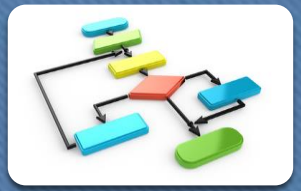
Searching

C++ Optimization Study

Jeongho Nam

INDEX



<algorithm>

- std::find()
- std::lower_bound()
- std::equal_range()

Analyses

- Recognitions
- Tree-Maps
- Hash-Maps

Measurements

- std::find() vs. Associative Containers
- Tree Container vs. lower_bound()
- Associative Containers

<algorithm>

1. std::find()
2. std::lower_bound()
3. std::equal_range()

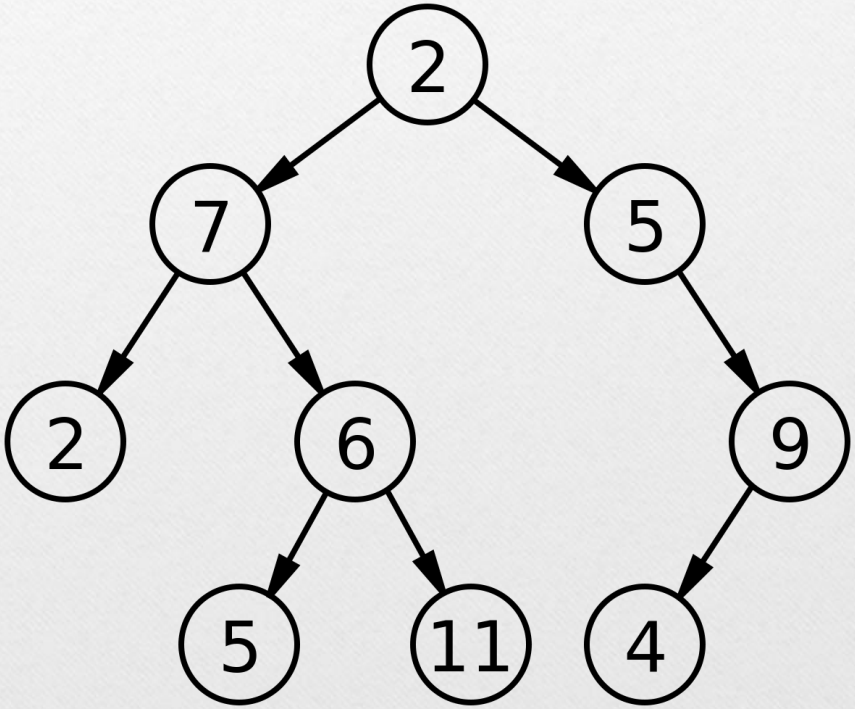
1. std::find()

- 복잡도: **O(N)**
- 원하는 키값을 찾을 때까지
  - 처음부터 끝까지
  - 다 뒤져본다.

1. std::find()

```
template <class Iterator, class T>
Iterator find(Iterator first, Iterator last, const T &val)
{
    // 처음부터 끝까지 싹 뒤져본다.
    for (auto it = first; it != last; ++it)
        if (*it == val)
            return it; // 찾으면 그대로 리턴
    return last; // 못 찾으면 last 리턴
}
```

2. std::lower_bound()



2. std::lower_bound()

- 복잡도: **$O(\log_2 N)$**
- 어느 컨테이너로부터, 특정 value 를 찾는다 할 때,
  - (반드시) **정렬된 컨테이너**를
  - **이진트리**처럼 탐색한다.
- 찾는 데 성공하면, 그대로 리턴
- 못 찾으면 last (end()) 리턴

2. std::lower_bound()

```
template <class Iterator, class T>
Iterator lower_bound(Iterator first, Iterator last, const T& val)
{
    size_t count = distance(first, last);
    while (count > 0) {
        Iterator it = first;
        advance(it, count / 2); // 마치 이진트리처럼, 절반씩 건너뛴다.
        . . .
        first = ++it;
    }
    return first;
}
```

