

STL 기초

Hash

한컴에듀케이션



Hash?

- Search의 한가지 방법
- key값과 일치하는 data를 빠른 시간에 검색하기 위해 사용
- key 값은 고유 값 (ex. 회원id, 주민번호)
- indexing 기법 활용

Search : 원하는 data의 실제 저장 정보를 찾아나가는 방법

Linear Search

Binary Search

Hash

Trie

etc

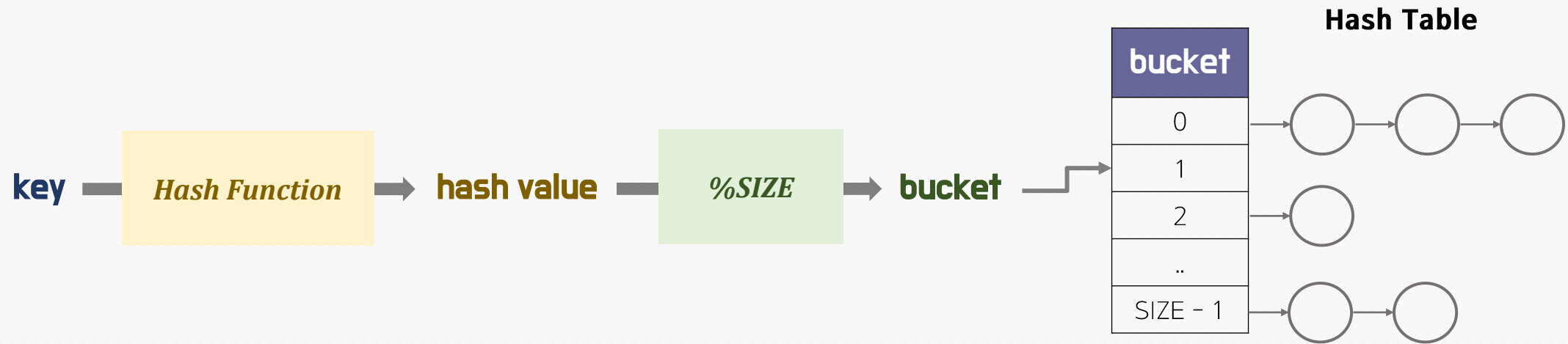
Linear Search

data의 수, 검색 횟수가 적은 경우 사용



	name	no	age
1	son	7	3
2	roony	1	4
3	kane	2	65
4	terry	34	6
5	santez	5	7
6	herry	6	0
7	kim	7	9

Hash



key

int
long long
int[]
char[]
int[][]
...

hash value

unsigned int
unsigned long long

bucket

0 ~ SIZE - 1

probing

구한 bucket에 저장된 모든 원소들을 검색하며 원하는 data를 찾는다.

동일한 key값은 항상 같은 bucket

다른 key값이어도 같은 bucket 가능 (충돌)

Hash 설정

1. Hash Key

- data의 고유값을 나타낼 수 있는 정보로 설정

2. Hash Function

- hash value 생성 : *unsigned int* or *unsigned long long*
- 일반적으로 key값을 전부 활용하여 진법 변환
- key값의 일부만 활용하는 경우도 존재

