

Container

TOGATER

what is Container?

- 컨테이너(Container)란?



What is Container?

기본 자료형과 사용자 정의 자료형을
담는 일종의 **자료구조**

Types of Containers

- 시퀀스 컨테이너 (Sequence Container)
- 연관 컨테이너 (Associative Container)
- 컨테이너 어댑터 (Container Adapter)

Features of Containers

컨테이너에 삽입이 이루어질 경우, 내부적으로 **복사본을 생성**한다. 즉, 모든 STL 컨테이너의 원소들은 복사될 수 있어야 한다.

Features of Containers

- 컨테이너의_특징[1]

- 모든 원소들은 순서를 가지고 있다.
- 각각의 컨테이너는 자신의 원소를 순회(traverse)할 수 있도록 자신만의 반복자(iterator)를 제공한다.

Features of Containers

- 컨테이너의_특징[2]


- 대부분의 경우, STL 자체는 **예외를 발생시키지 않는다.**
(내부적으로 예외처리를 하지 않았음)
- 그렇기 때문에, 각각의 동작에 대한 인자를 제공할 때,
올바른 인자를 보장해야만 한다.

Iterator

TOGATER

what is Iterator?

영어사전 단어·속어 1-1 / 1건

iterate 미국·영국 [ˈɪtəreɪt] 

(계산·컴퓨터 처리 절차를) 반복하다

영어사전 단어·속어 1-5 / 67건

iteration 미국·영국 [ˌɪtəˈreɪʃn] 

1. (계산·컴퓨터 처리 절차의) 반복
2. (컴퓨터 소프트웨어의) 신판(新版)

What is Iterator?

- 반복(iteration)을 수행하지는 않는다.

(An iterator performs traversal and also gives access to data elements in a container, but **does not perform**

iteration.) 출처 - 영문 Wikipedia의 Iterator 항목

what is Iterator?

컨테이너의 원소들을 **순회**(traverse)할 수 있는 **객체**

Types of Iterators

- 반복자의_종류
 - iterator
 - const_iterator

Features of Iterator

- 반복자의_특징[0]

- 반복자는 컨테이너의 특정 위치를 가리킨다.

Features of Iterator

- 반복자의_특징[1]

- 포인터와 비슷!

- `*iter, iter++, iter--, iter_1 != iter_2`

- 이러한 연산자들을 사용할 수 있다

- 복잡한 컨테이너들을 순회할 수 있는 스마트포인터

How to use Iterator?

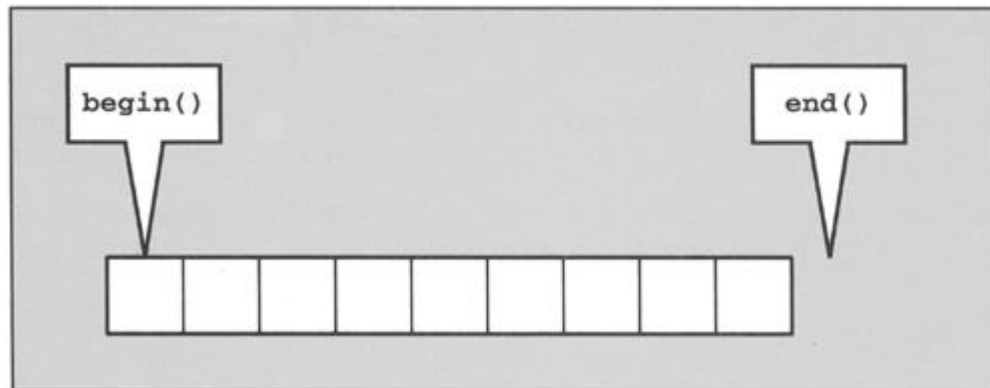
- 반복자의_사용법[0]

- `container<typename>::iterator iter;`
 - `vector<int>::iterator iter_1;`
 - `map<int, char*>::iterator iter_2;`

How to use Iterator?

- 반복자의_사용법[1]

- `vector<int> oh_my_vector;`
- `vector<int>::iterator iter = oh_my_vector.begin();`

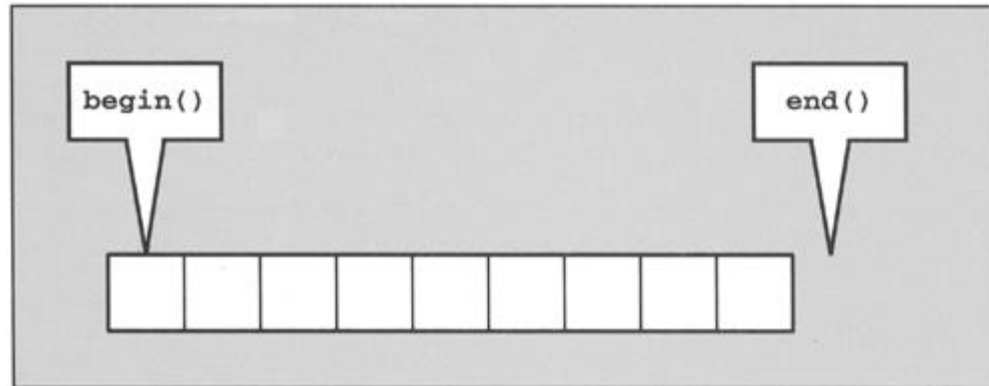


How to use Iterator?

• 반복자의_사용법[1]

와 직관적이다

- `for(iter = v.begin(); iter != v.end(); ++iter) { ... }`



How to use Iterator?

• 반복자의_사용법[2]

- 객체를 담는 컨테이너의 경우, 객체 내부의 멤버에도 접근할 수 있다.

- `iter->oh_my_member`
- `(*iter).oh_my_member`

```
class my_class
{
public:
    int oh_my_member;
};
```

How to use Iterator?

- 반복자의_사용법[3]

- `std::vector<std::vector<int>> what_the_fector;` (2차원 벡터)
- `std::vector<std::vector<int>>::iterator iter = what_the_fector.begin();`

How to use Iterator?

- 반복자의_사용법[3]

- `std::vector<std::vector<int>> what_the_fector;` (2차원 벡터)
- `auto iter = what_the_fector.begin();`

Sequence Container

Sequence Container



임의의 타입의 동일한 객체 집합을
선형으로 구성한 컬렉션

특별한 삽입과 삭제의 규칙이 존재하지 않는 컨테이너이다.

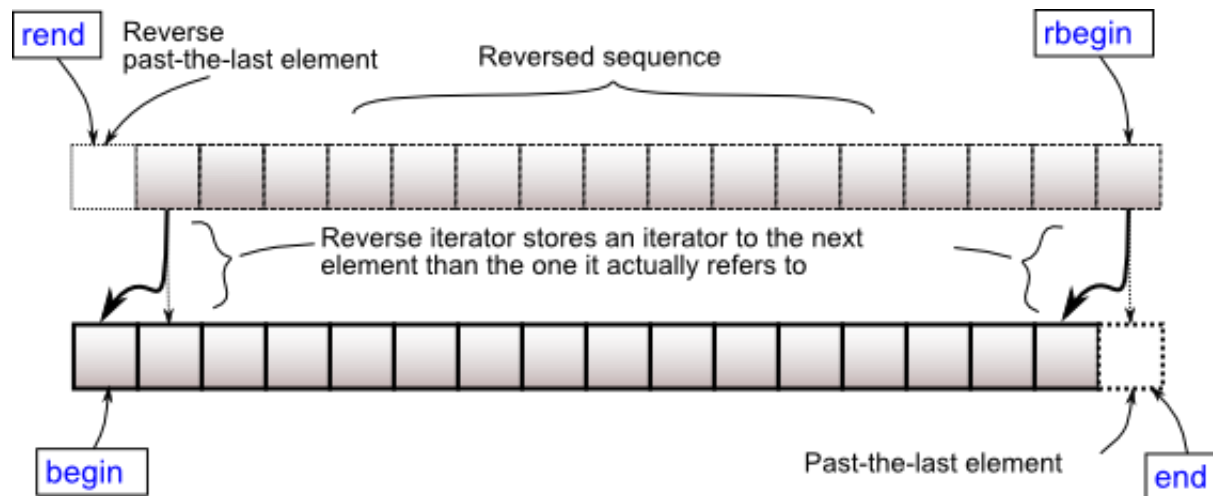
Sequence Container In C++

<code>std::array</code>	<code>(boost -> c++ 11)</code>
<code>std::vector</code>	<code>(1988)</code>
<code>std::deque</code>	<code>(1988)</code>
<code>std::list</code>	<code>(1988)</code>
<code>std::forward_list</code>	<code>(c++ 11)</code>

