STL 기초

Set / Map

한컴에듀케이션

STL Container

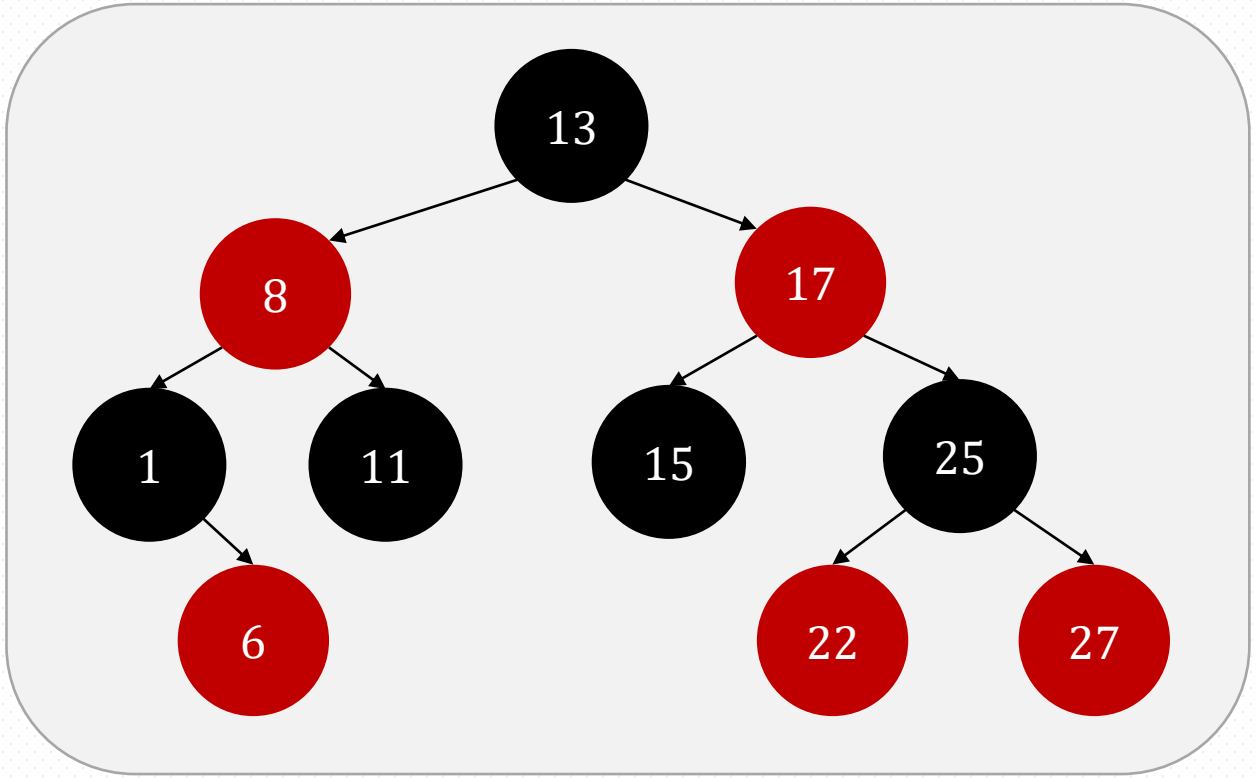
- **Sequence**
  - array : static array
  - vector : dynamic array
  - deque : dynamic array
  - foward_list : singly linked list
  - list : doubly linked list