3. Bucket

- 나눗셈 법 : $hash\ value \% SIZE$

• 곱셈 법

4. Collision 처리 방식

- Chaining

bucket 별로 리스트화 하여 관리

• Linear Probing

5. Hash Table Size

- chaining 시

hash table에 등록되는 data수 이상의 2의 제곱수

• linear probing 시

최소 data수의 두배 이상

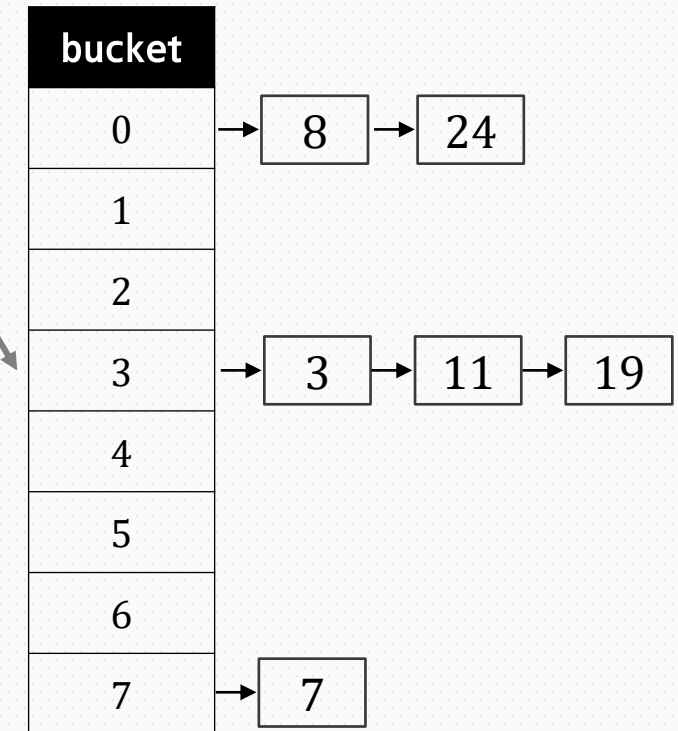
Example 1

SIZE = 8
key 값 = 정수 한 개



key	hash value	bucket
3	3	3
8	8	0
7	7	7
11	11	3
19	19	3
24	24	0

Hash Table



Example2

SIZE = 8

key 값 = 정수 한 개

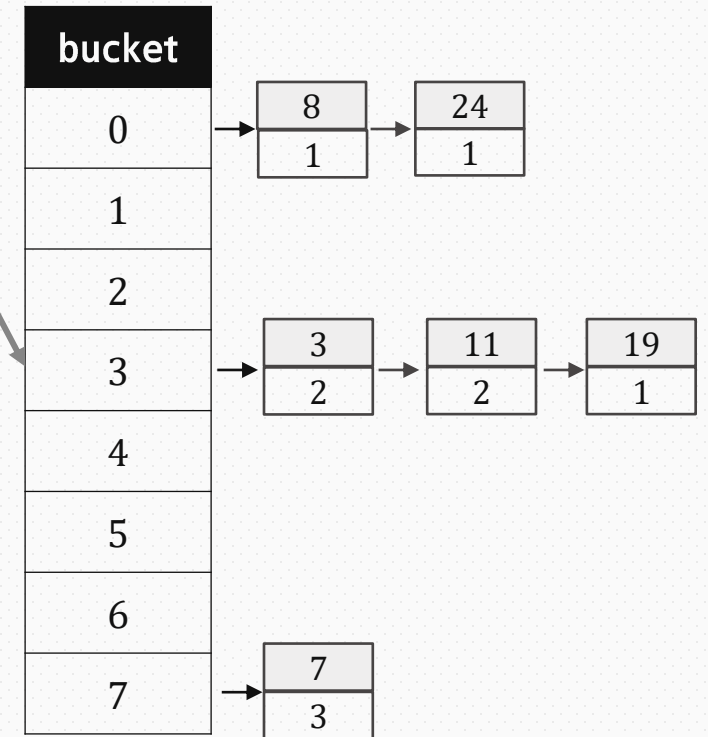
value 값 = 검색 횟수



검색 : 3, 8, 11, 11, 3, 19, 24, 7, 7, 7

key	hash value	bucket
3	3	3
8	8	0
7	7	7
11	11	3
19	19	3
24	24	0

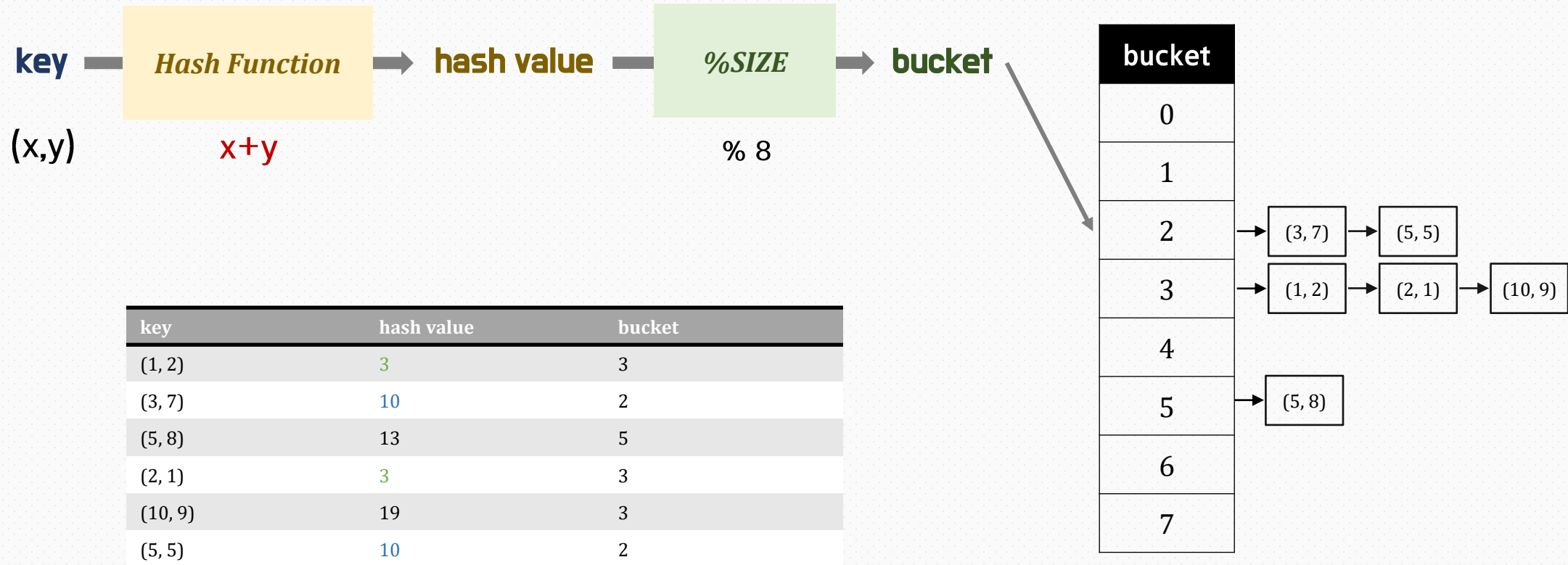
Hash Table



Example3

SIZE = 8

key 값 = (0~99,999 , 0~99,999) 범위의 (x, y) 좌표