Sequence Container In C++

컨테이너 공통 특이 사항

- 멤버 함수에 `c`가 붙으면 상수 함수 ex) `cbegin()`, `begin()`
- 멤버 함수에 `r`이 붙으면 반대 개념 ex) `rbegin()`, `rend()`
- `capacity()` vs `size()` vs `max_size()`
- `empty()` vs `size() == 0`
- 모든 컨테이너는 "레퍼런스 의미론"보다는 "값 의미론"을 제공한다.
`auto_ptr`, `pointer` -> 문제 발생 가능



std::array in C++

구현 특징

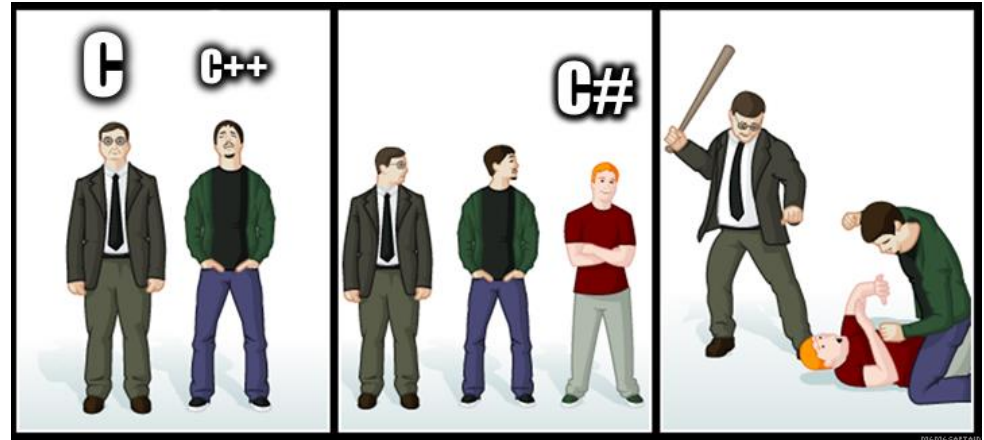
- array == 일반적인 배열 == compile_time non-resizeable array
- allocator를 지원하지 않는다. == stack에 저장된다.

왜 쓰지?

- C++ 배열에 없는 다양한 인터페이스 제공. = 편함
- STL의 새로운 문법과 결합도가 좋음. = 편함
- 임의 요소 접근이 빠름

array in C++

```
int iarray[23] = { 5, 3, 2, };  
iarray.
```



array in C#

```
Int32[] array = new Int32[] { 5, 3, 2, };
```

array.Length

- GetLength
- GetLongLength
- Length**
- LongLength

```
int Array.Length { get; }
```

모든 자원의 Array에 있는 요소의 총 수를 나타내는 32비트 정수를 가져옵니다.

std::array in C++

```
std::string seasons[4] = { "spring", "summer", "autumn", "winter" };

std::array<std::string, 10> testArray;

for (int i = 0; i < 4; ++i)
    testArray[i] = seasons[i];

std::cout << "testArray size = " << testArray.size() << std::endl;

for (auto iter = testArray.begin(); iter != testArray.end(); ++iter)
    std::cout << *iter << std::endl;
```

std::vector in C++

구현 특징

- vector == 연속된 메모리 공간 == array
- dynamically allocated array
- allocator를 통해 heap에 할당

장점

- 임의 접근이 빠르다. $O(1)$
- low memory usage, locality of reference, data cache utilization.

단점

- 삽입&삭제가 느리다. (뒤에서는 빠름)
- 주의 사항이 있다.

std::vector in C++

일반적인 사용법

```
std::vector<std::string> my_vector = {"hello", "hi", "welcome"};
```

```
for (auto iter = my_vector.begin(); iter != my_vector.end(); iter++)  
    std::cout << *iter << std::endl;
```

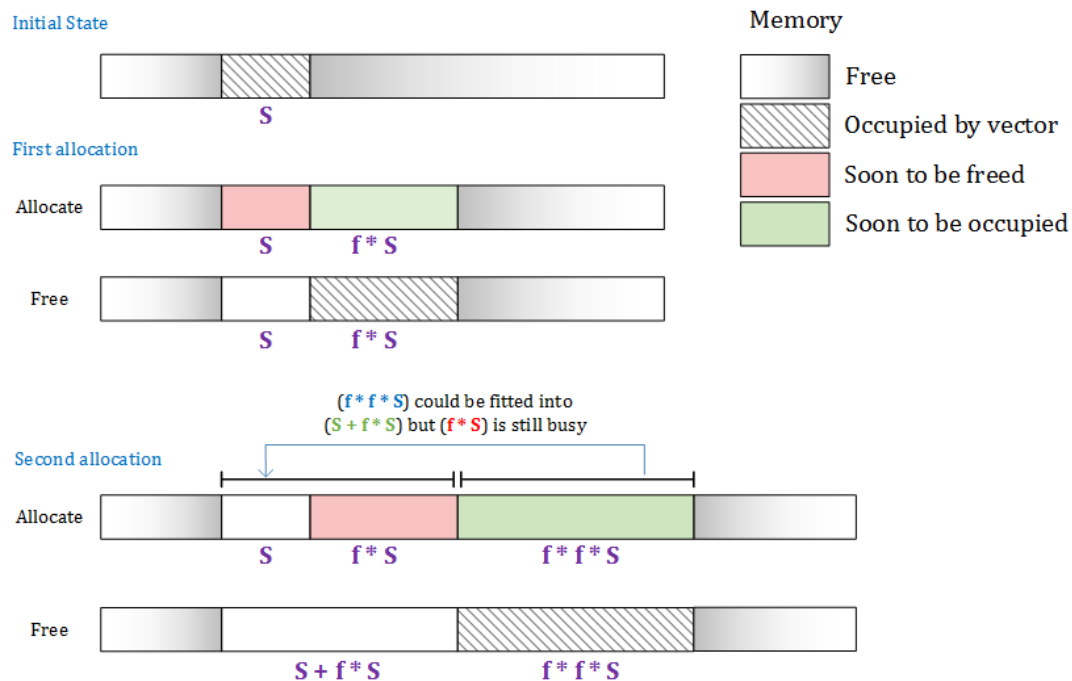
```
std::cout << my_vector[1] << std::endl;
```

std::vector in C++

주의 사항 1. reallocation

Vector는 새로운 공간이 필요할 때, 기존 메모리와 필요한 메모리의 전체 크기만큼 메모리 할당을 하고 복사한 후에 기존 배열을 제거한다.

=> 이 순간 벡터 내부 요소를 가리키는 reference나 iterator는 무효화 된다.



std::vector in C++

```
std::vector<int> my_vector;
```

```
my_vector.push_back(1);  
my_vector.push_back(2);  
my_vector.push_back(3);  
my_vector.push_back(4);
```

```
my_vector.resize(10);
```

```
// result  
// max_size = 1073741823  
// size = 10, capacity = 10
```

```
std::vector<int> my_vector;
```

```
my_vector.push_back(1);  
my_vector.push_back(2);  
my_vector.push_back(3);  
my_vector.push_back(4);
```

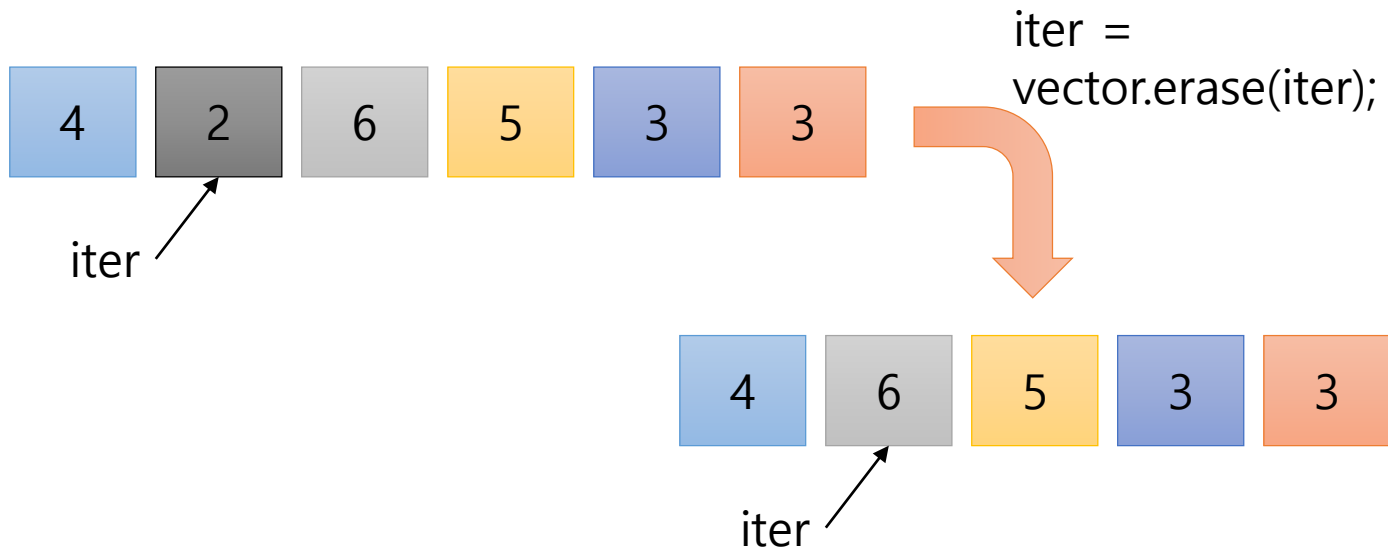
```
my_vector.reserve(10);
```

```
// result  
// max_size = 1073741823  
// size = 4, capacity = 10
```

