * (R - (x % R)) % hashTableSize

재해싱(Rehashing)

재해싱은 이미 저장된 엔트리들(키-값 쌍)의 해시 값을 다시 계산하여 한계점에 도달하거나 초과할 때 다른 더 큰 크기의 해시 테이블로 옮기는 과정입니다. 새로운 키-값 쌍을 맵에 삽입할 때마다 적재율(load factor)이 증가하여 복잡도가 증가하므로 재해싱을 수행합니다. 복잡도가 증가하면 이 해시 맵은 상수 O(1) 시간 복잡도를 가지지 않습니다.

방법:

- 맵의 모든 새로운 엔트리의 적재율을 확인합니다.
- 적재율이 한계치(해시 테이블의 경우 기본값 1.0)보다 크면 재해싱합니다.=
  - 재해싱의 경우, 이전 배열의 두 배 크기와 다음 소수(prime)로 새로운 배열을 초기화합니다.
  - 새로운 배열에 모든 요소를 재삽입하여 새 버킷 배열로 만듭니다.

시작하기

이 Pset 은 3 가지 유형의 해시 테이블을 구현합니다.

- 첫 번째 방법은 해시 테이블과 문자열 타입 키로 STL **리스트** 배열을 사용합니다.
- 두 번째 방법은 해시 테이블에 리스트 배열을 사용합니다. 각 요소는 각각 키와 값에 대한 문자열과 정수형 데이터로 구성됩니다.
- 세 번째 방법은 STL 의 **unordered_map** 클래스를 사용하여 hash2.cpp 의 기능을 대체합니다.

파일 목록은 다음과 같이 제공됩니다.

- hashing.pdf - 본 파일
- hash1.h, **hash1.cpp**, hash1Driver.cpp - 문자열 유형 요소를 저장하는 해시 테이블
- hash2.h, **hash2.cpp**, hash2Driver.cpp - 사용자 정의 데이터 유형을 저장하는 해시 테이블
- **hashmap.cpp** - hash2 의 기능은 STL 의 **unordered_map** 을 사용하여 시뮬레이션합니다.
- hash1x.exe, hash2x.exe, hashmapx.exe - 예시 답안
- makefile - 실행 파일 3 개를 빌드하기 위한 makefile - 편의를 위해 제공됩니다.
- 다음 text(ASCII) 파일은 테스트를 위해 제공됩니다. 이 파일에 직접 액세스하려면 자신의 실행 파일 또는 MS 프로젝트 파일이 저장된 위치에 파일을 저장하세요.
  ps23.txt - 시편 23 편 (NIV)
  1co13.txt - 고린도전서 13 장 (NIV)

kjv.txt - 킹 제임스 성경
shakespeare.txt - 세익스피어 작가의 "한여름 밤의 꿈"

hash1.cpp, hash2.cpp, 그리고 **hashmap.cpp** 에서 코딩을 완료해야 합니다.

Step1: hash1.cpp 완성하기

이 hash1.cpp 뼈대 코드는 STL 의 **리스트** 클래스를 사용하여 해시 테이블을 구현합니다. 해시 구조는 아래와 같이 **hash1.h** 에 정의되어 있습니다.

```
struct Hash {
  int           tablesize;             // hash table size
  list<string>* hashtable;             // pointer to an array of buckets
  int           nelements;             // number of elements in table
  double        threshold;             // max_loadfactor

  Hash(int size = 2, double lf = 1.0) {// a magic number, use a small prime
     tablesize = size;                  // using list<string> for pedagogical purpose
     hashtable = new list<string>[size];// but vector<list<string>> may be used
     nelements = 0;
     threshold = lf;                    // rehashes if loadfactor >= threshold
  }
  ~Hash() {
     delete[] hashtable;
  }
};
```

실행 파일을 빌드하는 3 가지 방법

실행 파일을 빌드하는 몇 가지 방법이 있습니다. 예를 들어, hash1.exe 를 빌드해 봅시다.

1. hash1Driver.cpp 미포함 - 빠른 코드 테스트를 위한 것입니다.
   A. hash1.cpp 의 main() 위에 매크로를 #if 1 로 설정합니다.
   B. 다음 빌드 명령을 사용합니다: OSX 는 **-std=c++11** 를 추가하세요.

   g++ hash1.cpp -o hash1 -I../../include

2. hash1Driver.cpp 포함 - 개발을 위한 것입니다.
   A. hash1.cpp 의 main() 위에 매크로를 #if 0 으로 설정합니다.
   B. 다음 빌드 명령을 사용합니다.
      OSX 는 **-std=c++11** 를 추가하고 **-lnowic** 대신 **-lnowic_mac** 를 사용하세요.