- **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

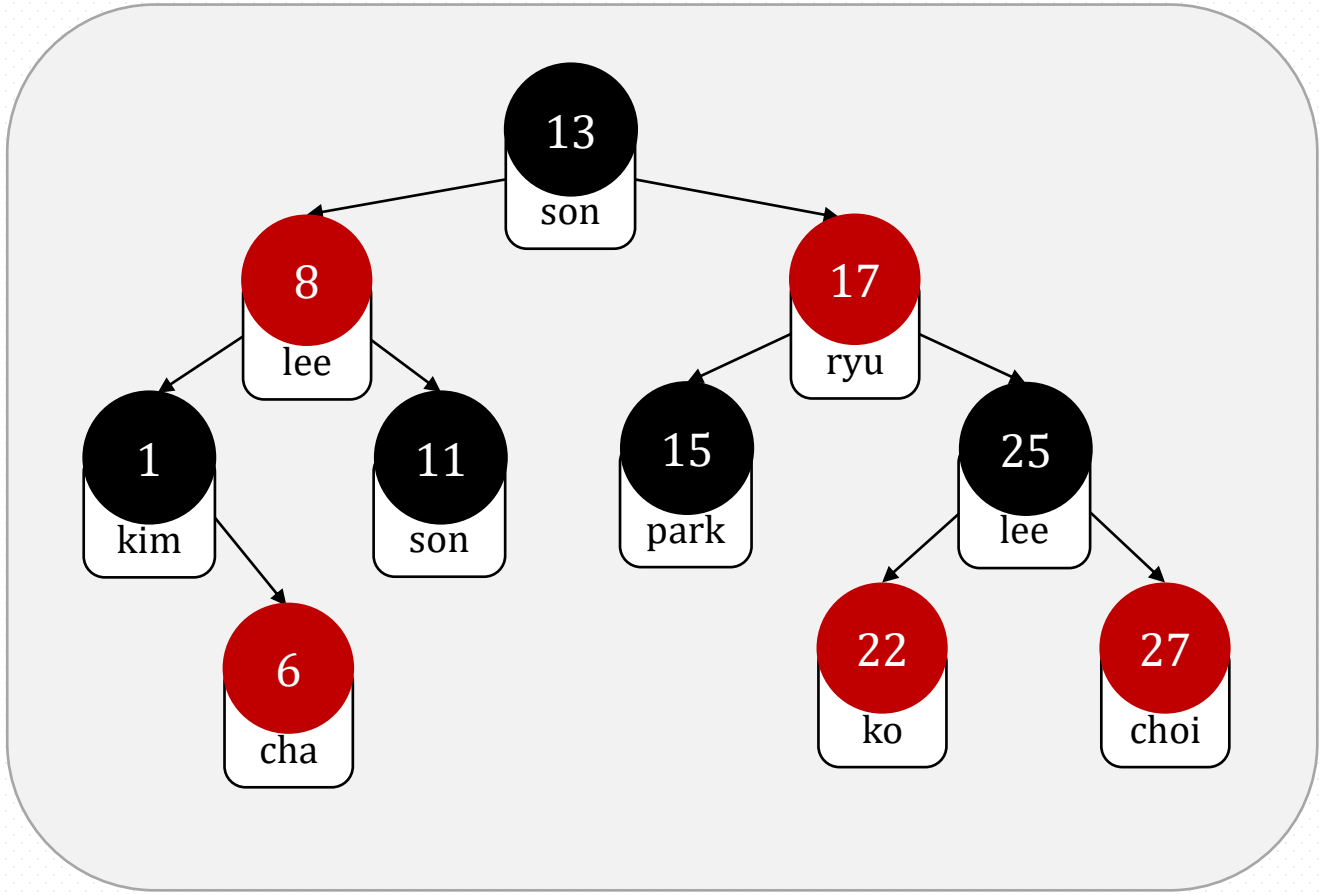
set 구조



- **Red-Black tree**
  : self-balancing binary search tree

	Average	Worst Case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

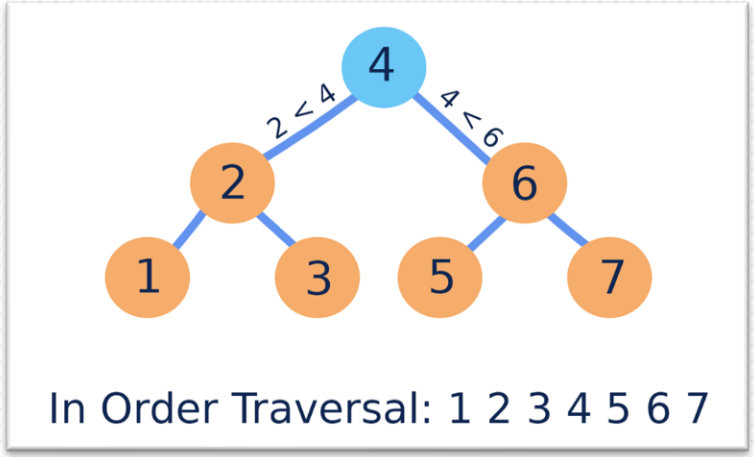
map 구조



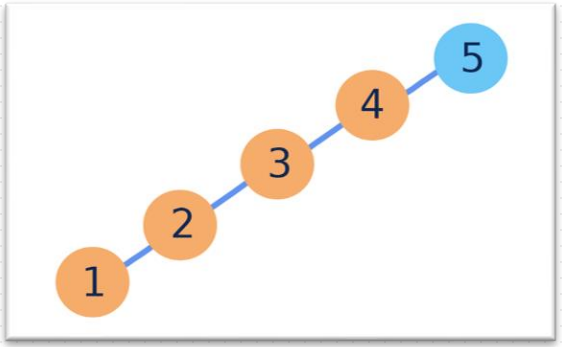
- **Red-Black tree**
  : self-balancing binary search tree

