Chapter 10.
Optimize Data Structures

목차

- 시간측정
- std::map and std::multimap
- std::set and set::mutliset
- std::unordered_map and std::unordered_multimap
- SUMMARY

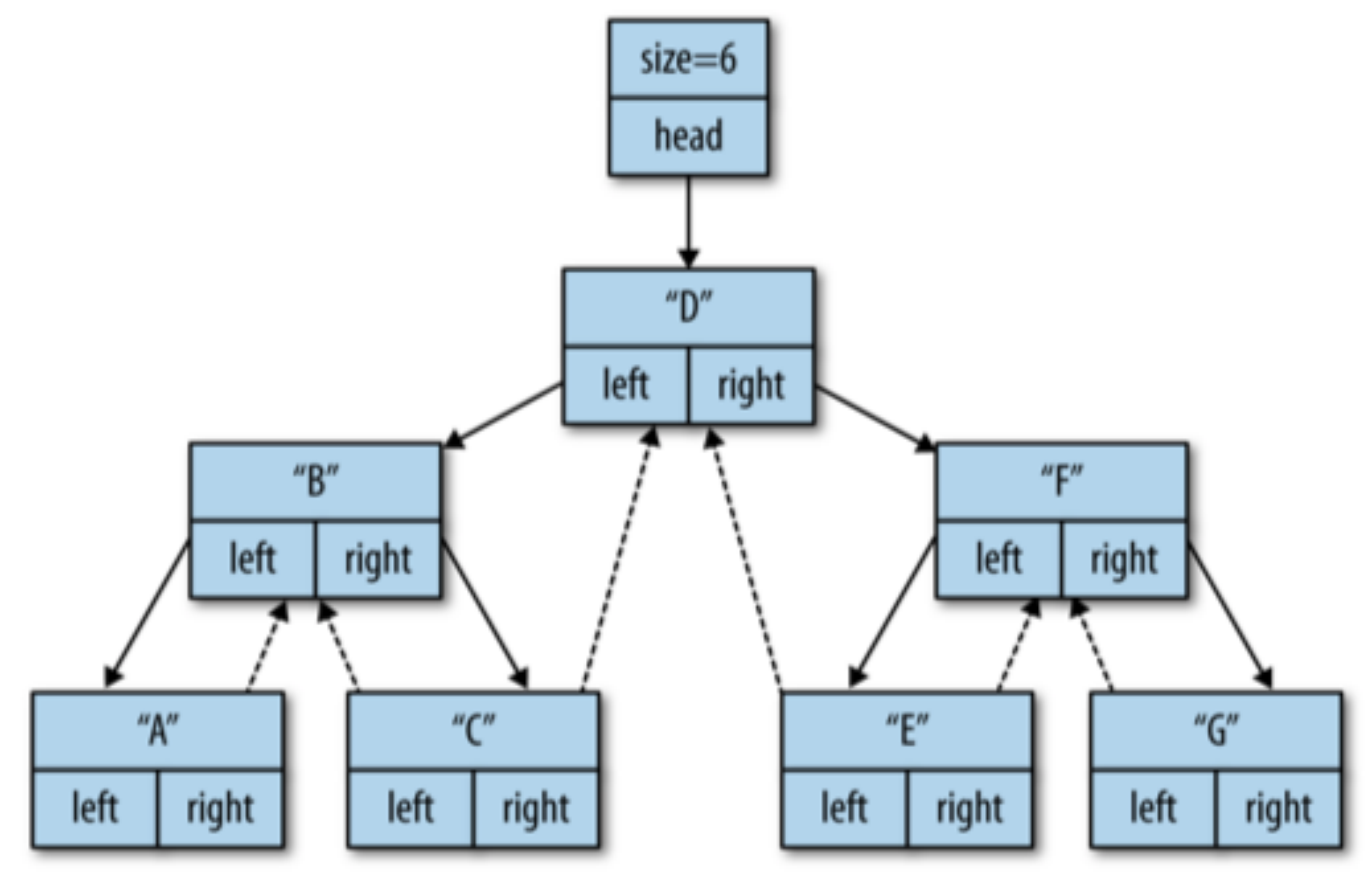
시간 측정

```
typedef std::pair<unsigned, unsigned>  kvType;

typedef void (*testFunc)( void *, std::vector<kvType>& );

void test( const char               * aName,
           void                     * aContainer,
           std::vector<kvType>    & aVector ,
           testFunc                 aFunction )
{
    clock_t sBefore = 0;
    clock_t sAfter = 0;

    sBefore = clock();
    aFunction( aContainer, aVector );
    sAfter = clock();

    printf( "%s : %.8f\n", aName,
            (float)(sAfter - sBefore) / (float)CLOCKS_PER_SEC );

}
```

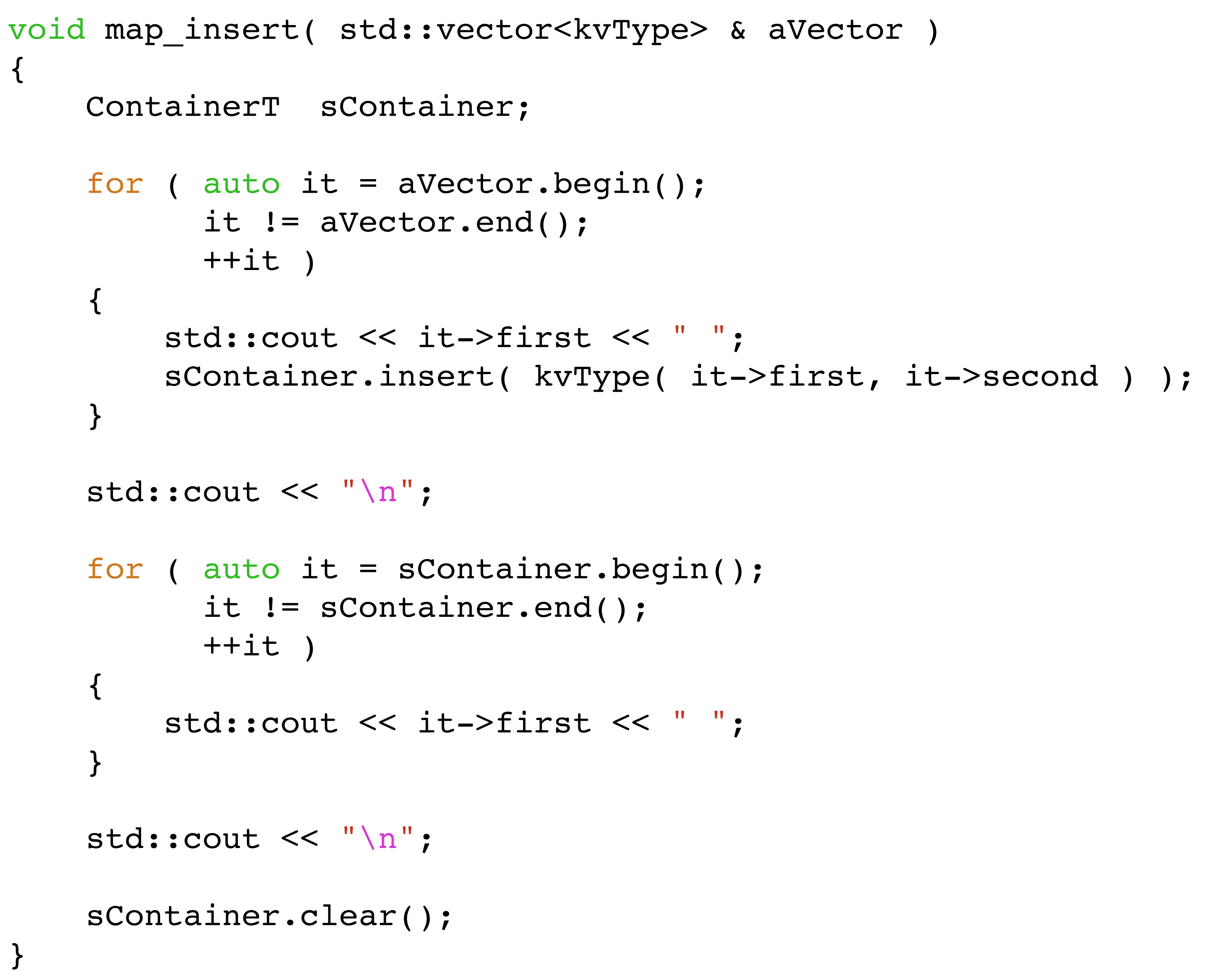
STD::MAP AND STD::MULTIMAP

- Ordered associative container
- Insert Time : O(log n)
- Index time : by key O(log n)
- item 제거시 Iterator 와 reference 는 무효화
- Iterator 는 정렬되거나 역순으로 정렬된 item 을 생성