   g++ hash1.cpp hash1Driver.cpp -o hash1 -I../../include -L../../lib -lnowic

3. makefile 을 사용하세요. 콘솔에 "make"만 입력하면 됩니다. 이렇게 하면 세 가지 실행 파일들(hash1.exe, hash2.exe, hashmap.exe)을 모두 검사하고 필요한 경우 빌드합니다. 필요에 따라 또는 파일 이름에 따라 제공된 makefile 을 편집해야 할 수도 있습니다.

   make

(hash1Driver.cpp 을 포함한) 실행 예시:

```
[HashTable] tablesizeM:2 sizeN:0 max_loadfactr:1 loadfactor(N/M):0
i - insert                      e - erase
j - insert N                    x - erase N
f - find                        h - max_loadfactor & rehash
o - show empty buckets[ON]      c - clear

Command(q to quit): j

Enter keys to insert: In the beginning God created the heavens and earth.
REHASHED(tablesize: 2 -> 5)
REHASHED(tablesize: 5 -> 11)

[0]
[1]  heavens
[2]  and
[3]
[4]
[5]
[6]  In
[7]  beginning created
[8]  God the the
[9]
[10]  earth.

[HashTable] tablesizeM:11 sizeN:9 max_loadfactr:1 loadfactor(N/M):0.818182
i - insert                      e - erase
j - insert N                    x - erase N
f - find                        h - max_loadfactor & rehash
o - show empty buckets[ON]      c - clear

Command(q to quit):
```

insert(), erase(), find() 및 rehash()와 같은 해시 테이블을 다루기 위한 일반적인 커맨드 집합을 구현합니다. 간소화를 위해 이 구현은 값이 아닌 키만 사용할 수 있습니다. 충돌을 해결하기 위해 분리 **연결법(Separate Chaining)**방식을 사용합니다.

1. **insert**
   A. 사용자로부터 키를 받아 해시 테이블에 삽입합니다.
   B. 키를 삽입한 **후** 적재율이 max_loadfactor(또는 코드의 한계치)보다 크다면 재해시합니다.
2. **erase**
   A. 키를 가져와 해시 테이블에서 삭제합니다. 재해시는 필요하지 않습니다. 해시 테이블은 요소의 수가 줄어들더라도 더 작아지기 위해 재해시하지 않습니다.
3. insert N and erase N
   A. insert 또는 erase 와 같이 작동하지만 여러 개의 입력 키가 필요합니다.
4. **find**
   A. 사용자로부터 키를 받아 테이블에서 검색한 후 결과를 표시합니다.
   B. 키와 동일한 버킷을 반환합니다. 해시 값이 동일한 요소가 여러 개 있을 수 있습니다. 메인 함수 드라이버는 버킷의 모든 요소를 출력합니다.
5. max_loadfactor & **rehash**
   A. 부동 소수점 숫자를 받아서 max_loadfactor 를 설정합니다.
   B. max_loadfactor 와 loadfactor 를 비교합니다.
   만약 loadfactor ≥ max_loadfactor 이라면 재해시합니다.
   C. 재해시 후 max_loadfactor 가 설정되고 loadfactor 가 제대로 표시됩니다.
6. clear
   A. 해시 테이블의 모든 요소를 제거하고 새로운 해시 테이블을 시작합니다.

7. show empty bucket [ON]
   A. 빈 버킷 표시 여부를 설정/해제합니다.

Step2: hash2.cpp 완성하기

이 hash2.cpp 뼈대 코드 또한 STL 의 **리스트** 클래스를 사용하여 해시 테이블을 구현합니다. 각 요소는 "wordcount"라 불리는 (문자열 유형 데이터와 정수 유형 데이터와 같은) 데이터 쌍으로 구성되어 있습니다. 다음과 같이 정의할 수 있습니다.

해시 구조는 아래와 같이 **hash2.h** 에 정의되어 있습니다.

```
typedef std::pair<string, int> wordcount;

struct Hash {
  int               tablesize;            // hash table size or bucket_count()
  list<wordcount>* hashtable;             // pointer to an array of buckets
  int               nelements;            // number of elements in table or size()
  double            threshold;            // threshold(or max_loadfactor)

  Hash(int size = 2, double lf = 1.0) {  // a magic number, use a small prime
     tablesize = size;
     hashtable = new list<wordcount>[size];
     nelements = 0;
     threshold = lf;                      // rehashes if loadfactor >= threshold
  }
  ~Hash() {
     delete[] hashtable;
  }
};
```

