첫 시간인 오늘은 여러분이 앞으로 배울 자료구조에 대한 간략한 개요로, C++ containers library 에 있는 자료구조를 알아보겠습니다.

각 container 의 자세한 사용법은 아래 링크를 참고하십시오.
https://en.cppreference.com/w/cpp/container

Sequence Containers

Sequence Containers 는 데이터를 **순차적으로 저장하는** 자료구조입니다. 구현이 단순하므로 가볍고 빠릅니다. 여러분이 저장할 데이터가 정렬 상태를 계속 유지할 필요가 없다면 좋은 선택입니다.

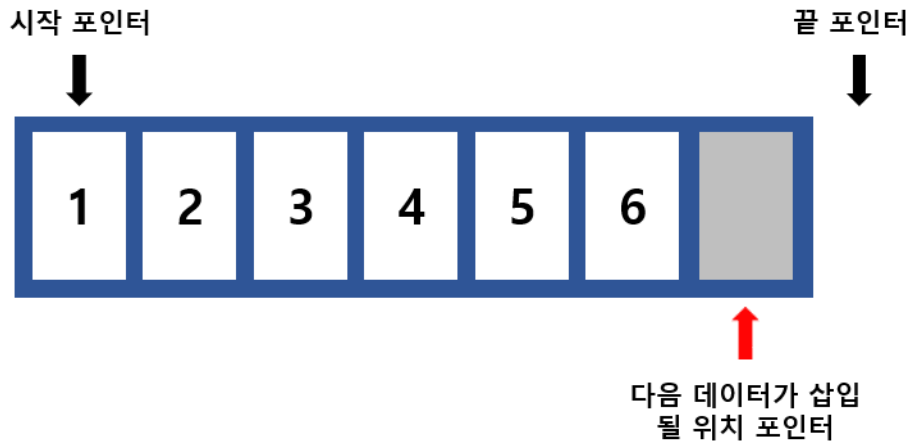
std::vector (std::array)

메모리상에서 데이터가 연속적으로 위치하는 배열입니다.

std::vector 의 경우 배열의 크기를 런타임에서 조절할 수 있지만 std::array 의 크기는 컴파일 타임에 결정되어야 하고 런타임에서 바꿀 수 없습니다. 여러분이 사용할 배열의 크기를 미리 알 수 있고 크기가 변하지 않는다면 std::array 를, 그렇지 않다면 std::vector 를 선택하시면 됩니다.

std::vector 는 포인터 세 개로 구현되어 있습니다.
1. 할당된 배열의 시작 주소를 가리키는 포인터
2. 다음에 데이터가 삽입될 위치를 가리키는 포인터
3. 할당된 배열의 끝 주소를 가리키는 포인터

그림으로 나타내면 다음과 같습니다.



데이터를 뒤에 삽입하면 빨간 포인터가 가리키는 곳에 삽입되고, 빨간 포인터가 1 증가합니다.

vector 데이터 삽입 시 주의사항:
일반적으로 데이터를 뒤에 삽입하는 push_back 연산은 **상수 시간복잡도**를 가지지만. 할당된 공간이 전부 차면 배열을 통째로 복사해 새로운 vector 에 할당하는 reallocation 이 발생해 시간이 많이 소요됩니다. 따라서, 알고리즘 문제 풀이 시에는 가급적 vector 의 size 를 충분히 확보한 상태로 사용하는 것이 좋습니다. 해당 내용은 아래 예시 코드를 통해 직접 확인해 볼 수 있습니다.

예시 코드:

```
vector<int> array(8, 1);
cout << array.capacity() << endl;
array.push_back(2); // Reallocation happens
cout << array.capacity() << endl;
```

용량이 8 인 vector 가 전부 차있는 상태에서 push_back 을 하면, vector 재할당이 일어나고, vector 의 용량이 12 로 변경된 것을 확인할 수 있습니다.