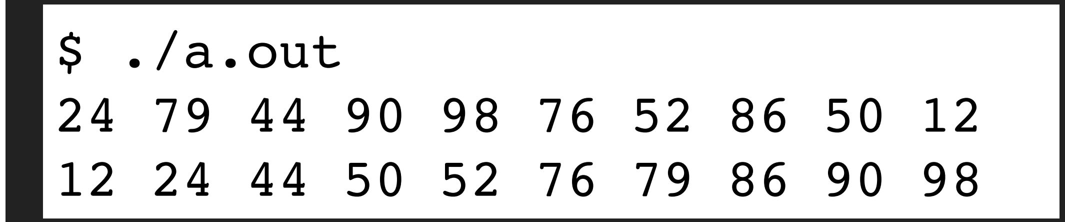
STD::MAP AND STD::MULTIMAP



INSERT IN STD::MAP



```
void map_insert( std::vector<kvType> & aVector )
{
    ContainerT  sContainer;

    for ( auto it = aVector.begin();
          it != aVector.end();
          ++it )
    {
        std::cout << it->first << " ";
        sContainer.insert( kvType( it->first, it->second ) );
    }

    std::cout << "\n";

    for ( auto it = sContainer.begin();
          it != sContainer.end();
          ++it )
    {
        std::cout << it->first << " ";
    }

    std::cout << "\n";

    sContainer.clear();
}
```

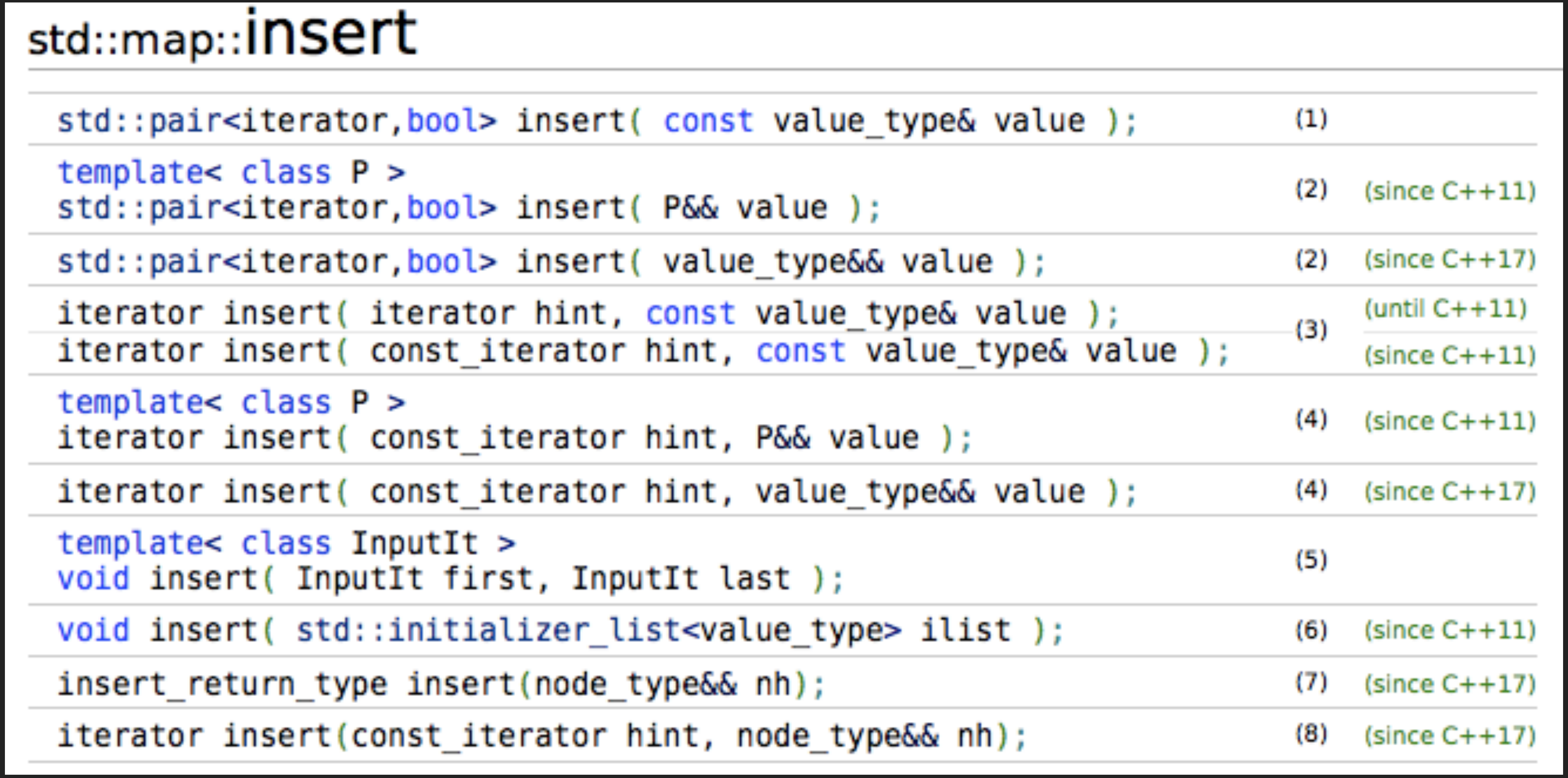


```
$ ./a.out
24 79 44 90 98 76 52 86 50 12
12 24 44 50 52 76 79 86 90 98
```

INSERTING AND DELETING IN STD::MAP

- Insert : O (log n)
  - 내부 트리에 삽입 지점을 찾아야 함
- hint 를 사용하는것이 효율적일수 있음

INSERTING AND DELETING IN STD::MAP



std::map::insert

`std::pair<iterator,bool> insert( const value_type& value );`	(1)	

`template< class P >`
`std::pair<iterator,bool> insert( P&& value );`
	(2)	(since C++11)