std::vector in C++

주의 사항 2. 요소 삭제

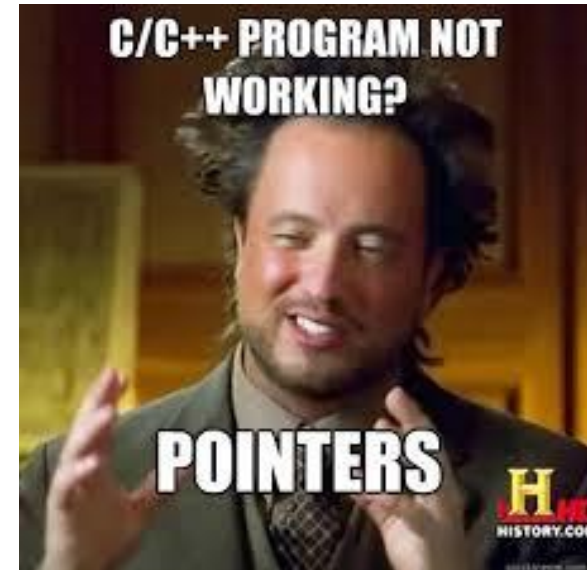
```
for (auto iter = my_vector.begin(); iter != my_vector.end(); )  
{  
    delete (*iter);  
    iter = my_vector.erase(iter);  
}
```



std::vector in C++

주의 사항 3. insert

Vector는 중간에 insert가 되면 해당 위치의 iterator는 무효화된다.



```
std::vector<std::string> my_vector = {"hello", "hi", "ImYourFather", "welcome"};

for (auto iter = my_vector.begin(); iter != my_vector.end(); iter++)
{
    if (*iter == "hi") // error!!!!!!!!!!!!
        my_vector.insert(my_vector.begin(), "temp");

    std::cout << (*iter) << std::endl;
}
```

std::vector in C++

C++ 11 : vector tip

Vector에 값 저장 시 value 타입으로 저장할 경우,

-> push_back vs emplace_back

push_back : 값을 element로 넣을 때, element와 대상 간의 복사생성자를 이용한 값 복사가 일어남

emplace_back : 값을 element로 넣을 때, 생성자에 필요한 인자들을 넣어서 생성함으로 값 복사가 일어나지 않음

```
vec_inst.emplace_back(param_1, param_2, param_3);
```

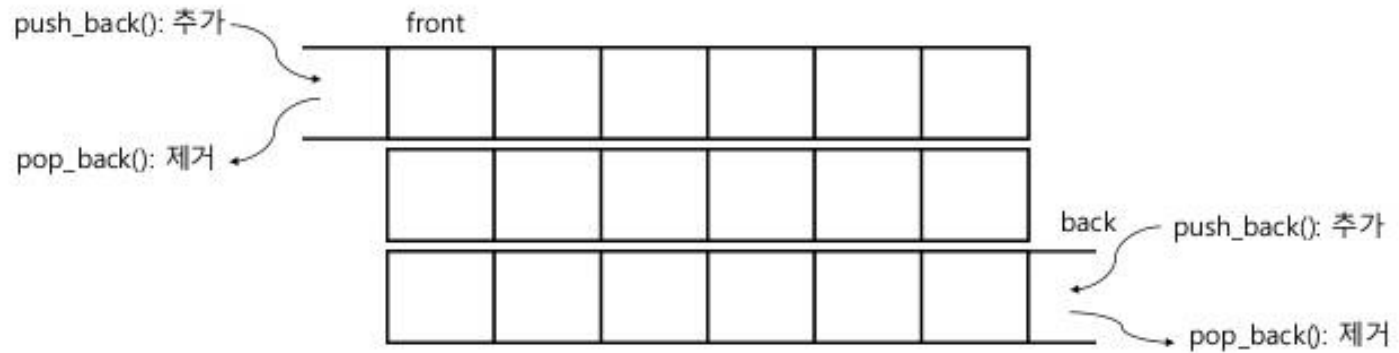
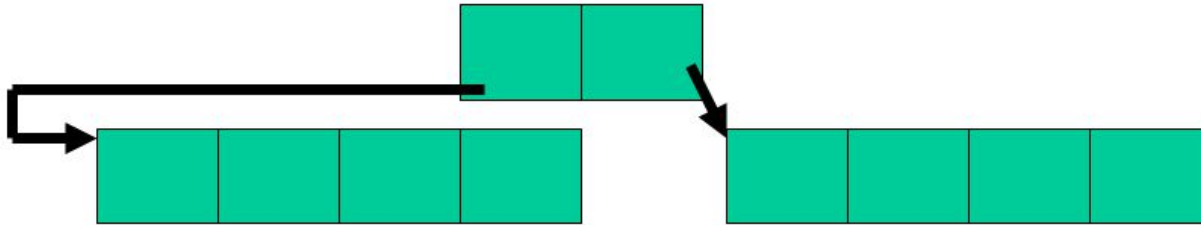
```
vec_inst.push_back(elem_class(param_1, param_2, param_3));
```

std::vector in C++

std::vector를 사용해야 하는 경우

저장할 데이터의 개수가 가변적일 때
중간에 데이터 삽입, 삭제가 적거나 없을 때
저장된 요소를 자주 검색하지 않을 때
임의 접근을 자주 할 때

std::deque in C++



std::deque in C++

구현 특징

- std::deque == double_ended queue
- 연속적이지 않은 메모리 공간 사용

장점

- 임의 접근이 가능하다.
- Vector와 비슷한 계산 복잡도를 가진다.
- 양 끝 단 삽입&삭제가 빠르다.

단점

- std::deque의 앞, 뒤 삽입, 삭제 성능을 제외한 다른 위치에서의 삽입, 삭제는 std::vector가 더 좋다.

std::deque in C++

	deque	vector
크기 변경 가능	O	O
앞에 삽입, 삭제 용이	O	X
뒤에 삽입, 삭제 용이	O	O
중간 삽입, 삭제 용이	X	X
순차 접근 가능	O	O
랜덤 접근 가능	O	X

std::deque in C++

일반적인 사용법

```
std::deque<int> dq;

dq.push_back(10);
dq.push_back(20);
dq.push_back(30);

dq.push_front(100);
dq.push_front(200);

for (deque<int>::size_type i = 0; i < dq.size(); ++i)
    cout << dq[i] << ' ';
cout << endl;

for (auto iter = dq.begin(); iter != dq.end(); ++iter)
    cout << *iter << ' ';
cout << endl;

auto iter = dq.begin() + 2;

cout << *dq.insert(iter, 500) << endl;

for (auto iter = dq.begin(); iter != dq.end(); ++iter)
    cout << *iter << ' ';
```

std::deque in C++

※ **std::deque**를 사용해야 하는 경우
앞과 뒤에서 삽입, 삭제를 자주 할 때
저장할 데이터의 개수가 가변적 일 때
데이터 검색을 거의 하지 않을 때
임의 접근을 해야 할 때
서버처럼 받은 패킷을 차례대로 처리할 때

std::list in C++

A Doubly-Linked List



std::list in C++

구현 특징

- std::list == double_linked list
- 배열이 아닌 노드 형식으로 구현

장점

- 삽입&삭제가 매우 빠르다. $O(1)$

단점

- std::vector보다 메모리 사용량이 크다.
(node 크기 : $\text{sizeof}(\text{type}) + 2 * \text{sizeof}(\text{type}^*)$)
- 순회 속도 시간이 느리다. $O(n)$
- 임의 접근을 지원하지 않는다