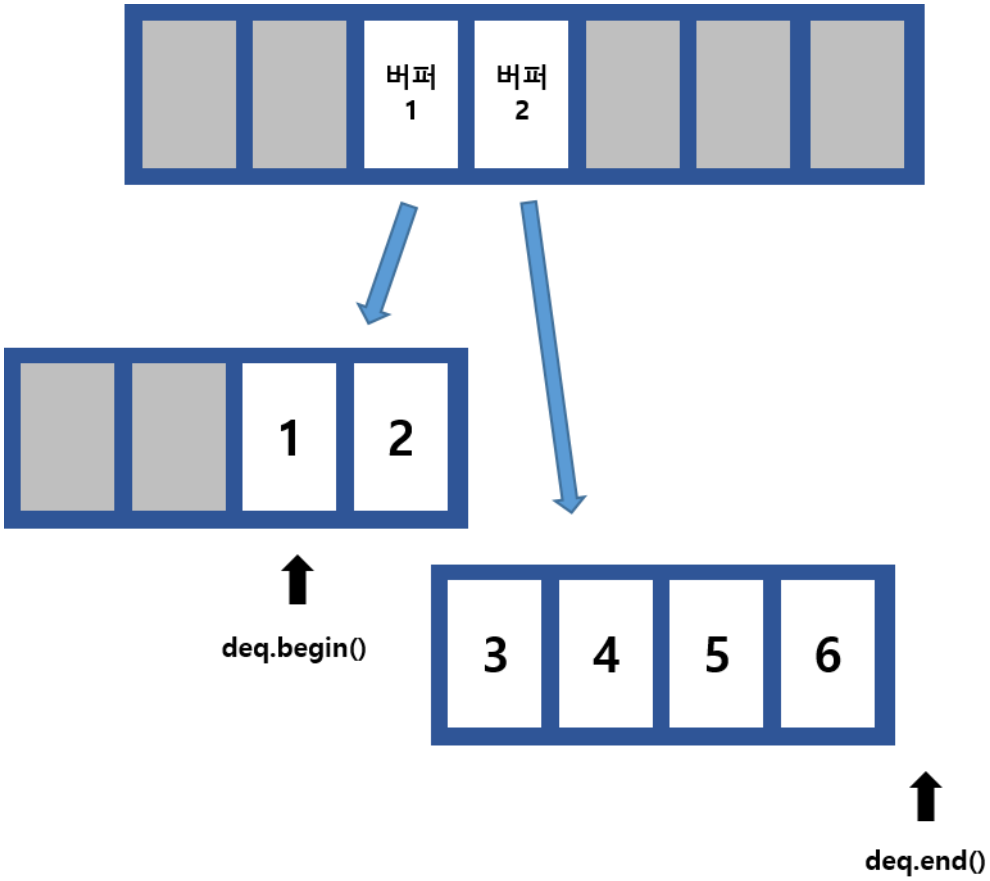
참고 : 해당 내용에 더하여, C++0x 에서부터는 백터의 길이가 늘어날 때 원소를 하나하나 복사하는 것이 아닌 우측 값 참조(메모리 상에서의 이동)으로 이루어지는 성능 향상이 있었습니다.
(https://skstormdummy.tistory.com/entry/%EC%9A%B0%EC%B8%A1-%EA%B0%92-%EC%B0%B8%EC%A1%B0-RValue-Reference)