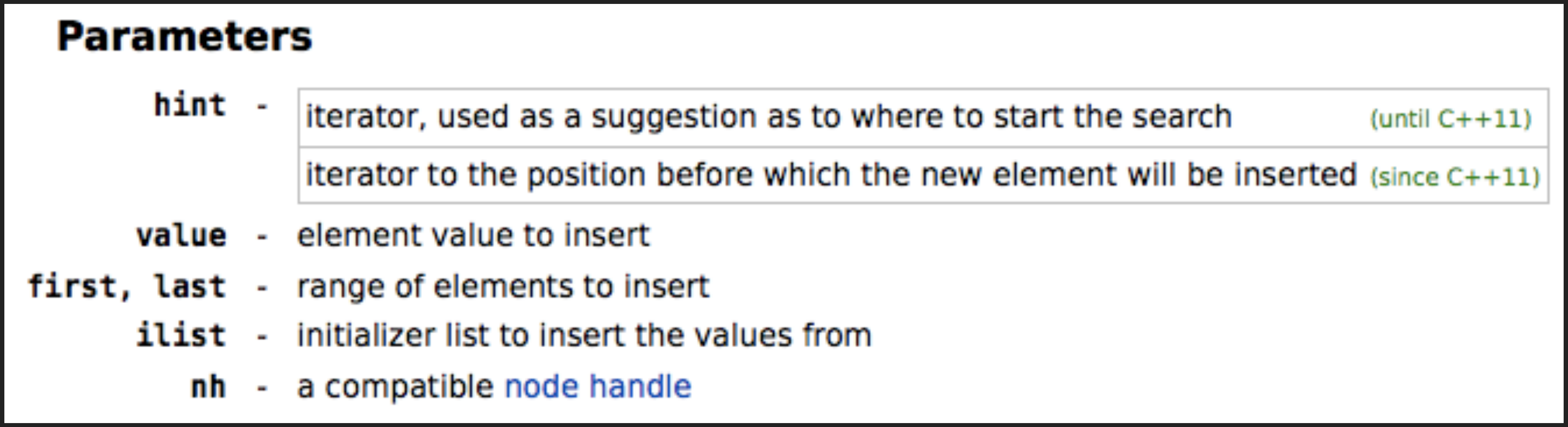
`std::pair<iterator,bool> insert( value_type&& value );`	(2)	(since C++17)
`iterator insert( iterator hint, const value_type& value );`	(3)	(until C++11)
`iterator insert( const_iterator hint, const value_type& value );`		(since C++11)

`template< class P >`
`iterator insert( const_iterator hint, P&& value );`
	(4)	(since C++11)

`iterator insert( const_iterator hint, value_type&& value );`	(4)	(since C++17)

`template< class InputIt >`
`void insert( InputIt first, InputIt last );`
	(5)	

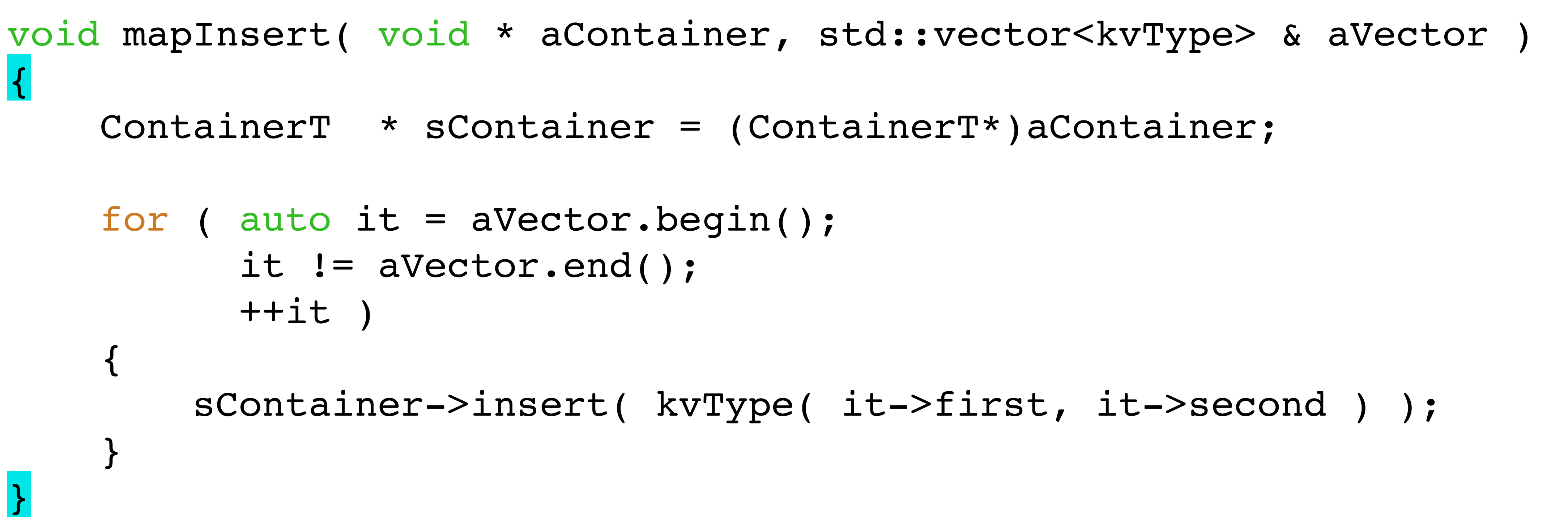
`void insert( std::initializer_list<value_type> ilist );`	(6)	(since C++11)
`insert_return_type insert(node_type&& nh);`	(7)	(since C++17)
`iterator insert(const_iterator hint, node_type&& nh);`	(8)	(since C++17)



Parameters

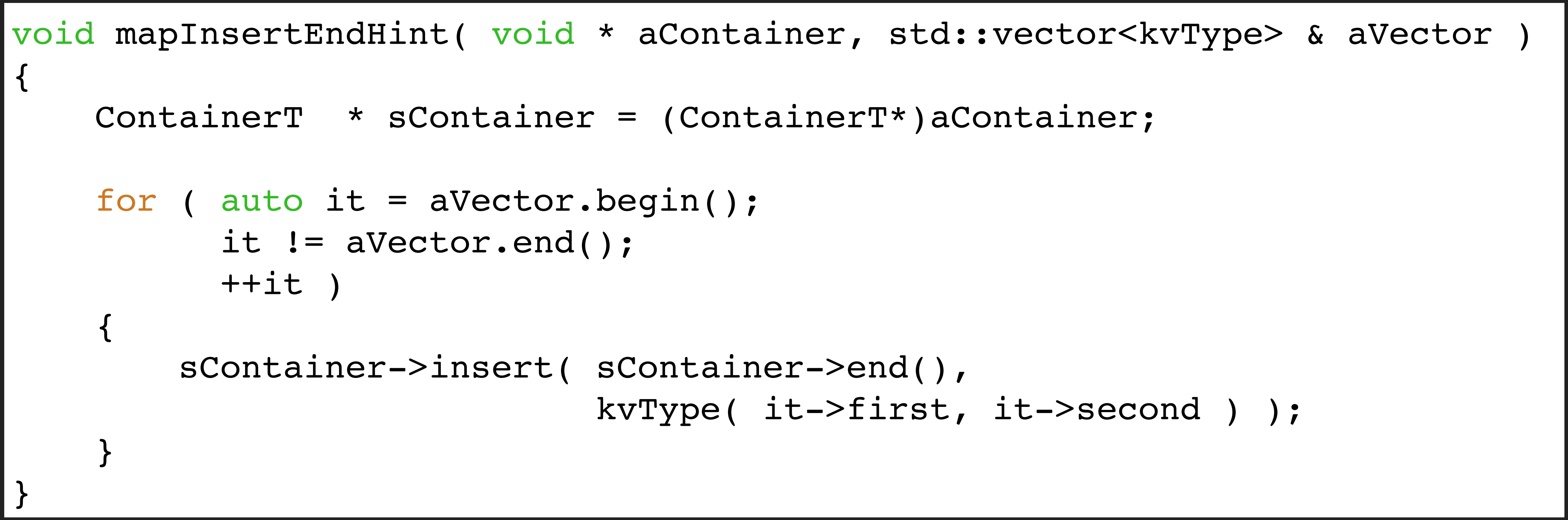
hint	-	iterator, used as a suggestion as to where to start the search	(until C++11)
		iterator to the position before which the new element will be inserted	(since C++11)
value	-	element value to insert	
first, last	-	range of elements to insert	
ilist	-	initializer list to insert the values from	
nh	-	a compatible node handle	