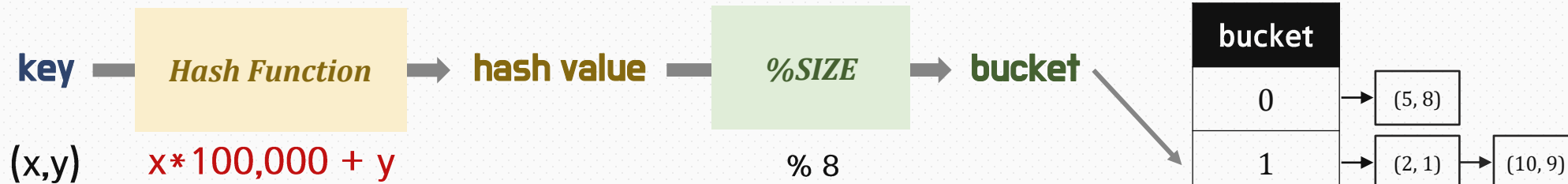


Example4

SIZE = 8

key 값 = (0~99,999 , 0~99,999) 범위의 (x, y) 좌표

Hash Table



key	hash value	bucket
(1, 2)	10,002	2
(3, 7)	30,007	7
(5, 8)	50,008	0
(2, 1)	20,001	1
(10, 9)	100,009	1
(5, 5)	50,005	5

bucket	
0	→ (5, 8)
1	→ (2, 1) → (10, 9)
2	→ (1, 2)
3	
4	
5	→ (5, 5)
6	
7	→ (3, 7)

Hash Function

hash value를 고유하게 생성하는 방법

- 일반적으로는 hash value를 최대한 고유하게 생성하여 충돌 확률을 낮춘다.
- 만약, hash value 범위가 unsigned long long 범위를 벗어나면 %연산이 들어가므로 고유하지 않음
key값의 모든 정보를 사용하지 않으면 고유하지 않음

1. 개수가 고정되어 있을 때, 수의 범위를 0부터로 맞춰주고 (max값+1) 진법 변환

- -100 ~ 100 범위의 1개 정수
- 0 ~ 100 범위의 4개 정수
- -100 ~ 100 범위의 3개 정수
- 소문자 10자리

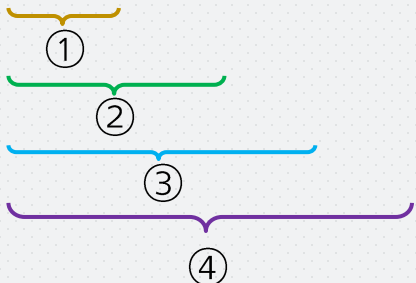
2. 개수가 고정되지 않을 때, 수의 범위를 1부터로 맞춰주고 (max값+1) 진법 변환