std::list in C++

그림 4-2 배열에서 데이터 삽입하기

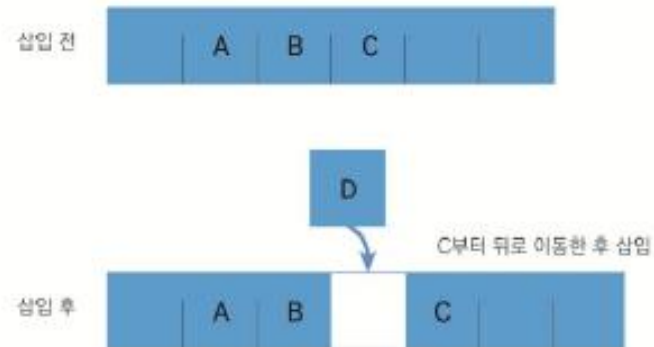


그림 4-3 연결 리스트에서 데이터 삽입하기

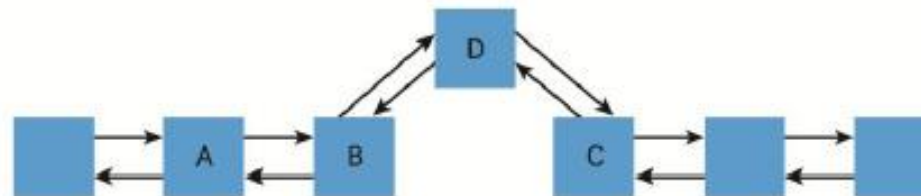
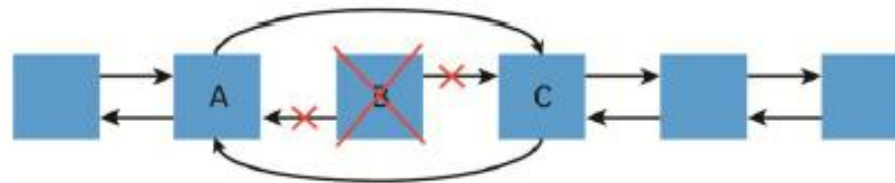


그림 4-4 리스트에서 데이터 삭제하기



std::list in C++

일반적인 사용법

```
std::list<int> my_list;  
std::list<int> my_list2 = {6, 3, 5};  
  
my_list.push_back(3);  
my_list.push_front(6);  
  
my_list.insert(my_list.begin(), 4);  
  
for (auto iter = my_list.begin(); iter != my_list.end(); iter++)  
    std::cout << *iter << std::endl;  
  
my_list.pop_back();  
my_list.pop_front();  
my_list.emplace_back(3);  
  
my_list.sort();  
my_list2.sort();  
  
my_list.merge(my_list2);  
  
for (auto iter = my_list.begin(); iter != my_list.end(); iter++)  
    std::cout << *iter << std::endl;
```

std::list in C++

※ **std::list**를 사용해야 하는 경우

저장할 데이터의 개수가 가변적일 때

저장된 요소를 자주 검색하지 않을 때

중간에 데이터 삽입, 삭제가 자주 발생할 때

랜덤 액세스를 자주 안할 때

std::forward_list in C++

구현 특징

- std::list == single_linked list
- 연속적이지 않은 메모리 공간 사용

장점

- std::list 보다 메모리 사용량이 적고 빠르다.
- 삽입&삭제가 매우 빠르다. $O(1)$

단점

- std::list와 동일한 단점을 지닌다.
- size 함수를 지원하지 않는다.
- 삽입, 삭제는 다음 요소에 대해서만 가능하다.
(insert, erase 제공 x)

std::forward_list in C++

※ **std::forward_list**를 사용해야 하는 경우
double linked list를 사용해야 하는데
single linked list로 충분할 때

Container Adaptor



TOGATER

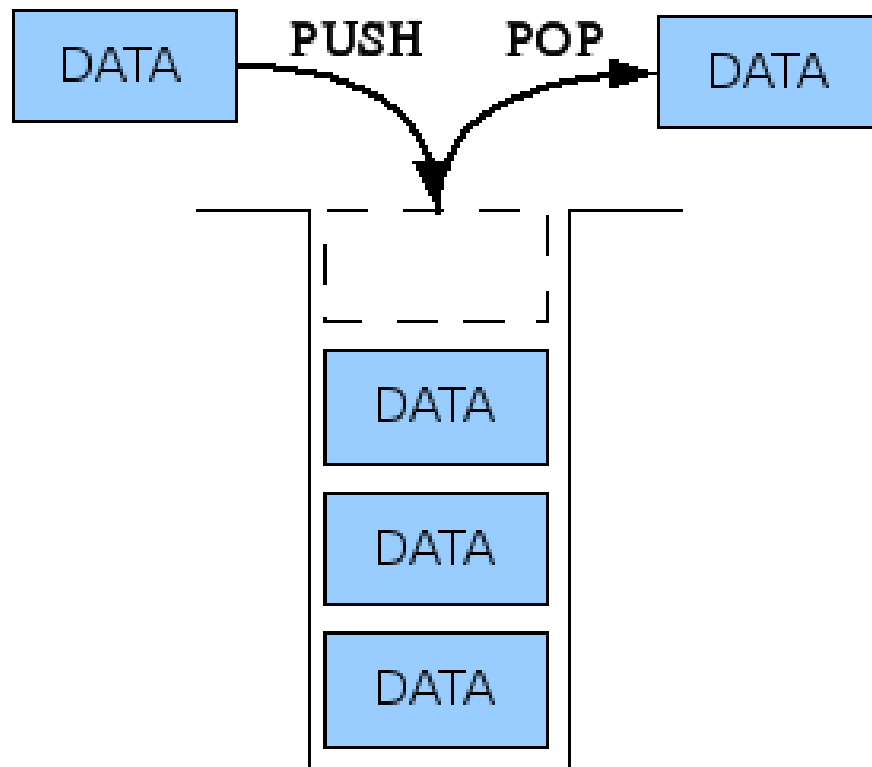
Container Adaptor

- stack
- queue
- priority_queue

Container Adaptor

시퀀스 컨테이너를 변형하여 자료를
미리 정해진 일정한 방식에 따라 관리
반복기를 지원하지 않음

Stack



Stack

stack은 기본적으로 deque으로 구현

생성자를 통해 내부를 vector로 구현 가능

`push_back() = push()`

`pop_back() = pop()`

`back() = top()`

Stack

push() : 원소 삽입

pop() : 원소 제거 및 반환

top() : 원소 반환

Stack

`empty()` : 원소가 없는가?

`size()` : 원소의 개수 반환

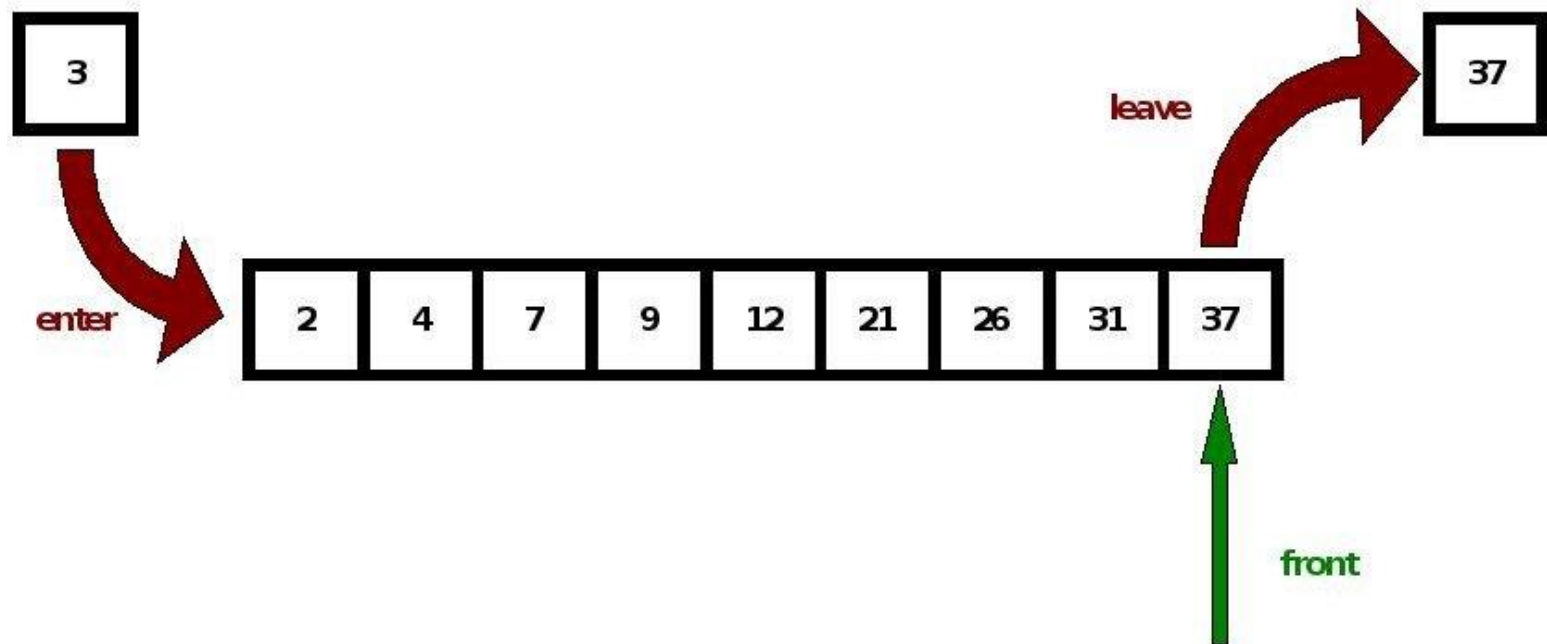
Stack

```
int main()
{
    stack<int> stack1;
    stack<int, vector<int>> stack2;

    stack1.push(1);
    stack1.push(2);
    stack1.push(3);
    stack1.push(4);
    stack1.push(5);

    while (!stack1.empty())
    {
        cout << stack1.top() << endl;
        stack1.pop();
    }
}
```