MAPINSERT



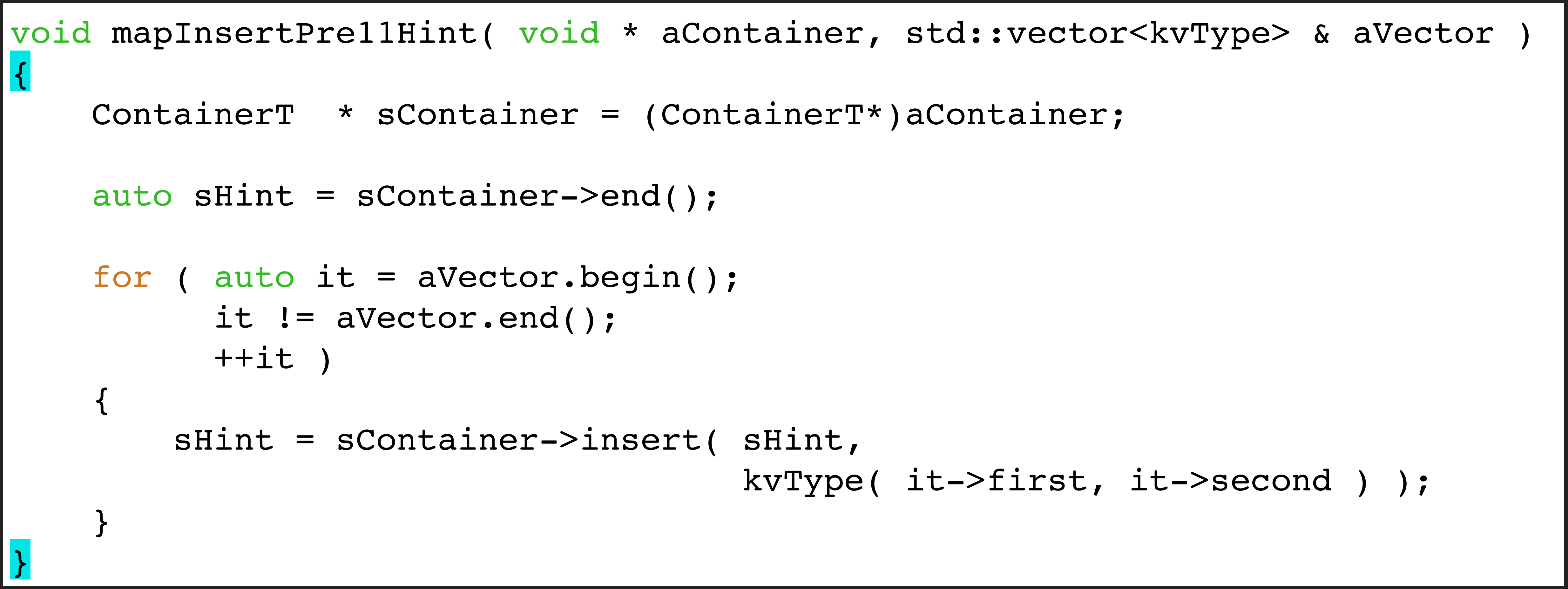
```cpp
void mapInsert( void * aContainer, std::vector<kvType> & aVector )
{
    ContainerT  * sContainer = (ContainerT*)aContainer;

    for ( auto it = aVector.begin();
          it != aVector.end();
          ++it )
    {
        sContainer->insert( kvType( it->first, it->second ) );
    }
}
```

MAP INSERT END HINT



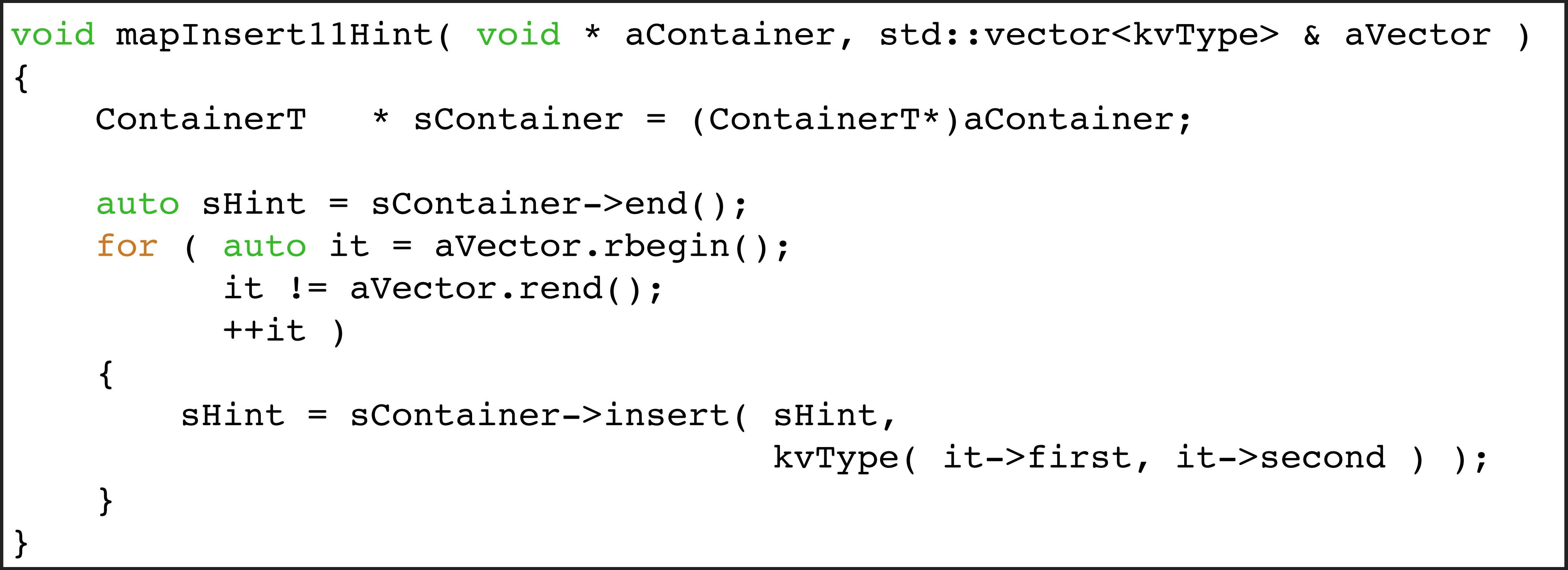
```
void mapInsertEndHint( void * aContainer, std::vector<kvType> & aVector )
{
    ContainerT  * sContainer = (ContainerT*)aContainer;

    for ( auto it = aVector.begin();
          it != aVector.end();
          ++it )
    {
        sContainer->insert( sContainer->end(),
                            kvType( it->first, it->second ) );
    }
}
```

MAP INSERT PRE C++11 HINT



```
void mapInsertPre11Hint( void * aContainer, std::vector<kvType> & aVector )
{
    ContainerT  * sContainer = (ContainerT*)aContainer;

    auto sHint = sContainer->end();

    for ( auto it = aVector.begin();
          it != aVector.end();
          ++it )
    {
        sHint = sContainer->insert( sHint,
                                    kvType( it->first, it->second ) );
    }
}
```