실행 예시:

```
[HashTable] tablesizeM:2 sizeN:0 max_loadfactor:1 loadfactor(N/M):0
i - insert                    e - erase
j - insert N                  x - erase N
k - insert by file            z - erase by file
f - find                      h - max_loadfactor & rehash
s - show [ALL] buckets        t - tablesize & rehash
o - show empty buckets[ON]    c - clear

Command(q to quit): j

Enter keys to insert: In the beginning God created the earth and heavens.
REHASHED(tablesize: 2 -> 5)
REHASHED(tablesize: 5 -> 11)
cpu: 0.007 sec
[0]
[1]  heavens.: 1
[2]  and: 1
[3]
[4]  earth: 1
[5]
[6]  In: 1
[7]  beginning: 1       created: 1
[8]  God: 1     the: 2
[9]
[10]

[HashTable] tablesizeM:11 sizeN:8 max_loadfactor:1 loadfactor(N/M):0.727273
i - insert                    e - erase
j - insert N                  x - erase N
k - insert by file            z - erase by file
f - find                      h - max_loadfactor & rehash
s - show [ALL] buckets        t - tablesize & rehash
o - show empty buckets[ON]    c - clear

Command(q to quit):
```

insert(), erase(), find() 및 rehash()와 같은 해시 테이블을 다루기 위한 일반적인 커맨드 집합을 구현합니다. 분리 연결법을 사용하여 충돌을 해결합니다.

1. **insert**
   A. 사용자로부터 키를 받아 해시 테이블에 삽입하고, 새로운 키인 경우 값을 1 로 설정합니다. 중복 키인 경우 값을 1 증가합니다.
   예를 들어, 위 화면 캡처에 "the 2"가 표시되므로 두 개의 "the"가 있어야 합니다. 버킷 또는 요소가 단어 카운터(word counter)와 같은 역할을 합니다.
   B. 키를 삽입한 **후** 적재율이 max_loadfactor(또는 코드의 한계치)보다 크거나 같다면 재해시합니다.
2. **erase**
   A. 키를 가져와 해시 테이블에서 삭제합니다. 카운트를 1 감소하는 것이 아닌, 요소 자체를 제거합니다. 재해시는 필요하지 않습니다. 해시 테이블은 요소의 수가 줄어들더라도 더 작아지기 위해 재해시하지 **않습니다**.
3. **insert N and erase N**
   A. insert 또는 erase 와 같이 작동하지만 여러 개의 입력 키가 필요합니다.
4. **insert by file, erase by file**
   A. 해시 테이블의 모든 단어를 파일에서 읽어올 때 모두 삽입하거나 삭제합니다.
   B. 숫자로 시작하는 단어는 제외합니다.
   C. 제거 후 마침표로 끝나는 단어를 포함합니다.
5. **find**
   A. 사용자로부터 키를 받아 테이블에서 검색한 후 결과를 표시합니다.
   B. 키의 해시 값과 동일한 버킷을 반환합니다. 해시 값이 동일한 요소가 여러 개 있을 수 있습니다. 메인 함수 드라이버는 [7]과 [8] 버킷과 같은 버킷의 모든 요소를 출력합니다.
6. max_loadfactor & **rehash**
   A. 부동 소수점 숫자를 받아서 max_loadfactor 를 설정합니다.
   B. max_loadfactor 와 loadfactor 를 비교합니다.
   만약 loadfactor ≥ max_loadfactor 이라면 재해시합니다.
   C. 재해시 후 max_loadfactor 가 설정되고 loadfactor 가 제대로 표시됩니다.
7. tablesize & **rehash**
   A. 정수 숫자를 받아서 **새로운 tablesize 로 설정합니다**.
   이 경우에 tablesize 는 소수(prime number)가 아닐 수 있습니다.
   B. 새로운 테이블 크기가 이전 테이블 크기와 다르다면 재해시해야 합니다..
8. show empty bucket [ON]
   A. 빈 버킷 표시 여부를 설정/해제합니다.

Step3: hashmap.cpp 완성하기

STL 의 **unordered_map** 을 사용하여 이전 단계에서 구현한 것과 동일한 기능을 구현합니다. "Find" 커맨드를 제외한 hash2.cpp 의 결과가 hashmap 의 결과와 최대한 일치하도록 구현합니다. 의도적으로 또는 의도하지 않게 일치하지 않는 것이 있다는 점을 기억하세요.

- **"find" 옵션:**

- "find" 커맨드는 <string, int>의 목록이 아닌 **일치하는 <string, int> 요소**를 반환합니다. 이 부분이 hash2.cpp 와 다릅니다.