queue



queue

queue는 기본적으로 deque으로 구현
하지만 stack과 달리 내부는 vector로 사용 불가

queue

vector의 앞쪽에서 요소를 추가하거나 제거하면
나머지 요소들이 모두 당겨지고 밀려야 하므로
성능상의 문제가 생김

queue

push() : 원소 삽입

front() : 첫 원소 참조

pop() : 원소 제거

back() : 마지막 원소 참조

queue

`empty()` : 원소가 없는가?

`size()` : 원소의 개수 반환

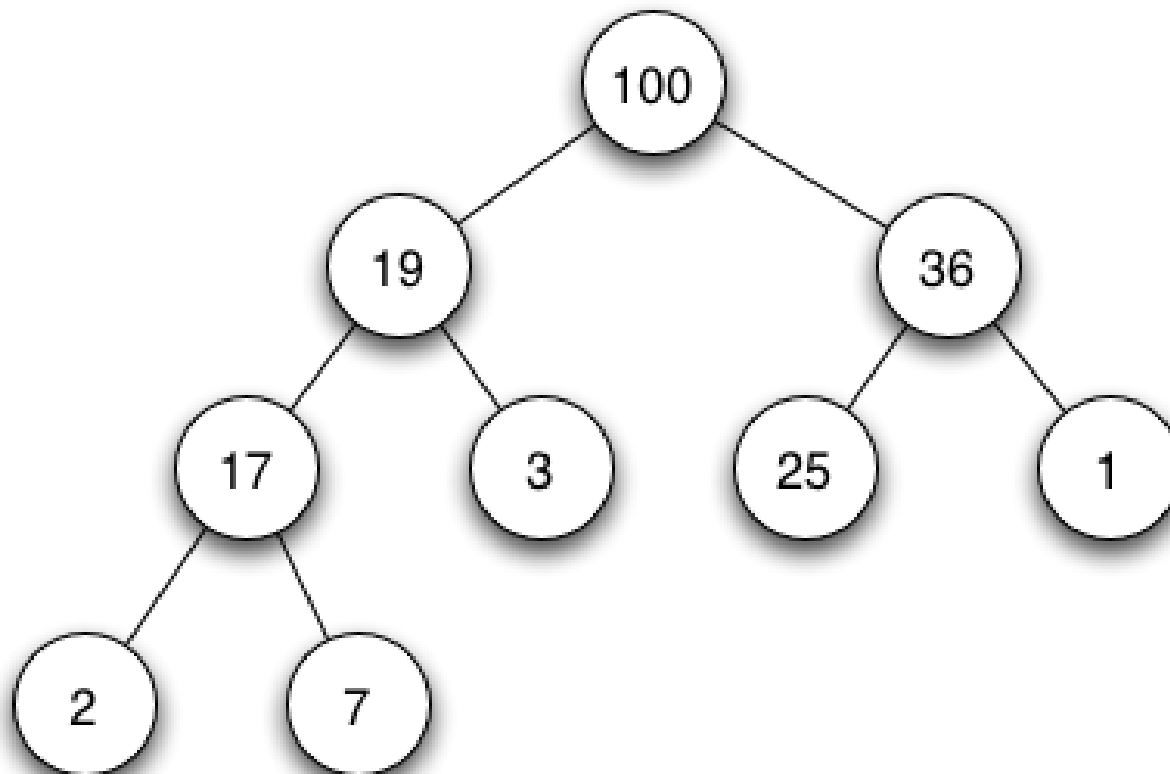
queue

```
int main()
{
    queue<int> queue1;

    queue1.push(1);
    queue1.push(2);
    queue1.push(3);
    queue1.push(4);
    queue1.push(5);

    while (!queue1.empty())
    {
        cout << queue1.front() << endl;
        queue1.pop();
    }
}
```

priority_queue



priority_queue

들어간 순서에 상관없이
우선 순위에 따라 데이터를 출력하는 자료구조

priority_queue

완전 이진 트리인 heap으로 구현
make_heap(), push_heap(), pop_heap()

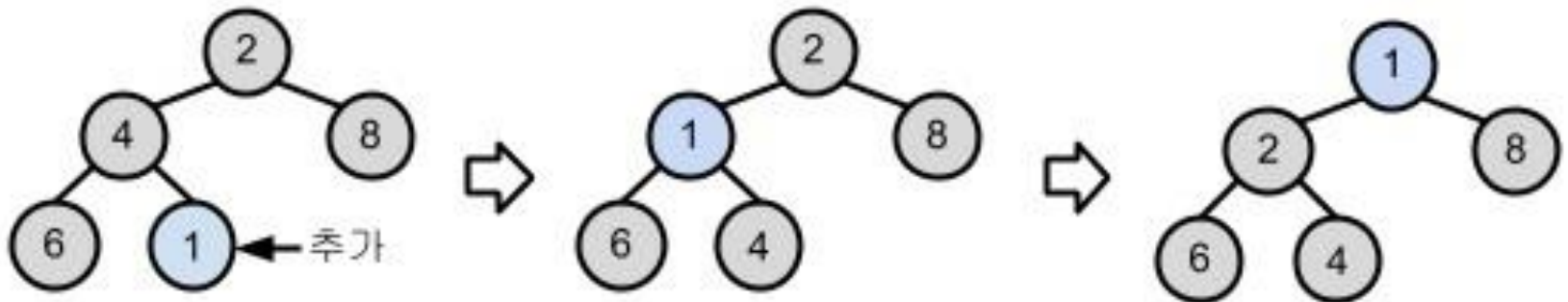
priority_queue

heap은 일반적으로 배열로 구현

따라서 내부는 vector, deque로 이루어짐

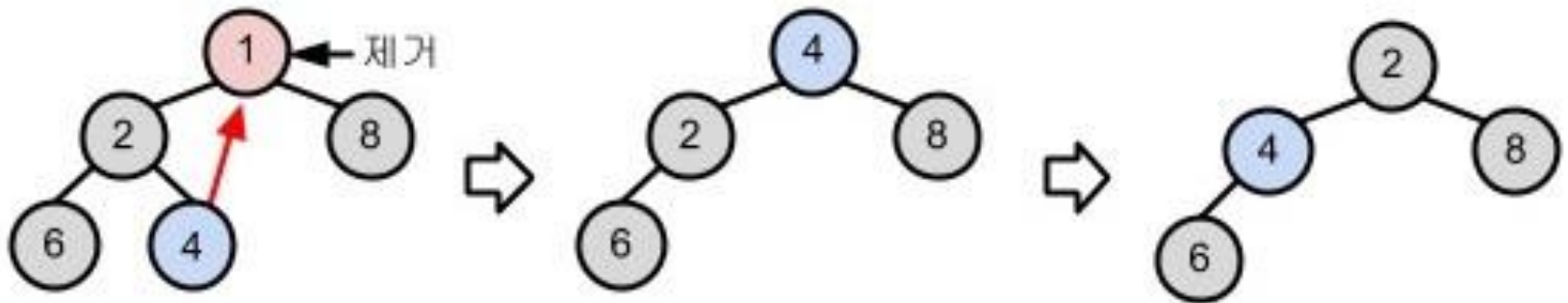
priority_queue

· 노드 추가



priority_queue

- 노드 제거



priority_queue

비교 연산은 부모 자식 간에만 일어남

시간 복잡도 : $O(\log_2 n)$

priority_queue

push() : 원소 삽입

top() : 앞 원소 참조

pop() : 원소 제거

priority_queue

`empty()` : 원소가 없는가?