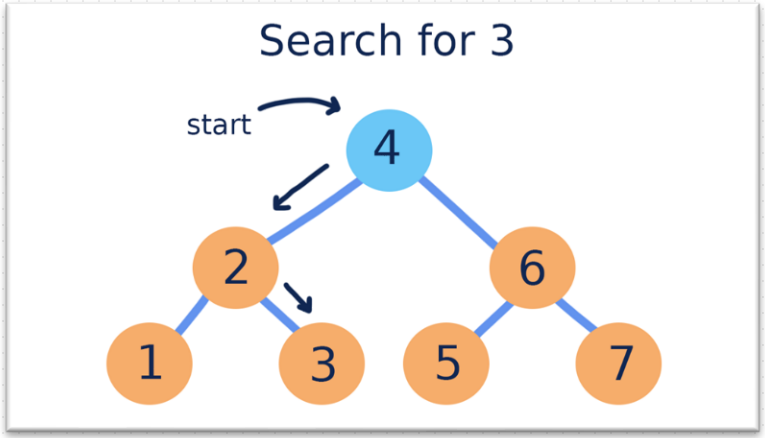
	Average	Worst Case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Binary Search Tree



- binary tree
- left child : 작은 값
- right child : 큰 값
- search time complexity : worst case O(n)



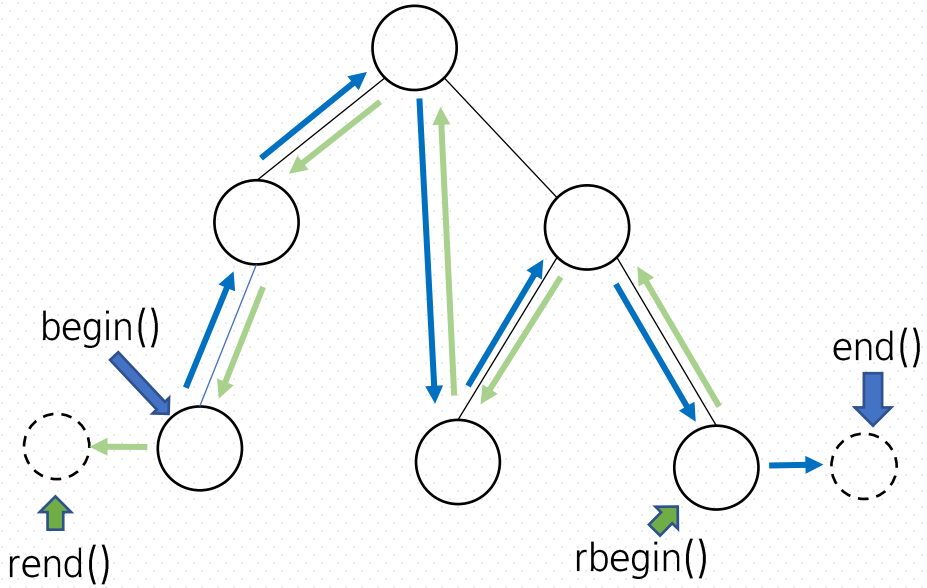


이를 개선한 구조 : self-balancing binary search tree
ex) **Red-Black tree**, AVL tree, Splay tree , …