MAP INSERT C++ 11 HINT

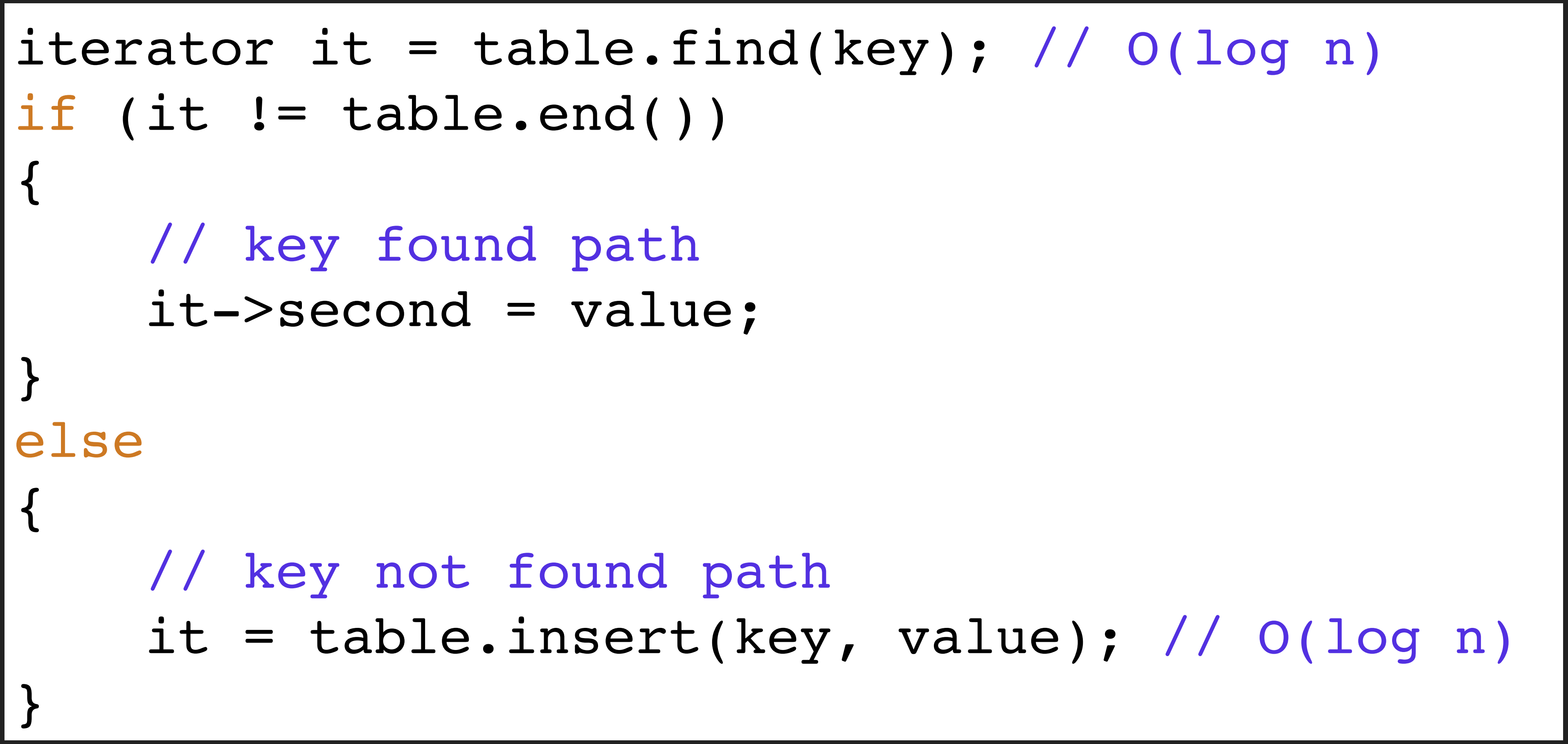


```
void mapInsert11Hint( void * aContainer, std::vector<kvType> & aVector )
{
    ContainerT   * sContainer = (ContainerT*)aContainer;

    auto sHint = sContainer->end();
    for ( auto it = aVector.rbegin();
          it != aVector.rend();
          ++it )
    {
        sHint = sContainer->insert( sHint,
                                    kvType( it->first, it->second ) );
    }
}
```

1000000건 속도 비교

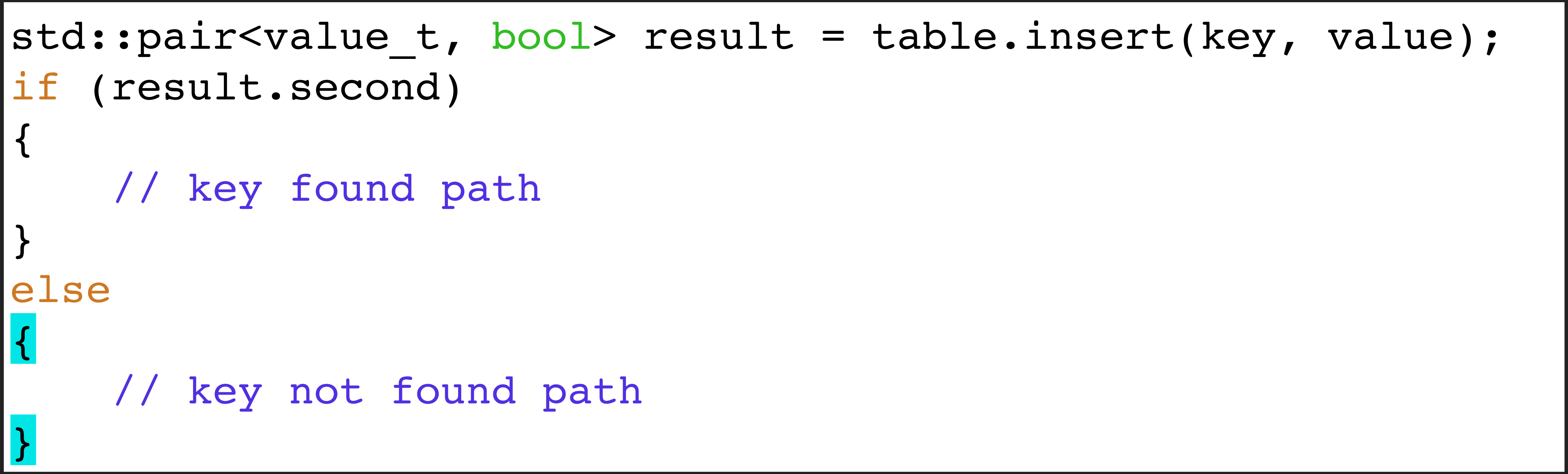
함수	시간	비고
mapInsert	0.7472	
mapInsertEndHint	0.5815	
mapInsertPre11Hint	0.6596	
mapInsert11Hint	0.3989	