- 소문자 4~10자리
- 대소문자 4~10자리

Hash Function 구현

Hornor's method 활용

k 진법 \rightarrow 10진법

$$5k^3 + 3k^2 + 2k^1 + 4k^0$$
$$= (((0*k+5)*k+3)*k+2)*k+4$$


hash=0

- ① hash=hash*k + 5
- ② hash=hash*k + 3
- ③ hash=hash*k + 2
- ④ hash=hash*k + 4

Example code

key값: 0 ~ $k-1$ 범위의 정수 n 개

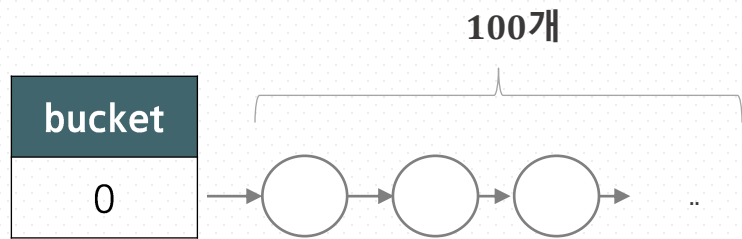
```
ull hashFunc(int key[]) {  
    ull hash = 0;  
    for (int i = 0; i < n; i++)  
        hash = hash * k + key[i];  
    return hash;  
}
```

Hash Table Size

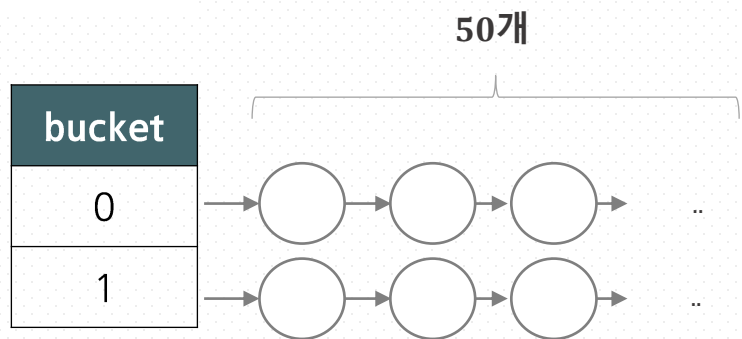
hash value를 최대한 고유하게 설정하였을 때, hash table size가 성능에 미치는 영향