std::deque

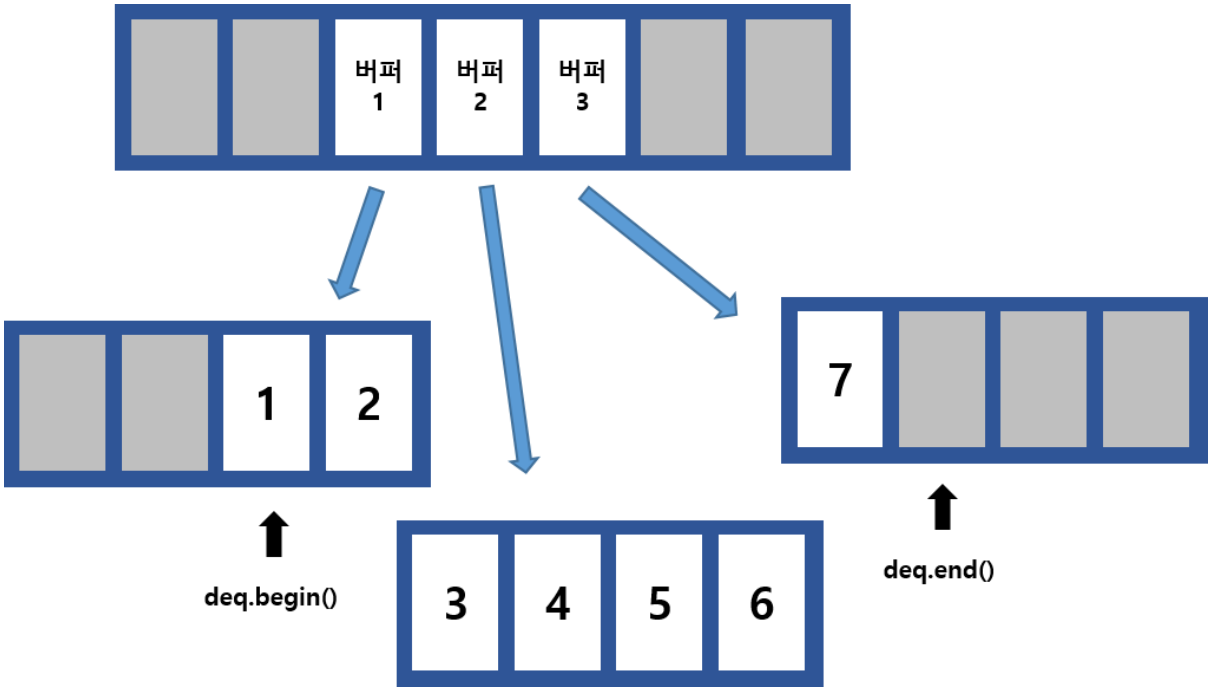
Container 앞, 뒤에 데이터를 빠르게 넣고 뺄 수 있는 double-ended queue 입니다.

std::deque 는 여러 개의 버퍼에 데이터를 나눠서 저장합니다. std::vector 는 할당된 공간이 전부 차면 배열을 통째로 새로 할당하는 반면, std::deque 는 버퍼 하나만 할당하면 되므로, 데이터 삽입이 언제든 O(1)입니다. 컨테이너의 앞에 데이터 삽입/삭제, 컨테이너의 뒤에 데이터 삽입/삭제의 기능이 동시에 필요한 경우, deque 는 유용한 선택이 됩니다.

그림으로 나타내면 다음과 같습니다.



새로운 값 7 을 맨 뒤에 삽입한 이후의 그림은 아래와 같습니다.



std::vector 와 달리 데이터가 저장된 공간의 **앞**과 뒤에 빈 공간을 계속 넣을 수 있습니다.

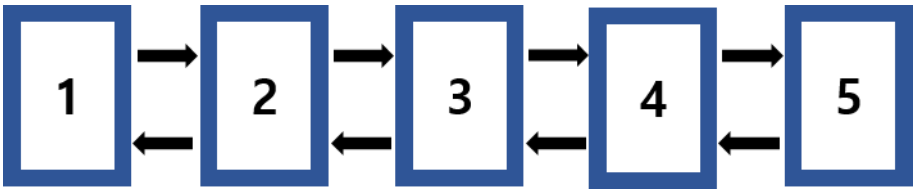
Vector vs Deque 추가적인 차이점:

해당 내용만 보면 deque 가 vector 보다 대부분의 상황에서 유리해 보일 수 있습니다. 하지만 vector 에는 deque 에는 없는 또다른 차별점이 있습니다. Vector 의 요소들은 메모리상에 연속적으로 존재하는 것이 보장되지만, deque 는 메모리에서 요소들이 연속적이지 않습니다. 따라서 C 배열의 라이브러리와 상호작용해야 하는 상황이거나, 공간지역성을 고려해야하는 상황에서는 deque 가 불리한 점이 있습니다.