`size()` : 원소의 개수 반환

priority_queue

greater<T>

less<T>

priority_queue

알고리즘의 기본 동작을 변형하거나
확장시켜주는 객체

priority_queue

```
int main()
{
    priority_queue<int> priorityQueue1;
    priority_queue<int, vector<int>, greater<int>> priorityQueue2;

    priorityQueue1.push(3);
    priorityQueue1.push(4);
    priorityQueue1.push(2);
    priorityQueue1.push(5);
    priorityQueue1.push(1);

    while (!priorityQueue1.empty())
    {
        cout << priorityQueue1.top() << endl;
        priorityQueue1.pop();
    }
}
```

priority_queue

```
class Data
{
public:
    int _data;
    Data(int data) : _data(data) {}
};

bool operator <(const Data &a, const Data &b)
{
    return a._data < b._data;
}

int main()
{
    priority_queue<Data> priorityQueue1;
    priority_queue<Data, vector<Data>, greater<Data>> priorityQueue2;
```

Associative Containers

TOGATER

what is Associative Containers?

- Associative Containers?
 - 요소들은 미리 정의된 순서로 삽입되고 정렬된다.
 - 앞에서 소개한 컨테이너들처럼 반복자를 제공한다.
 - 연관 컨테이너는 Map , Set 으로 나눌 수 있다.
 - 주로 검색기능을 활용할 때 사용하는 Container이다

what is Associative Containers?

• 장점

- 검색시에 매우 유리하다. $O(\log n)$ 의 시간복잡도를 가진다.

단점

- 원소(key)값을 직접 수정할 수 없다. 삭제 후 수정해야만 한다.

what is Map?

- **Map?**

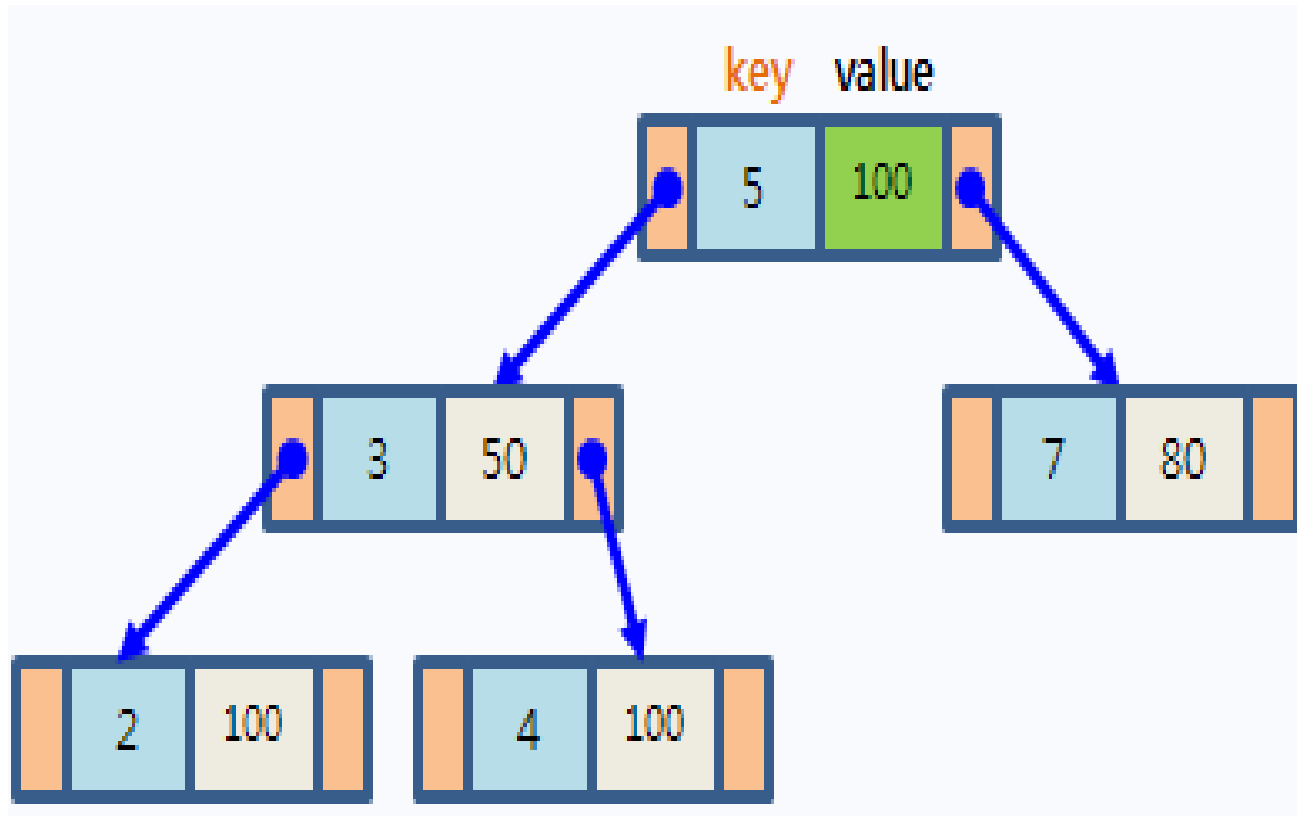
- 일종의 사전으로 <key,value> 쌍의 원소를 관리하는 컨테이너
- 모든 Key들은 중복되면 안된다.
- 내부 구조는 key 값에 따라 정렬된 균형 이진 트리 구조를 가진다.

what is Map?

- Map class

```
template <
    class Key,
    class Type,
    class Traits=less<Key>,
    class Allocator=allocator<pair <const Key, Type> >
> class multimap;
```

Structure of Map



How to use

- Map 의 사용법(삽입)

- `insert(std::pair<key,value>(key,value))` 로 삽입
- 배열 첨자[key]=value로 삽입

(기존에 있던 key라면 덮어 쓰기 처리)

Pair

- **Pair?**

- 페어는 두 개의 값으로 구성되어 있는데, 각각의 값의 실제 타입에 상관없이 첫번째 것을 first라고 하고 두 번째 것을 second라고 한다.
- Map 에서는 <key,value> 쌍을 pair로 받아 요소로 사용하며 first는 key가 되고 second는 value가 된다.

Method of Map

- Map의 사용법 (검색)

- Find(key) 로 검색 (key가 있다면 <key,value>쌍의 반복자 리턴, 없다면 map.end() 리턴)
- 배열 첨자[key]로 검색 (key가 있다면 value 리턴 , 없다면 value=0인 <key,value> 쌍 생성)

Method of Map

- Map의 사용 예제 (삽입 및 검색)

```
int main()
{
    std::map<std::string, int> map_;

    map_.insert(std::pair<std::string, int>("smile", 20));

    std::cout << map_["smile"] << std::endl;

    auto find = map_.find("smile");
    if (find != map_.end())
    {
        std::cout << find->second << std::endl;
    }

    find = map_.find("gate");
    if (find == map_.end())
    {
        std::cout << "failed to find" << std::endl;
    }
    std::cout << map_["gate"] << std::endl;

    return 0;
}
```

```
20
20
failed to find
0
```