data 수 : 100개

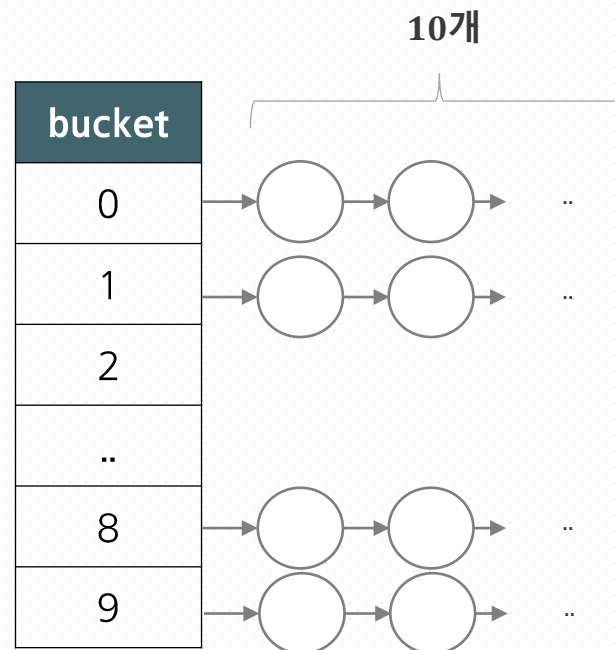
SIZE = 1



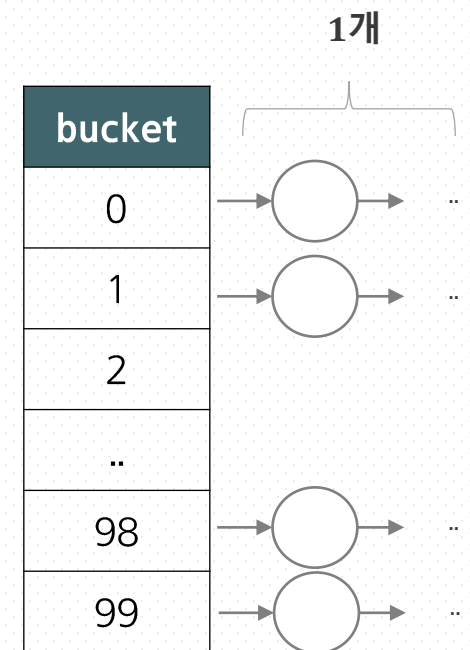
SIZE = 2



SIZE = 10



SIZE = 100



Hash 구현 방식

1. linked list 직접 구현 chaining

2. 일반 배열로 open addressing : `T htab[SIZE];`

3. vector로 chaining : `vector<T> htab[SIZE];` $n / \text{SIZE} = \text{조절}$

4. unordered_map, unordered_set : `unordered_map<T, U> um;`
`unordered_set<T> us;` $n / \text{SIZE} = 1$

STL Container

• Sequence

- array : static array
- vector : dynamic array
- deque : dynamic array
- forward_list : singly linked list
- list : doubly linked list

• Adaptors

- stack : LIFO
- queue : FIFO
- priority_queue : 우선순위 큐

• Associative (Red-Black Tree)

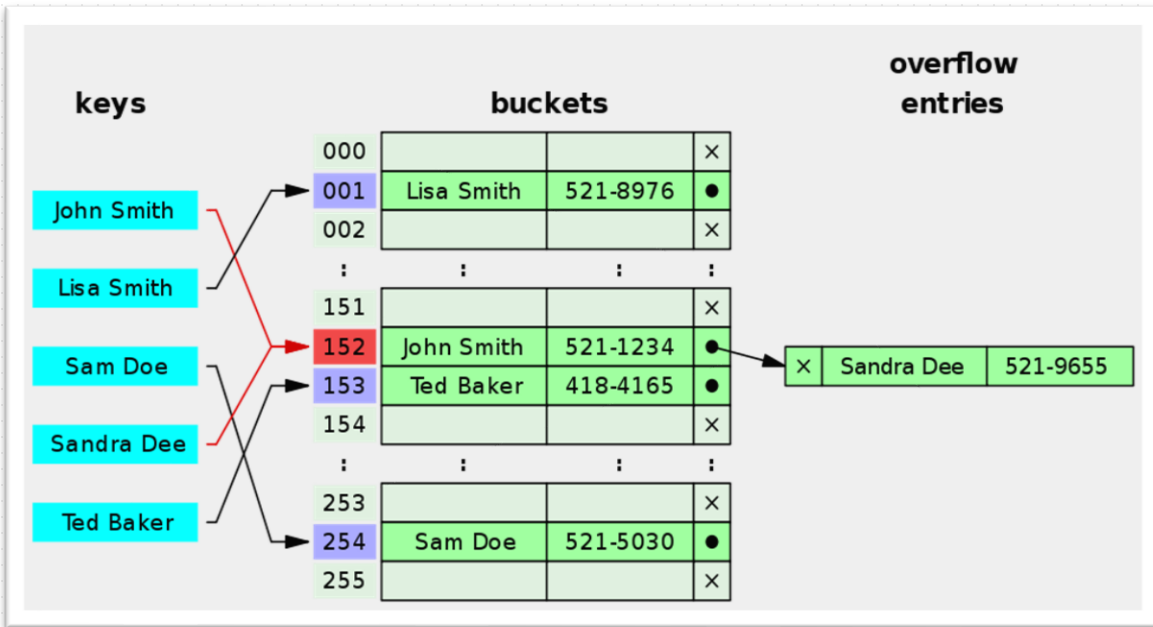
- set : (Key) 중복X
- multiset : (Key) 중복O
- map : (Key,Value), 중복X
- multimap : (Key,Value), 중복O

• Unordered associative (Hash)

- unordered_set : (Key) 중복X
- unordered_multiset : (Key) 중복O
- unordered_map : (Key,Value), 중복X
- unordered_multimap : (Key,Value), 중복O

Unordered Associative

- hash table
- chaining (linked list)
- Forward Iterator



	Average	Worst Case
Space	$O(n)$	$O(n)$
Search	$O(1)$	$O(n)$
Insert	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

key에 따라 bucket에 잘 분배되도록 hash function을 만들어야 한다.