OPTIMIZING THE CHECK AND UPDATE IDIOM



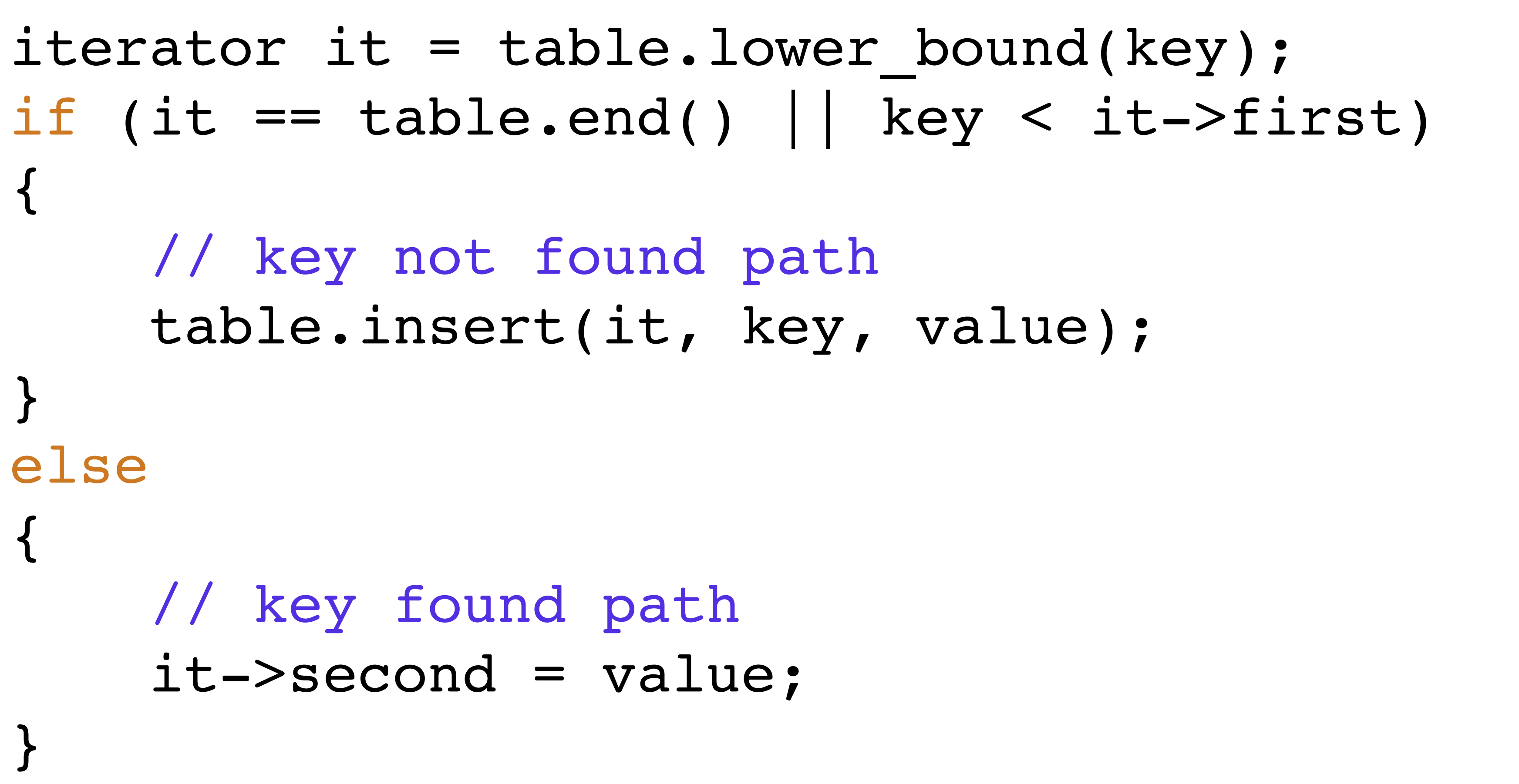
```
iterator it = table.find(key); // O(log n)
if (it != table.end())
{
    // key found path
    it->second = value;
}
else
{
    // key not found path
    it = table.insert(key, value); // O(log n)
}
```

OPTIMIZING THE CHECK AND UPDATE IDIOM



```
std::pair<value_t, bool> result = table.insert(key, value);
if (result.second)
{
    // key found path
}
else
{
    // key not found path
}
```

OPTIMIZING THE CHECK AND UPDATE IDIOM



```
iterator it = table.lower_bound(key);
if (it == table.end() || key < it->first)
{
    // key not found path
    table.insert(it, key, value);
}
else
{
    // key found path
    it->second = value;
}
```

LOOKUP WITH STD::MAP

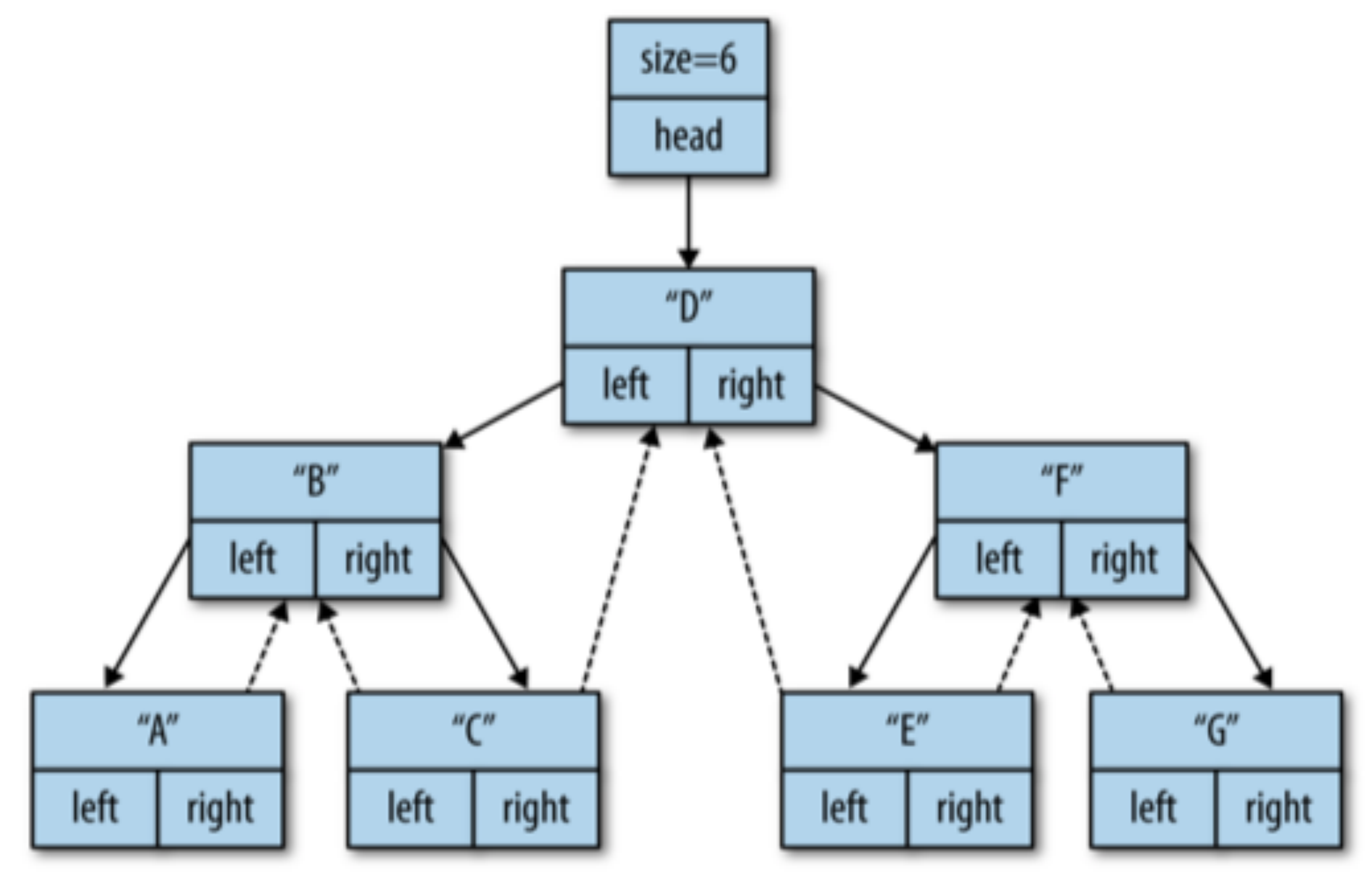
- 1000000 건 속도 비교

	insert + sort	lookup	비고
vector	0.5209	0.9782	
map	1.2877	1.0991	

STD::SET STD::MULTISET

- Ordered associative container
- Insert Time : O(log n)
- Index time : by key O(log n)
- item 제거시 Iterator 와 reference 는 무효화
- Iterator 는 정렬되거나 역순으로 정렬된 item 을 생성