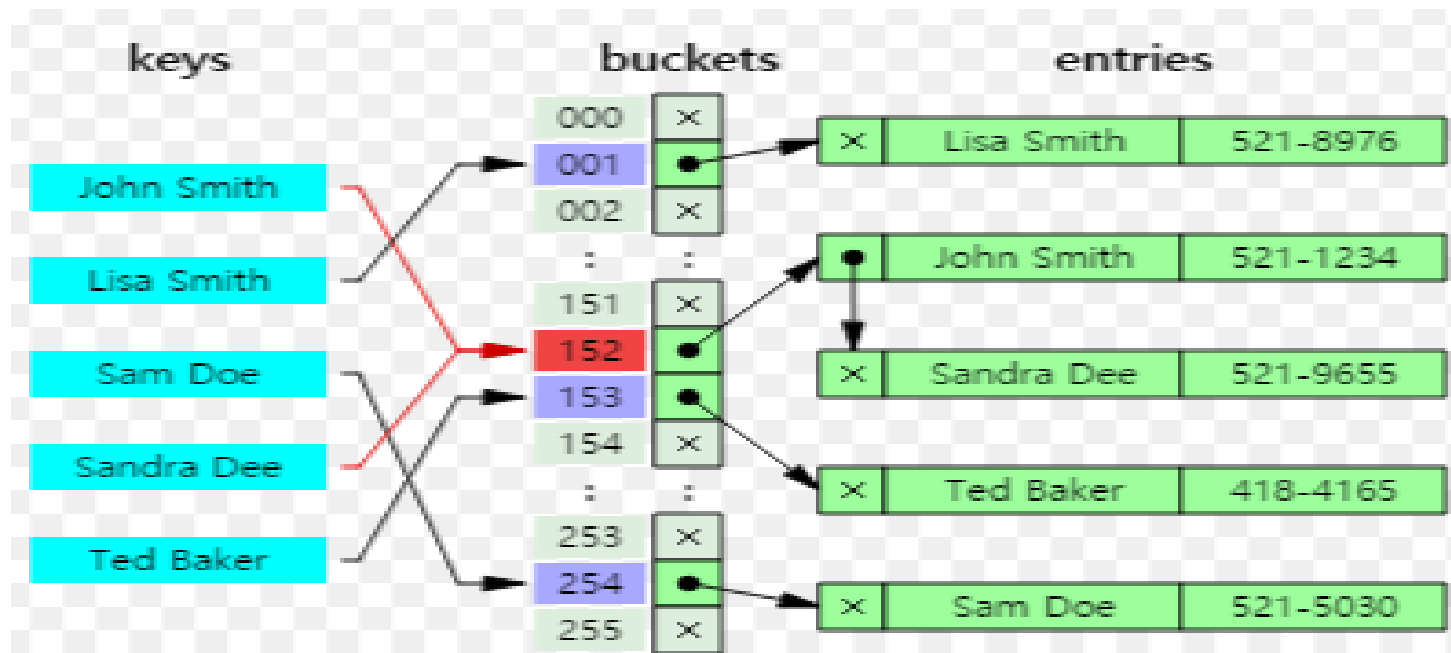
why use map?

- <key,value>를 순서대로 탐색 해야한다 -> map
(Ex. 이름으로 정렬된 전화번호부 등등등)
- <key,value>에 대한 접근만 중요함 -> unordered_map

why use map?

- **Unordered_map**

- Hash 구조로 되어 있어 모든 key에 접근속도가 일정하다.



what is Set?

- **Set?**

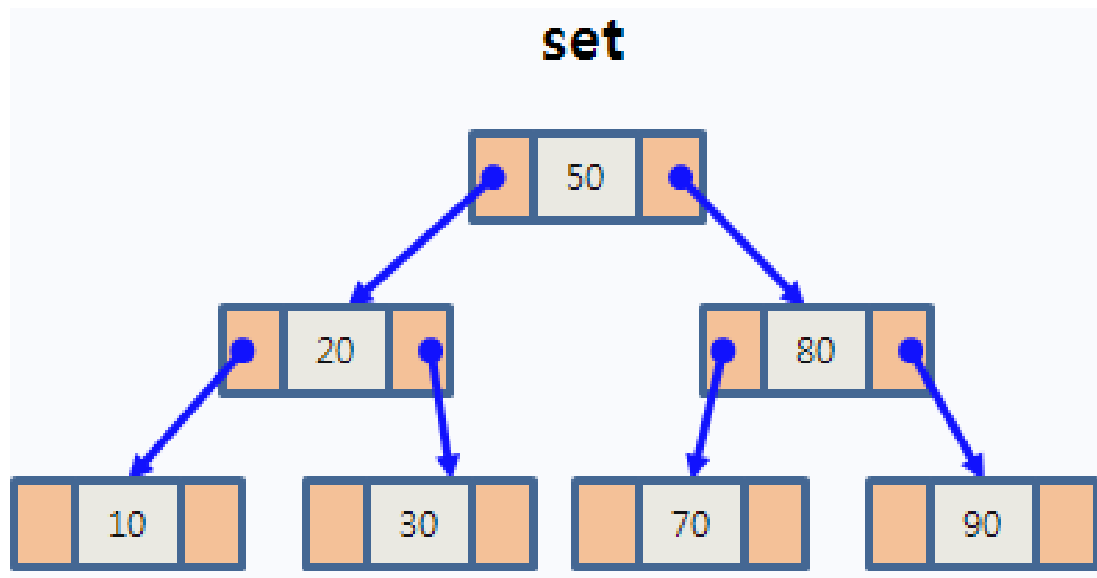
- <key> 들의 집합체 이다. Key들은 중복 되지 않는다.
- Map과 마찬가지로 균형 이진 트리 구조로 되어 있다.
- 직접적인 원소 액세스를 허락하지 않는다.

what is Set?

- Set class

```
template <
    class Key,
    class Traits=less<Key>,
    class Allocator=allocator<Key>
>
class set
```


Structure of Set



How to use

- Set 의 사용법(삽입)
 - Insert(key) 로 삽입

How to use

- **Set 의 사용법(검색)**

- `find(key)` 로 검색 (`key`가 있다면 해당 반복자 리턴 반대 경우 `set.end()` 리턴)
- `equal_range(key)` 로 검색 (`key`가 있다면 시작 지점의 반복자와 끝 지점의 반복자를 `pair` 타입으로 리턴 해 준다)

How to use

- Set 의 사용 예제(삽입 및 검색)

```
#include <iostream>
#include <set>
#include <functional>
int main()
{
    std::set<int> set_;

    set_.insert(1);
    set_.insert(2);
    set_.insert(3);

    if (set_.find(2) != set_.end())
    {
        std::cout << "find key : " <<*(set_.find(2)) << std::endl;
    }

    return 0;
}
```

why use set?

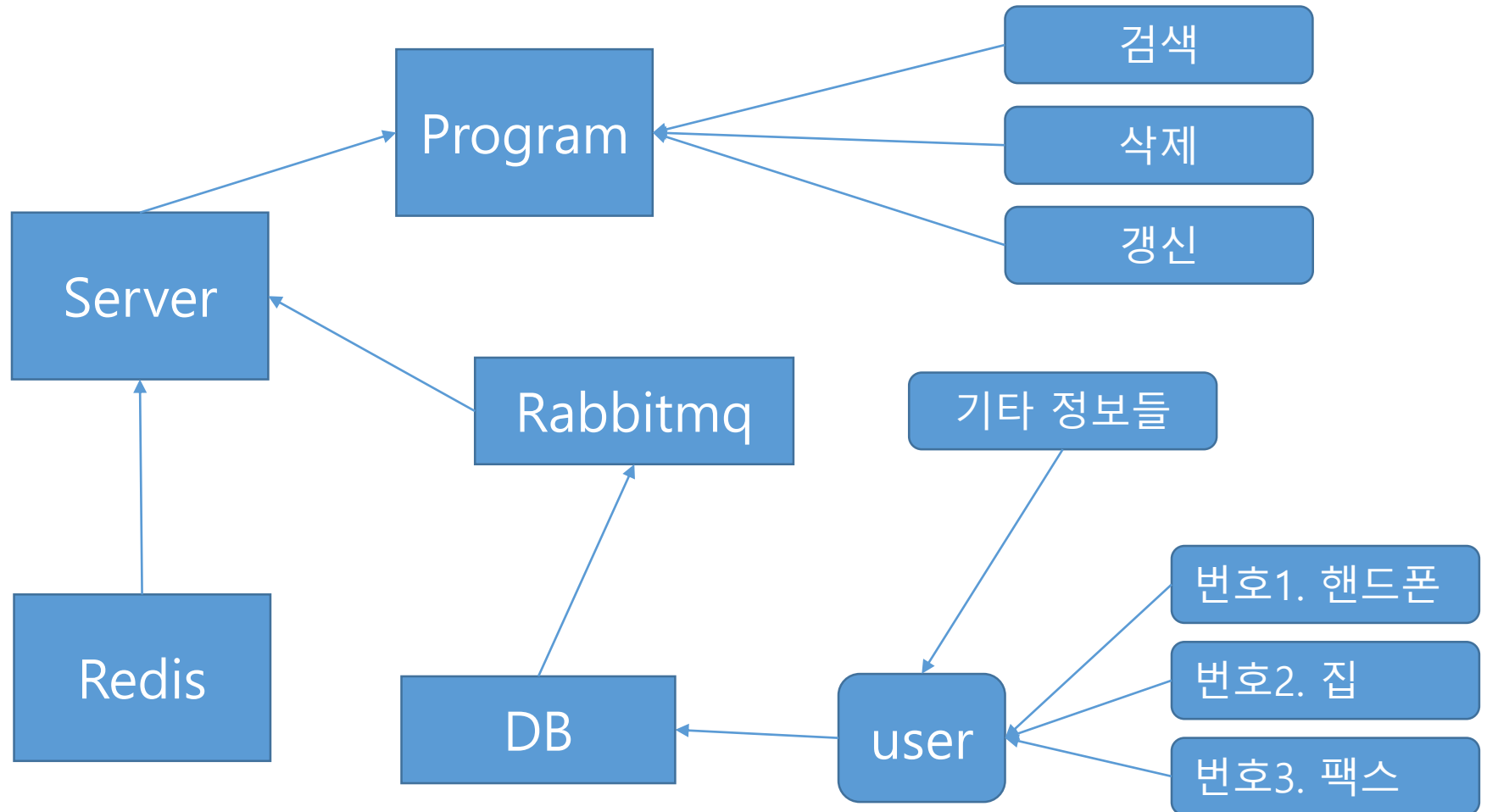
- 유니크한 집합을 순서대로 출력해야 한다 -> set
- key에 대한 유니크성 만을 체크해야한다 -> unordered_set