Unordered Associative



: class Hash

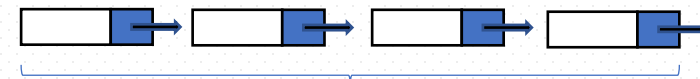
default : hash<Key>

※ specialization 되어 별도로 구현해줄 필요 없는 type

- primitive type : int, long long, char, bool, ...
- string
- pair

Hash Table

index
0
1
2
...
bucket_count-1



bucket_size(2) = 4



probing 기준

: class KeyEqual

default : equal_to<Key>

※ Key type에 operator== 정의 필요

unordered_set

```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_set;
```

unordered_map

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_map;
```


Predefined Functor : comparision operator

- unordered_set, unordered_map에서 default로 equal_to 사용
- data type에 각각 ==가 정의되어 있어야 한다.

1. equal_to

```
template<class T>
struct equal_to {
    bool operator()(const T& lhs, const T& rhs) const {
        return lhs == rhs;
    }
}
```

2. less

```
template<class T>
struct less {
    bool operator()(const T& lhs, const T& rhs) const {
        return lhs < rhs;
    }
}
```

3. greater

```
template<class T>
struct greater {
    bool operator()(const T& lhs, const T& rhs) const {
        return lhs > rhs;
    }
}
```

형태

1. less<T> : class
2. less<T>() : function

```
less<int>()(1,1) = false
less<int>()(1,2) = true
```

3. less<T>{} : function

```
less<int>{}(1,1) = false
less<int>{}(1,2) = true
```

Predefined Functor : hash

- Requirement

- key값에 대해 `size_t` type의 hash 값을 반환한다.
(`size_t == unsigned long long`)
- `k1, k2`가 같다면 `hash<Key>(k1) == hash<Key>(k2)` 이어야 한다.
- `k1, k2`가 다르다면 `hash<Key>(k1) == hash<Key>(k2)` 인 확률이 매우 적어야 한다.

- specialization 되어 있는 type

- 기본 primitive type (`int, long long, char, bool, double, float, ...`)
- `string`
- `pair`

※ specialization : 특정 data type에 대해 별도의 동작을 정의해준다.
즉, 각 type 특성에 맞게 hash function 이 적절히 구현되어 있다.

※ Key값을 받아, hash value를 반환한다.

```
struct hash {  
    size_t operator()(const Key &key) const {  
        return (hash value);  
    }  
};
```

Custom : Hash, KeyEqual

Hash : functor

KeyEqual : equal_to operator(==) overloading

```
struct Key {  
    int x, y;  
    bool operator==(const Key &r) const {  
        return x == r.x && y == r.y;  
    }  
};  
  
struct CustomHash {  
    size_t operator()(const Key &key) const {  
        return key.x * 10000 + key.y;  
    }  
};  
  
unordered_set<Key, CustomHash> us;  
unordered_map<Key, int, CustomHash> um;
```

Custom : Hash, KeyEqual

Hash : functor

KeyEqual : functor

```
struct Key { int x, y; };

struct MyHash {
    size_t operator()(const Key &key) const {
        return key.x * 10000 + key.y;
    }
};

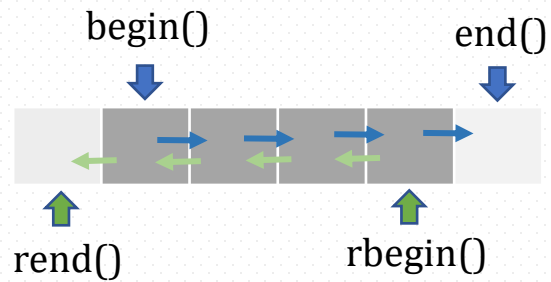
struct MyEqual {
    bool operator()(const Key &l, const Key &r) const {
        return l.x == r.x && l.y == r.y;
    }
};