Iterator

Bi-directional iterator

노드가 지워지지 않는 한, iterator는 유효
id별로 검색/삭제가 빈번할 시, iterator 기록 활용 필수



set 문법

```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
```

set<T> s

set<T, compare<T>> s

iterator s.begin(), s.end()

iterator s.rbegin(), s.rend()

bool s.empty()

void s.clear()

size_t s.size()

size_t s.count(T x) : 1, 0

iterator s.find(T x)

iterator s.lower_bound(T x)

※ pos, first, last 는 iterator
※ size_t : unsigned long long

pair<iterator, bool> s.insert(x)
: 등록된 iterator , 성공 여부

void s.insert(iterator first, iterator last)

iterator s.erase(iterator pos) *O(1)*

iterator s.erase(iterator first, iterator last) *O(dist(first,last))*
: 마지막 지워진 element의 다음 iterator

bool s.erase(T x) *O(log n)*
: 성공 여부

set example

compare 기준 : default less<int> , 오름차순

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

for (auto x : s) cout << x << ' ';                                  // 1 2 3 4 5
for (auto it = s.begin(); it != s.end(); ++it) cout << *it << ' ';  // 1 2 3 4 5
for (auto it = s.rbegin(); it != s.rend(); ++it) cout << *it << ' '; // 5 4 3 2 1

s.empty();          // 0
s.size();           // 5
s.count(1);         // 1
s.count(0);         // 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
auto ret = s.erase(--s.end());       // 맨 뒤 element 삭제, ret : 삭제된 element의 다음 iterator
```

Compare 설정 방법

※ Function Object으로 설정 *(다른 방법도 있지만 이 방법만 다룬다)*

1. pre-defined function object

- **less<T> : 오름차순**

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

- **greater<T> : 내림차순**

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

2. user-defined function object

- **int 절대값 오름차순**

```
struct AbsComp {
bool operator()(const int &l, const int &r) const {
        if(abs(l) != abs(r)) return abs(l) < abs(r)
        return l < r;
    }
};
```

※ 참조자(&) : 옵션
const : 필수

Compare 설정 예

- compare 기준 : greater<int> , 내림차순

```
set<int, greater<int>> s{ 1, 2, 3, 4, 5 };

for (auto x : s) cout << x << ' ';        // 5 4 3 2 1
```

- compare 기준 : 절대값 오름차순

```
struct AbsComp {
    bool operator()(const int &l, const int &r) const {
        if(abs(l) != abs(r)) return abs(l) < abs(r)
        return l < r;
    }
};

set<int, AbsComp> s{ -5, -3, 1, 4, 6 };
for (auto x : s) cout << x << ' '; // 1 -3 4 -5 6
```

Custom data type

1. **default - less<T> 활용** : operator< overloading 필요

```
struct Data {
    int a, b, c;
    bool operator<(const Data& r) const {
        return a < r.a;
    }
};

set<Data> s; // == set<Data, less<Data>> s;
```

2. **user-defined function object**

```
struct Data { int a, b, c; };

struct Comp {
    bool operator()(const Data&l, const Data&r) const {
        return l.a < r.a;
    }
};