STL Container Performance table

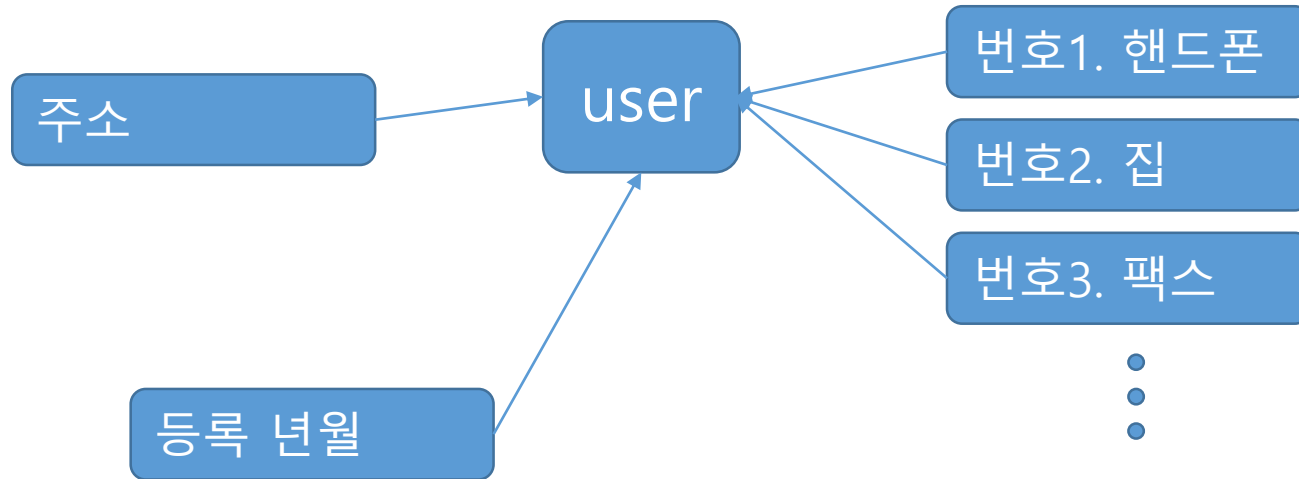
Container	Insertion	Access	Erase	Find	Persistent Iterators
vector / string	Back: $O(1)$ or $O(n)$ Other: $O(n)$	$O(1)$	Back: $O(1)$ Other: $O(n)$	Sorted: $O(\log n)$ Other: $O(n)$	No
deque	Back/Front: $O(1)$ Other: $O(n)$	$O(1)$	Back/Front: $O(1)$ Other: $O(n)$	Sorted: $O(\log n)$ Other: $O(n)$	Pointers only
list / forward_list	Back/Front: $O(1)$ With iterator: $O(1)$ Index: $O(n)$	Back/Front: $O(1)$ With iterator: $O(1)$ Index: $O(n)$	Back/Front: $O(1)$ With iterator: $O(1)$ Index: $O(n)$	$O(n)$	Yes
set / map	$O(\log n)$	-	$O(\log n)$	$O(\log n)$	Yes
unordered_set / unordered_map	$O(1)$ or $O(n)$	$O(1)$ or $O(n)$	$O(1)$ or $O(n)$	$O(1)$ or $O(n)$	Pointers only
priority_queue	$O(\log n)$	$O(1)$	$O(\log n)$	-	-

Example

인터넷 저장 전화번호부를 만든다고 생각해보자



Example

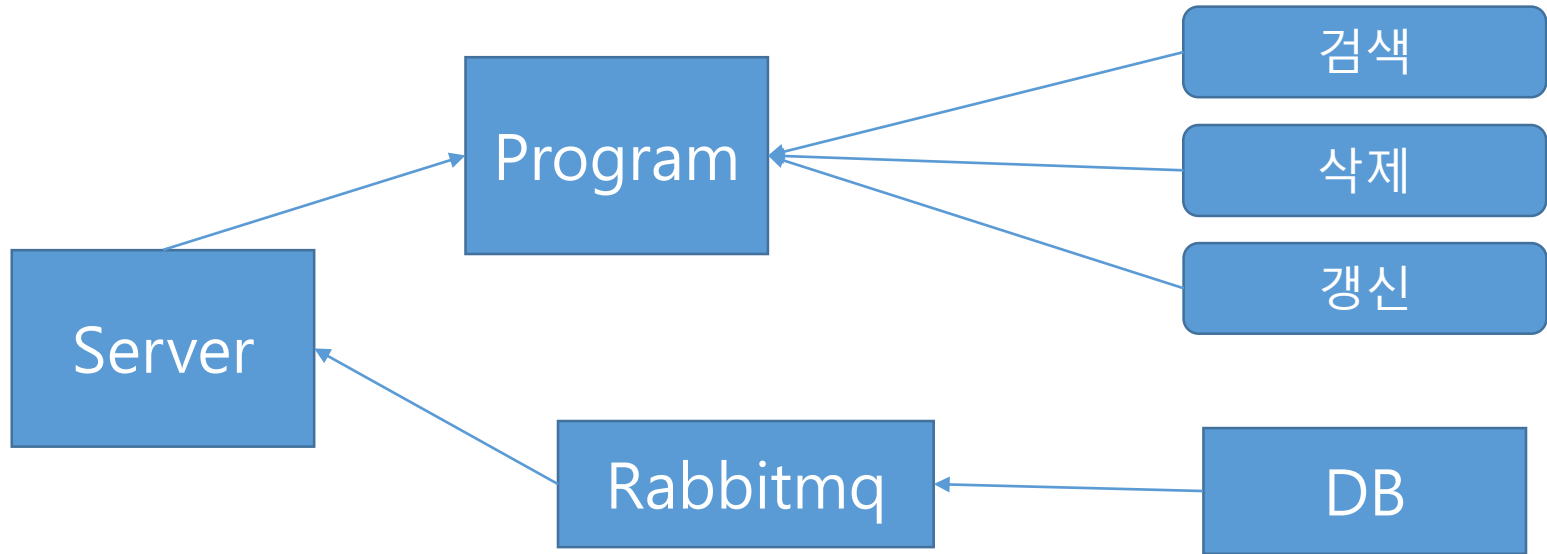


User가 가지는 정보는 쉽게 변하거나 수정되지 않는다.

정보 수가 유저마다 다르다.

⇒ 유저 내 데이터는 `vector` 또는 `list`를 사용하여 관리한다.

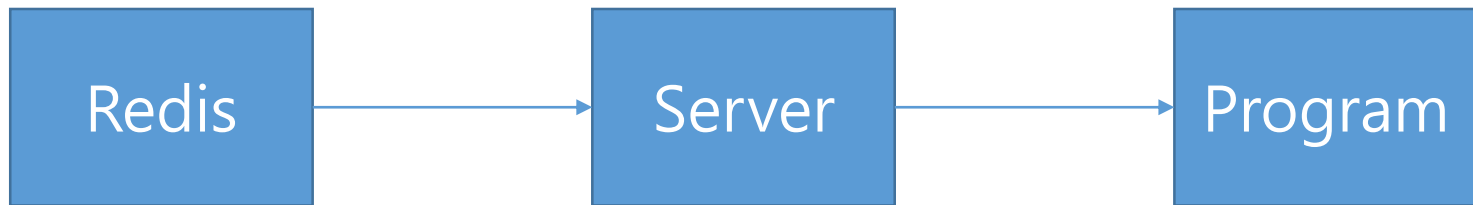
Example



client가 많아 server가 DB에 접근을 많이 한다면 DB에 보내는 query는 큐에 넣어서 하나씩 처리하는 것이 좋다

=> Rabbitmq => queue 구조

Example



DB Server에 자주 호출되는 사용자 정보는 캐시에 저장해 둔다고
생각할 때, 캐시는 key에 의한 검색과 키 자체를 저장해 두어야
한다.

=> map 또는 set류의 자료 사용

END