unordered_set<Data, MyHash, MyEqual> us;
unordered_map<Data, int, MyHash, MyEqual> um;
```

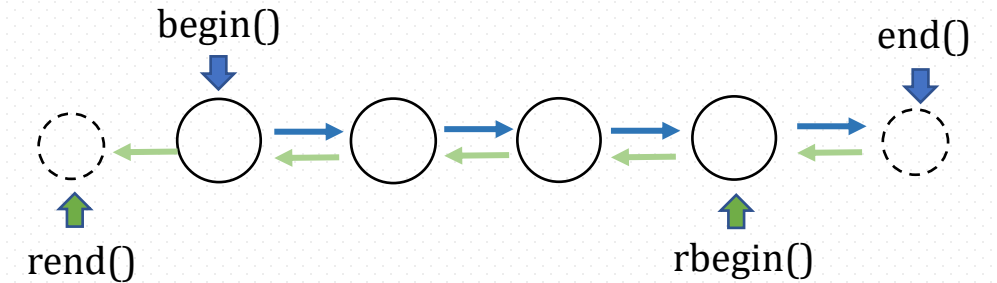
Iterator

→ iterator
← reverse_iterator

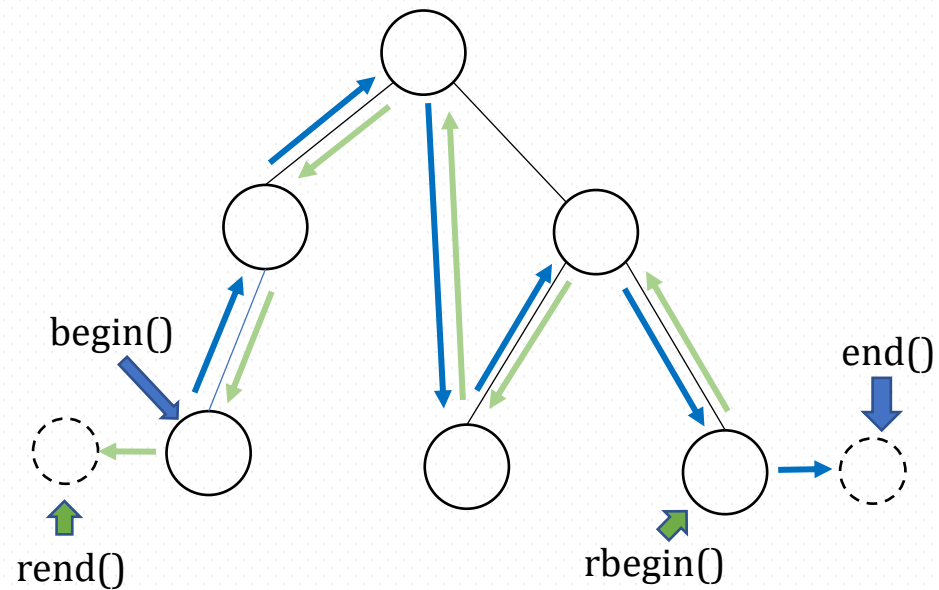
vector



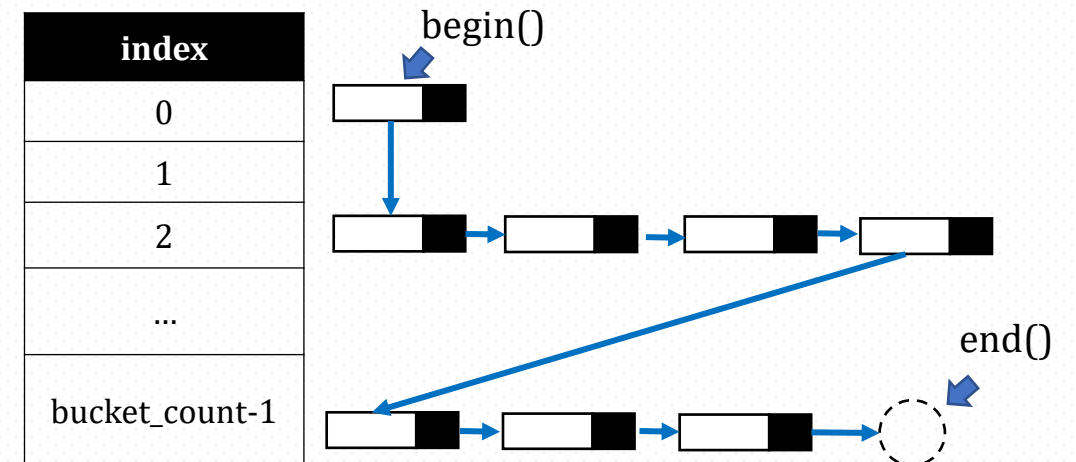
list



set/map



unordered_set/map



Policy

rehash

- bucket_count를 늘려서 hash table을 재구성 하는 과정
- $O(n)$

reserve

- bucket_count 설정
- 설정한 값 이상의 2^k 크기로 설정됨

load_factor

- average number of elements per bucket
- $n / \text{bucket_count}$

max_load_factor

- rehash가 일어나는 threshold
- default 1.0

❖ `htab.reserve(n)` 를 통하여 미리 hash table 크기를 결정하면 불필요한 rehashing이 일어나지 않아 효율적일거 같지만, 실제 test 해보면 별 차이 없고 오히려 느린 경우도 존재하므로 굳이 reserve하지 않고 사용해도 된다.

unordered_set

`unordered_set<T> htab`

`unordered_set<T, Hash> htab`

`unordered_set<T, Hash, KeyEqual> htab`

`iterator htab.begin(), htab.end()`

`bool htab.empty()`

`size_t htab.size()`

`void htab.clear()`

`size_t htab.count(T key)`

`iterator htab.find(T key)`

`void htab.reserve(int n)`

: n 이상의 2^k 로 bucket_count 설정