set<Data, Comp> s;
```

우선순위 판별

- compare 기준으로 우선순위 높은 값, 낮은 값, 같은 값 모두 판별

- 값 A, B에 대해
  1. **comp(*A*,*B*) && !comp(*B*,*A*)** : *A*(left) , *B*(right)
  2. **!comp(A,B) && comp(B,A)** : *B*(left) , *A*(right)
  3. **!comp(A,B) && !comp(B,A)** : *A == B* : Key값 중복

- 따라서, compare가 만족해야 하는 requirement 존재 (strict weak ordering)

Compare Requirements

- **Function object**

Predicate return type : bool

find_if
all_of
…

BinaryPredicate return type : bool , argument : two
bool f(T a, T b)

lower_bound
upper_buond
unordered_set
unordered_map
…

Compare

set
map
sort
max
priority_queue
…

strict weak ordering

1. !comp(a,a)
2. comp(a,b) => !comp(b,a)
3. comp(a,b) && comp(b,c) => comp(a,c)
4. !comp(a,b) && !comp(b,a) => (a==b)

※comp(a,b) : a<b or a>b

요약

- <, > operator만 사용
- 좌우항이 같은 형태
  ex) l.x < r.x , l.x+l.y < r.x+r.y , ~~l.x < r.y~~

Compare 설정 예 – custom data type

1. **default - less<T> 활용** : operator< overloading 필요

```
struct Data {
    int a, b, c;
    bool operator<(const Data& r) const {
        return a < r.a;
    }
};

set<Data> s; // == set<Data, less<Data>> s;
```

2. **user-defined function object**

```
struct Data { int a, b, c; };

struct Comp {
    bool operator()(const Data&l, const Data&r) const {
        return l.a < r.a;
    }
};

set<Data, Comp> s;
```

compare 기준 : a 값 오름차순

key 값 중복 판별 : b, c 관계없이 a가 같으면 중복

- {1 , 2 , 3} == {1, 0, 0}
- {1 , 2 , 3} != {0, 2, 3}

map 문법

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
```

map<T, U> m

map<T, U, Compare<T>> m

U m[T x]

iterator m.begin(), m.end()

iterator m.rbegin(), m.rend()

bool m.empty()

size_t m.size()

void m.clear()

size_t m.count(T x)

iterator m.find(T x)

iterator m.lower_bound(T x)

※ size_t : unsigned long long

pair<iterator, bool> m.insert({ T x , U y})

: first (등록된 iterator) , second(성공 여부)

void m.insert(iterator first, iterator last)

iterator m.erase(iterator pos)

iterator m.erase(iterator first, iterator last)

: 마지막 지워진 element 바로 다음 iterator

bool m.erase(T x)

: 성공 여부

map example

- key 값에 대한 value 값 추가된 것 외에 set과 동일
- compare 기준은 무조건 key 값

```
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)
for (auto x : m) cout << x.first << ',' << x.second;
for (auto it = m.begin(); it != m.end(); ++it) cout << it->first << ',' << it->second;

// (5,0) (4,0) (3,0) (2,0) (1,0)
for (auto it = m.rbegin(); it != m.rend(); ++it) cout << it->first << ',' << it->second;

m[1] = 3;         // (1,3) (2,0) (3,0) (4,0) (5,0)
m[6];             // index 접근시, 등록되어 있지 않은 key값이면 default 값(0)으로 insert  : (6, 0)
m[7]++;           // key값 7이 등록되어 있지 않으므로 value 0으로 insert 후, value 1 증가 : (7, 1)

m  : (1,3) (2,0) (3,0) (4,0) (5,0) (6,0) (7,1)
```

multiset , multimap

- equal_range 활용 : 원하는 key값의 범위

  auto ret = m.equal_range(x);

  - ret.first = m.lower_bound(x);
  - ret.second = m.upper_bound(x);

- m.find(x) : x중 하나의 iterator 반환(검색 시 가장 먼저 만나는 iterator)

- m.erase(x) : x인 모든 원소 제거

- m.erase(m.find(x)) : x인 한 개의 원소 제거

- [] opeartor 제공 X (multi_map)

set, map 특징

- Key 값을 이용하여 Compare 기준으로 정렬
- Compare는 function object으로 설정
- Key 값은 변경 불가 (필요시, erase 후 insert)
- map의 Value 값은 변경 가능
- 모든 검색 관련 작업은 오로지 Key값으로만 진행
- set: key값만 존재, map: key값에 대한 상태 존재
- element가 erase되지 않는 한 iterator는 항상 유효


감사합니다