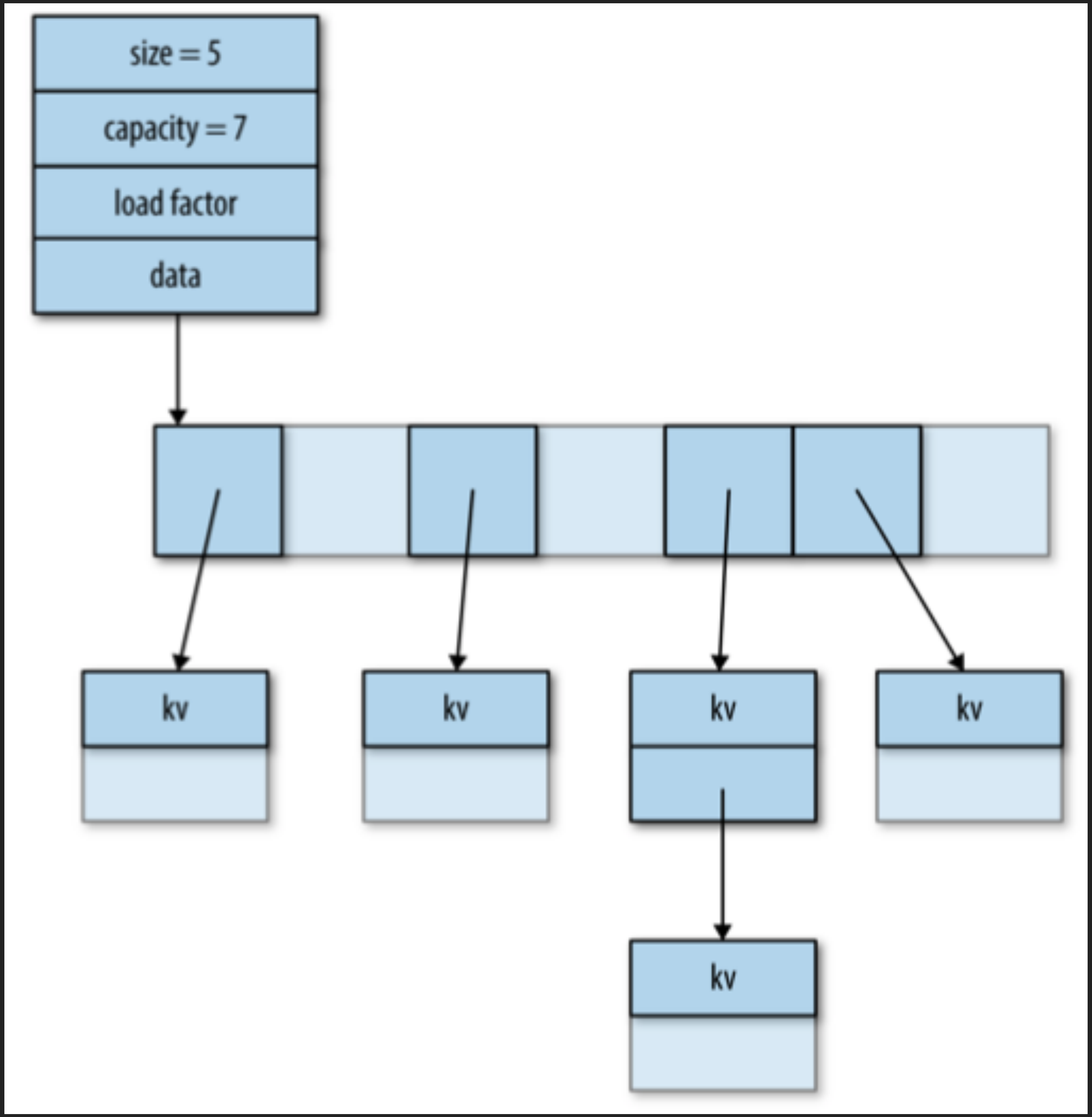
STD::SET STD::MULTISET



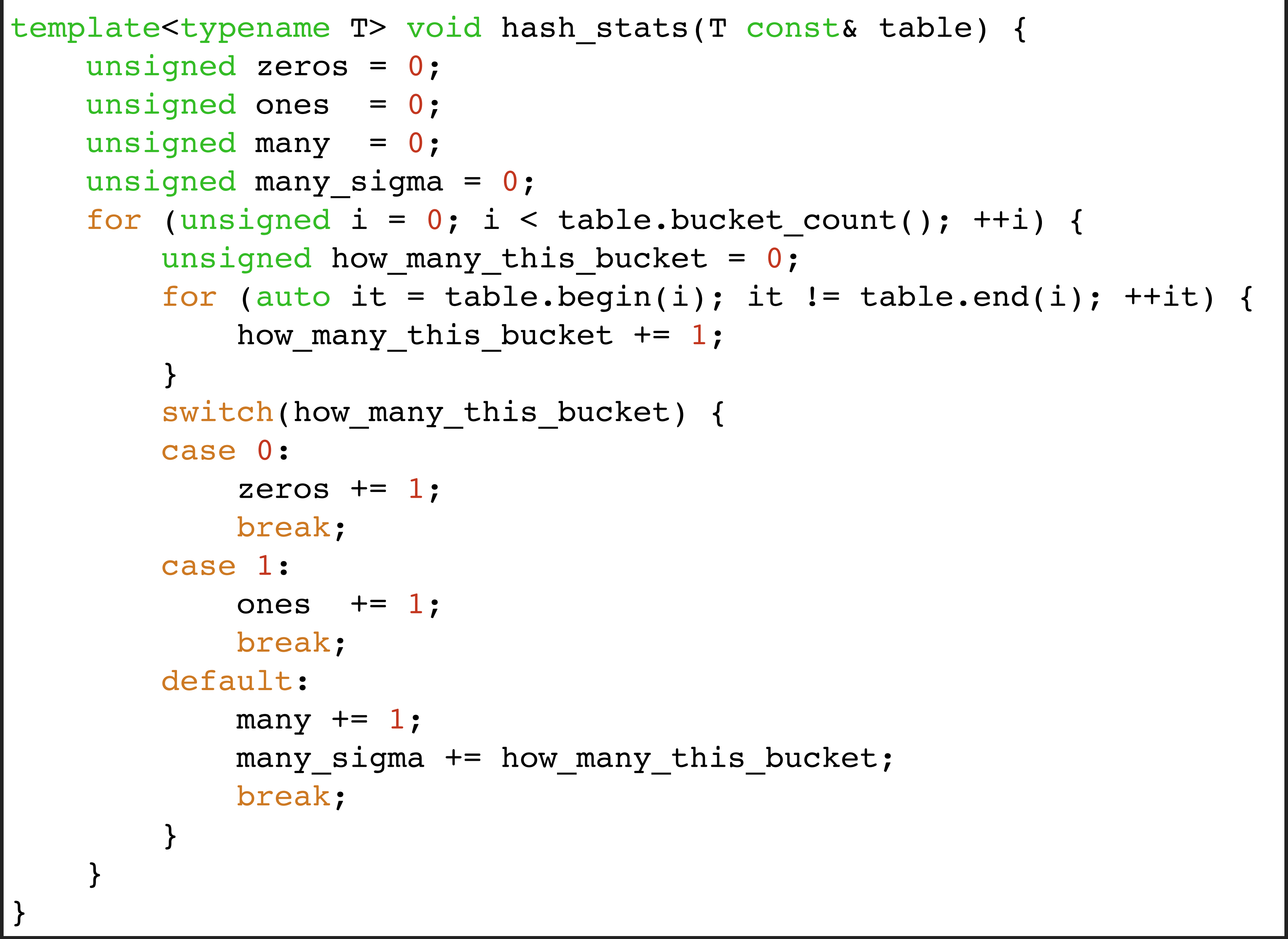
STD::UNORDERED_MAP STD::UNORDERED_MULTIMAP