std::list (std::forward_list)

linked list 입니다. Container 의 어느 위치든 O(1)에 데이터를 삽입하거나 삭제할 수 있지만 random access(Container 의 i 번째 데이터에 O(1)에 접근)는 불가능합니다.

std::list 는 doubly-linked list 이고, std::forward_list 는 singly-linked list 입니다. singly-linked list 는 몇 가지 기능이 제한됩니다. Container 맨 앞에 데이터를 삽입하는 건 빠르지만 맨 뒤에 삽입하는 건 느립니다. 어떤 iterator 의 다음 iterator 는 삭제할 수 있지만 그 iterator 자신은 삭제하지 못합니다. 하지만 doubly-linked list 에 비해 포인터를 하나 덜 가지므로 기본적인 연산이 빠르고 메모리를 적게 사용합니다. 만약 여러분이 doubly-linked list 의 기능까지 필요하지 않다면 더 가볍고 빠른 std::forward_list 가 좋은 선택일 것입니다.



Associative Containers

데이터를 정렬된 상태로 유지하는 자료구조입니다. **Red-Black Tree** 를 기반으로 하고 데이터의 추가/삭제/접근의 시간복잡도가 O(logn)입니다. 데이터를 정렬된 상태로 유지하는 것은 매우 강력한 기능이고 O(logn)은 적은 시간복잡도지만 연산에 붙는 상수가 크고 사용하는 메모리가 많으므로 주의가 필요합니다.

Red-Black Tree:

자가 균형 이진 탐색 트리(self-balancing binary search tree)로서, 대표적으로는 연관 배열 등을 구현하는 데 쓰이는 자료구조입니다. 트리에 n 개의 원소가 있을 때 O(log *n*)의 시간복잡도로 삽입, 삭제, 검색을 할 수 있으며 최악의 경우에도 우수한 실행 시간을 가집니다.

std::set (std::map)

Red-Black Tree 는 Binary Search Tree 이므로 어떤 key 를 기준으로 데이터를 저장합니다. std::set 은 데이터를 자체를 key 로 사용하고, std::map 은 (key, value) 쌍을 받아서 사용합니다.

단순히 데이터를 정렬 상태로 유지하고 싶다면 std::set 을, (key, value) 데이터 쌍을 key 를 기준으로 정렬하고 싶다면 std::map 을 사용하면 됩니다.

std::multiset (std::multimap)

원소의 중복을 허용합니다. 같은 key 를 여러 개 저장하고 싶을 때 사용합니다. 단, 시간복잡도 주의가 필요합니다. std::set 에서 key 의 개수를 세거나, key 를 지우는 함수는 모두 O(logn)이지만, std::multiset 에서는 같은 key 를 모두 세고, 모두 지우므로 O(logn + (같은 key 를 가지는 데이터의 개수)) 만큼의 시간이 듭니다.