- **"show [ALL/N] buckets" 옵션:**
  - 이전 단계와 동일한 방식으로 구현합니다.
- 재해시 후 max_loadfactor 가 설정되고 loadfactor 가 제대로 표시됩니다.

unordered_map 에 대한 관찰:

- unordered_map 에서 사용되는 해시 함수는 has1 과 hash2 에서 사용되는 것과 다릅니다.
- 소수(prime number)를 자주 사용하지 않으므로 테이블 크기와 연속 더블링 방식(successive doubling scheme)이 다릅니다.
- **reserve()** 함수를 사용하여 테이블 크기(버킷 카운트)를 조정할 수 있습니다. 아래의 c++ 참조에 따르면, N < M 이라도 크기를 **줄일 수 없습니다**.
  - 컨테이너의 버킷 수(bucket_count)를 적어도 n 개의 요소를 포함하기에 가장 적절한 값으로 설정합니다.
  - n 이 현재 bucket_count 에 max_load_factor 을 곱한 값보다 크면 컨테이너의 bucket_count 가 증가하고 재해시 작업이 강행됩니다.
  - **n 이 bucket_count 와 max_load_factor 을 곱한 값보다 작으면, 함수가 별다른 영향을 미치지 않습니다.**

STL 의 unordered_map 클래스를 사용하므로 hashmap.cpp 는 nowic.h 외의 다른 파일의 영향을 받지 않는 점에 **주의하세요**.

```
[HashTable] tabelsizeM:1 sizeN:0 max_loadfactor:1 loadfactor(N/M):0
i - insert                      e - erase
j - insert N                    x - erase N
k - insert by file              z - erase by file
f - find                        m - max_loadfactor & rehash
s - show [10] buckets           t - tablesize & rehash
o - show empty buckets[OFF]     c - clear
Command(q to quit): j
Enter keys to insert: In the beginning God created the earth and heavens.
time elapsed: 0.001 sec
[2]  In: 1
[4]  heavens.: 1        earth: 1        the: 2
[5]  God: 1
[10]  and: 1
[11]  beginning: 1
[12]  created: 1

[HashTable] tabelsizeM:13 sizeN:8 max_loadfactor:1 loadfactor(N/M):0.615385
i - insert                      e - erase
j - insert N                    x - erase N
k - insert by file              z - erase by file
f - find                        m - max_loadfactor & rehash
s - show [10] buckets           t - tablesize & rehash
o - show empty buckets[OFF]     c - clear
Command(q to quit):
```

Unordered_map 클래스를 간단하고 빠르게 테스트하는 방법

hashmap.cpp 의 맨 아래에 unordered_map 클래스를 테스트하기 위한 간단한 코드가 포함되어 있습니다. 얼마든지 이 테스트 코드를 사용해서 unordered_map 에 대해 공부해 보세요.

nowic.cpp 함수를 사용하는 main() 위에 **#if 0** 을 정의하여 매크로를 끌 수 있습니다. 그런 다음 nowic.cpp 없이 hashmap.cpp 를 빌드할 수 있습니다. 제공된 makefile 을 원하는 대로 사용해도 됩니다.

과제 제출

- 소스 파일 상단에 아래와 같이 아너 코드 문장을 적고 서명하세요.
  On my honor, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.
  서명: _____ 분반: _____ 학번: _____
- 제출하기 전에 코드가 제대로 컴파일이 되고 실행되는지 확인하세요. 제출 직전에 급하게 코드를 수정한 후 코드가 제대로 컴파일이 될 거라고 짐작하지 않는 게 좋습니다. "거의" 작동하는 코드도 틀린 것입니다.
- 과제가 컴파일 및 실행된다면, 마감 기한 전까지 과제의 일부만 완성했더라도 제출하기 바랍니다. 컴파일 및 실행되지 않는다면 제출하지 마세요. 마감 시간 이후 24 시간 이내 제출하면, 만점에서 25% 감점하고 채점합니다. 그 이상 늦은 것은 채점하지 않으며, 0 점 처리합니다.
- 제출 후, **마감 기한 전까지** 수정 및 재제출이 가능합니다. 파일 하나만 수정하더라도 해당 파일과 관련된 파일들을 모두 재제출해야 합니다. 재제출 횟수는 제한 없습니다. 마감 기한 전에 **가장 마지막으로** 제출된 파일을 채점할 것입니다.

제출 파일 목록 (PSet final)

final folder 에 다음 파일들을 제출합니다**.** 코드가 콘솔에서 g++로 컴파일되고 실행되는지 확인하세요.

- Step 1: hash1.cpp
- Step 2: hash2.cpp
- Step 3: hashmap.cpp
- 08-1 HashingFinal.pptx (이 파일을 편집하여 답변을 작성하세요.)

마감 기한 & 배점

- 마감 기한: 기말 고사이므로 **늦은 제출은 받지 않습니다**.

해싱 무료 온라인 강의

아래 두 강의를 권장합니다. 더 좋은 강의를 찾는다면 공유해 주세요.

1. 컴퓨터 알고리즘 기초 10 강 해쉬 알고리즘(1) | T 아카데미
2. 컴퓨터 알고리즘 기초 11 강 해쉬 알고리즘(2) | T 아카데미