```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_set;
```

* size_t : unsigned long long

`size_t htab.bucket(T key)` : key 값의 bucket 번호

`size_t htab.bucket_count()` : hash table bucket 수

`size_t htab.bucket_size(int bucket)` : bucket의 element 수

`pair<iterator, bool> htab.insert (T key)` : { 등록된 iterator, 성공 여부}

`void htab.insert(iterator first, iterator last)`

`iterator htab.erase(iterator pos)`

`iterator htab.erase(iterator first, iterator last)`

: 마지막으로 지운 element의 다음 iterator

`size_t htab.erase(T key)` : 성공 여부 0, 1

unordered_set - example

```
unordered_set<int> s;
for (int i = 5; i > 0; i--) s.insert(i);

for (auto x : s) cout << x << ' ';
for (auto it = s.begin(); it != s.end(); ++it) cout << *it << ' ';

s.empty();           // 0
s.size();             // 5
s.count(5);           // 1
s.count(6);           // 0

auto ret = s.find(1);           // 1 검색, ret : 값 1 의 iterator

auto ret = s.insert(6);         // 6 등록, ret.first : 등록된 iterator,   ret.second : 1 (등록 여부)
auto ret = s.insert(1).first;   // 1 등록, ret : 등록된 iterator
auto ret = s.insert(1).second;  // 1 등록, ret : 0 (등록 여부)

auto ret = s.erase(2);          // 2 삭제, ret : 1 (삭제 여부)
auto ret = s.erase(2);          // 2 삭제, ret : 0 (삭제 여부)

auto ret = s.erase(s.begin());  // 맨 앞 element 삭제, ret : 삭제된 element의 다음 iterator
```


unordered_map

```
unordered_map<T, U> htab
```

```
unordered_map<T, U, Hash> htab
```

```
unordered_map<T, U, Hash, KeyEqual> htab
```

```
iterator htab.begin(), htab.end()
```

U htab[T key] :key값이 등록되지 않았다면
 value=default값(0)으로 자동 등록

```
bool htab.empty()
```

```
size_t htab.size()
```

```
void htab.clear()
```

```
size_t htab.count(T key)
```

```
iterator htab.find(T key)
```

```
void htab.reserve(int n)
```

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_map;
```

```
size_t htab.bucket(T key)
```

```
size_t htab.bucket_count()
```

```
size_t htab.bucket_size(int key)
```

pair<T, U> 형태로 등록

```
pair<iterator, bool> htab.insert({ T key, U value })
```

```
void htab.insert(iterator first, iterator last)
```

```
iterator htab.erase(iterator pos)
```

```
iterator htab.erase(iterator first, iterator last)
```

```
size_t htab.erase(T key)
```

unordered_map – example

- key 값에 대한 value 값 추가
- pair<key, value> 형태로 존재

```
unordered_map<int, int> m;

for (int i = 5; i > 0; i--) m.insert({ i, 0 });

// {1,0} {2,0} {3,0} {4,0} {5,0} : 순서 다를 수 있음
for (auto x : m) cout << x.first << ', ' << x.second;
for (auto it = m.begin(); it != m.end(); ++it) cout << it->first << ', ' << it->second;

m[1] = 3; // {1,3} {2,0} {3,0} {4,0} {5,0}
m[6]; // 등록되어 있지 않은 key값이면 value=0으로 자동 insert : {6, 0}
m[7]++; // key값 7이 등록되어 있지 않으므로 value=0으로 insert 후, value 1 증가 : {7, 1}

m.insert({ 1, 4 }); // key값 1은 이미 등록되어 있으므로 무시된다.

m : {1,3} {2,0} {3,0} {4,0} {5,0} {6,0} {7,1}
```

감사합니다