- Unordered associative container
- Insert time : O (1) average , O(n) worst case
- Index time : by key O (1) average , O(n) worst case
- rehash 시 iterator 는 무효화
- item 제거시 reference 는 무효화
- capacity 는 증가 또는 감소 할수 있음

STD::UNORDERED_MAP STD::UNORDERED_MULTIMAP



SNOOPING ON STD::UNORDERED_MAP

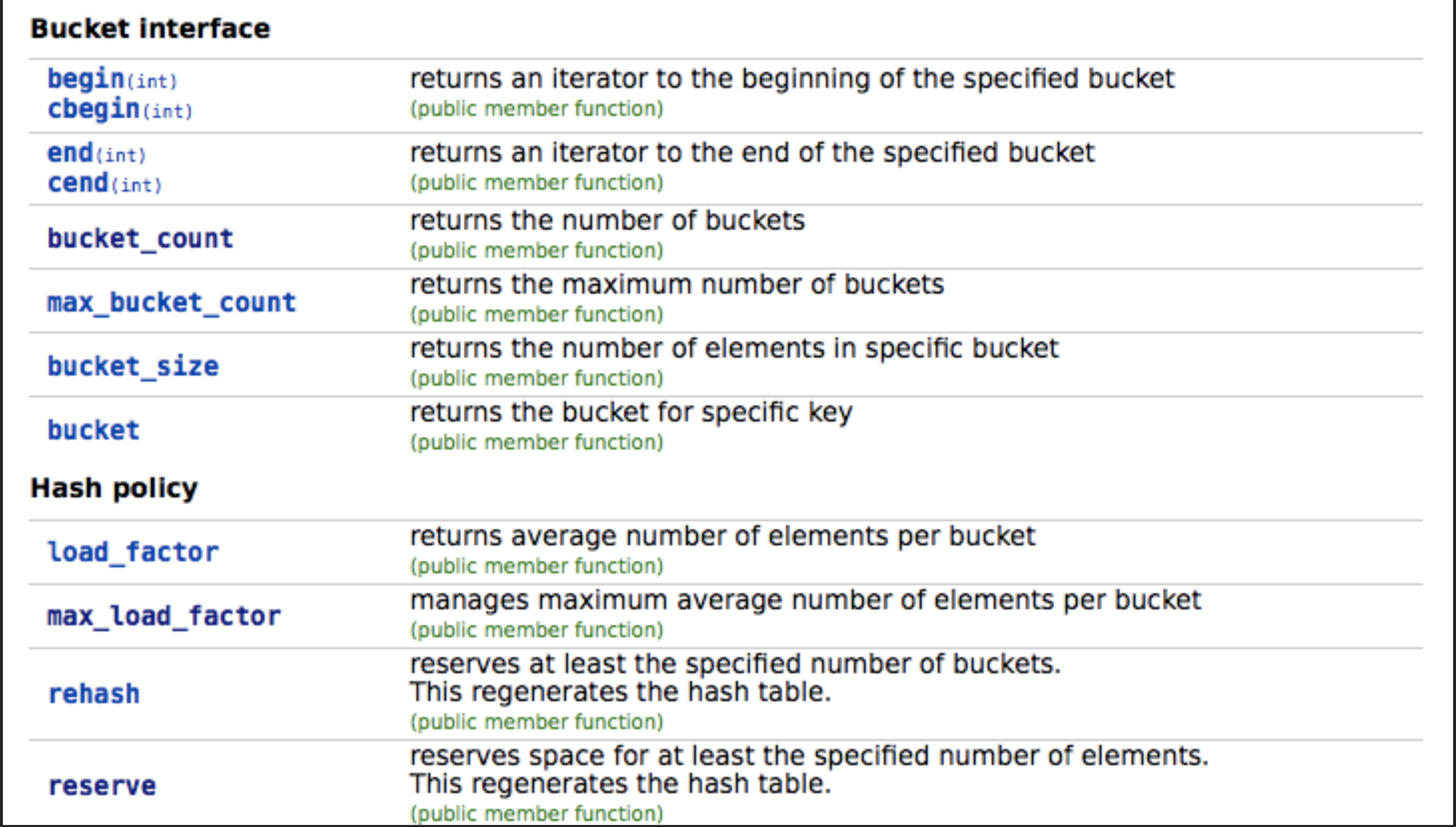


```
template<typename T> void hash_stats(T const& table) {
    unsigned zeros = 0;
    unsigned ones  = 0;
    unsigned many  = 0;
    unsigned many_sigma = 0;
    for (unsigned i = 0; i < table.bucket_count(); ++i) {
        unsigned how_many_this_bucket = 0;
        for (auto it = table.begin(i); it != table.end(i); ++it) {
            how_many_this_bucket += 1;
        }
        switch(how_many_this_bucket) {
        case 0:
            zeros += 1;
            break;
        case 1:
            ones  += 1;
            break;
        default:
            many += 1;
            many_sigma += how_many_this_bucket;
            break;
        }
    }
}
```

INSERTING AND DELETING IN UNORDERED _MAP

- Insertion 성능
  - reserve 함수 호출 rehashing 을 막기 위해 충분한 bucket 를 미리 확보
  - rehashing 은 max_load_factor() * bucket_count()

INSERTING AND DELETING IN UNORDERED _MAP



Bucket interface

begin(int)
cbegin(int)
	returns an iterator to the beginning of the specified bucket
	(public member function)

end(int)
cend(int)
	returns an iterator to the end of the specified bucket
	(public member function)

bucket_count	returns the number of buckets (public member function)
max_bucket_count	returns the maximum number of buckets (public member function)
bucket_size	returns the number of elements in specific bucket (public member function)
bucket	returns the bucket for specific key (public member function)

Hash policy

load_factor	returns average number of elements per bucket (public member function)
max_load_factor	manages maximum average number of elements per bucket (public member function)
rehash	reserves at least the specified number of buckets. This regenerates the hash table. (public member function)
reserve	reserves space for at least the specified number of elements. This regenerates the hash table. (public member function)

INSERTING AND DELETING IN UNORDERED _MAP

reserve	insert	bucket count	load factor	비고
no	0.7503	1646237	0.607446	
500000	0.5485	1000033	0.999996	
1000000	0.505	1000003	0.999997	

LOOKUP WITH STD::UNORDERED_MAP

- 1000000 건 속도 비교

	insert + sort	lookup	비고
vector	0.5209	0.9782	
map	1.2877	1.0991	
unordered_map	0.7421	0.2752	

OTHER DATA STRUCTURE

- boost::circular_buffer
- Boost.Container
- dynamic_bitset
- Fusion
- Boost Graph Library (BGL)
- boot.heap

OTHER DATA STRUCTURE

- Boot.Intrusive
- boost.lockfree
- Boot.MultiIndex

SUMMARY

- Big-O 표기법이 모든걸 이야기해주지 않는다.
- std::vector 가 insert, delete, iterate, sort 에 가장 빠르다.
- Lookup using std::lower_bound in a sorted std::vector can be competitive with std::map
- std::deque 는 std::list 보다 약간 빠르다.
- std::forward_list 는 std::list 보다 빠르지 않다.

SUMMARY

- std::map 보다 hash table std::unordered_map 이 빠르다. 그러나 생각처럼 많이 빠른것은 아니다.
- 인터넷은 standard library container 를 모방한 좋은 소스의 container 들이 있다.