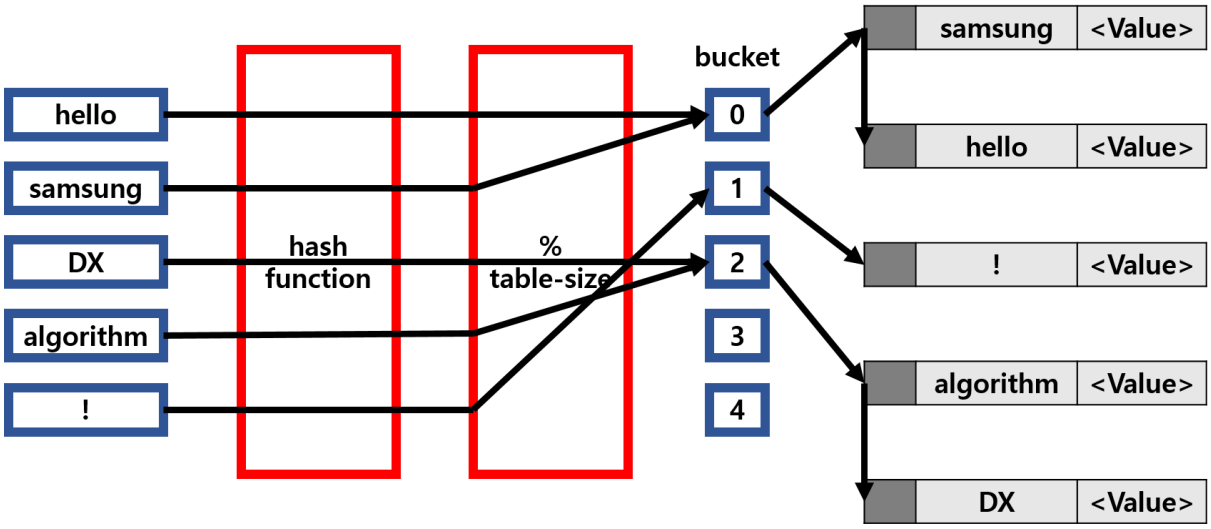
Unordered Associative Containers

해시값을 사용해 데이터를 저장하는 자료구조입니다. 대부분의 경우에서 데이터의 추가/삭제/접근이 O(1)이므로 Associated Container 보다 효율적입니다. 하지만 데이터를 정렬된 상태로 유지해야 하거나 (무수히 많은 key 값들로 인해) 해시 충돌이 걱정되는 상황이라면 Associated Container 를 사용하는 게 좋습니다.

std::unordered_set (std::unordered_map)

Data 를 중복 없이 저장하고 싶고, Data 의 순서가 상관없을 때 Associateve Container 대신 사용할 수 있습니다.

std::unordered_map 의 데이터 저장 방식은 아래와 같습니다.



데이터를 여러 개의 버킷에 나눠서 저장합니다. 주어진 key 의 해시값을 계산하고 이를 버킷 개수로 나눈 나머지를 구해서 어떤 버킷에 들어갈지 계산합니다.
중요한 점은 다른 key 여도 같은 버킷에 들어갈 수 있다는 것입니다. 이를 해시 충돌이라 하며, 해시 충돌에 대응하기 위해 std::unordered_map 은 각 버킷마다 linked list 로 (key, value) 쌍을 저장합니다. 만약 해시값이 고르지 않게 분포하면 하나의 버킷에 모든 데이터가 삽입될 수 있고, 데이터를 추가/삭제/접근하기 위해 linked list 를 순회해야 하므로 최악의 경우 시간복잡도가 O(n)이 됩니다.

기본으로 사용하는 hash function 은 std::hash 이고, std::hash 에서 기본으로 지원되는 타입은 int, double 등의 primitive type 과 std::string 등이 있습니다.

std::unordered_multiset (std::unordered_multimap)

Associateve Container 때와 마찬가지로, 같은 key 를 가진 데이터를 중복으로 가져야 할 때 사용됩니다.

Container Adaptors

이들은 std library 에 실제로 구현되어 있지 않습니다. Sequence Container 를 건네주면 그것을 자기 용도에 맞춰서 사용할 수 있도록 인터페이스만 제공합니다.

std::stack, std::queue

LIFO (Last-In, First-Out) 자료구조인 stack 과 FIFO (First-In, First-Out) 자료구조인 queue 의 기능을 제공합니다. 이때 기본적으로 deque 컨테이너를 이용하여 저장하고 있습니다.

예를 들어 queue 정의를 보면 다음과 같습니다.

template<class T,class Container = std::deque> class queue;

std::priority_queue

Container 를 max heap 으로 유지합니다. 데이터가 완벽히 정렬된 상태는 아니지만 최댓값은 빠르게 찾을 수 있습니다. 연속적으로 데이터의 최댓값 또는 최솟값만 필요할 때는 상수가 큰 std::set 보다 훨씬 효율적으로 동작합니다.