3. std::equal_range()

```
template <class Iterator, class T>
pair<Iterator, Iterator> equal_range
    (Iterator first, Iterator last, const T &val)
{
    return make_pair
    (
        lower_bound(first, last, val),
        upper_bound(first, last, val)
    );
}
```

3. std::equal_range()

find: **2**

lower_bound **upper_bound**

Analyses

1. Recognitions
2. Tree-Maps
3. Hash-Maps

1. Recognitions

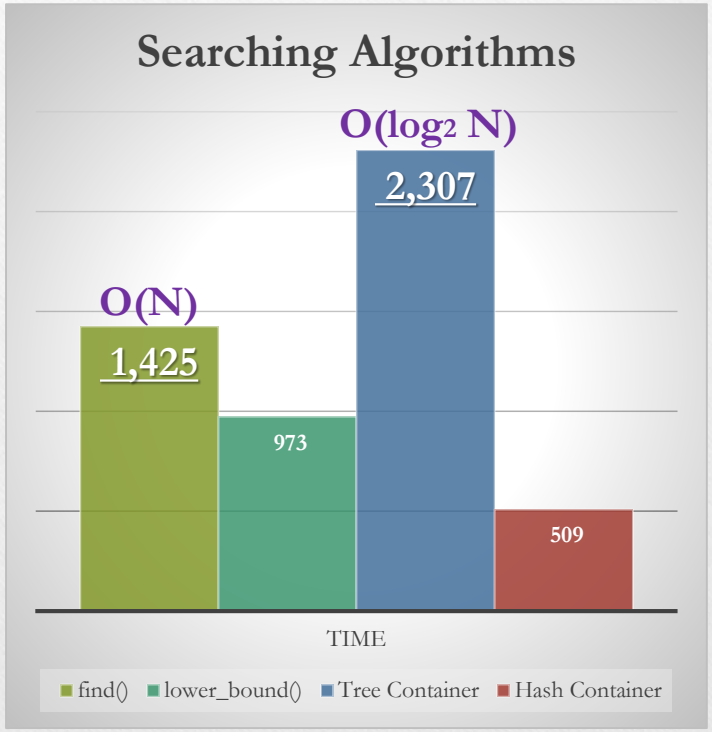
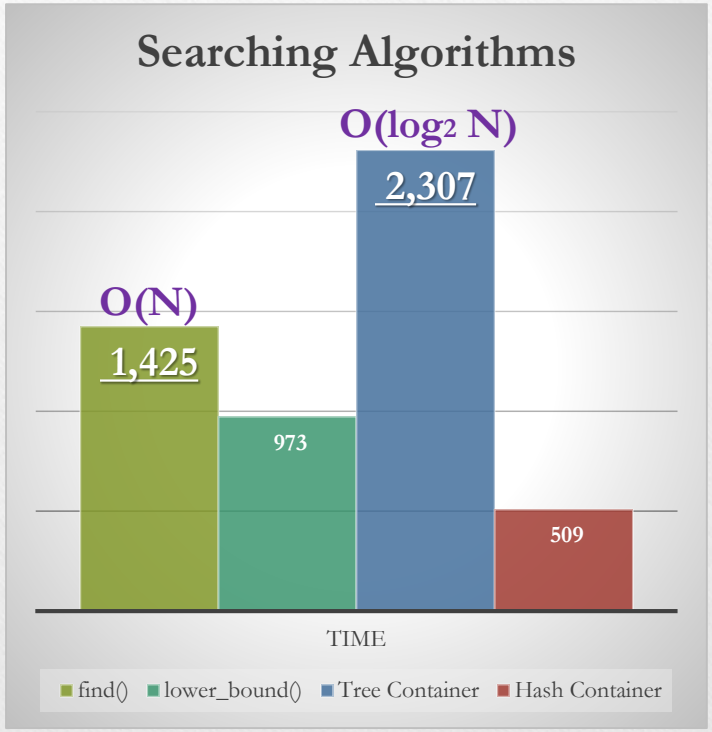


Table 9-1. Summary of search performance experiments

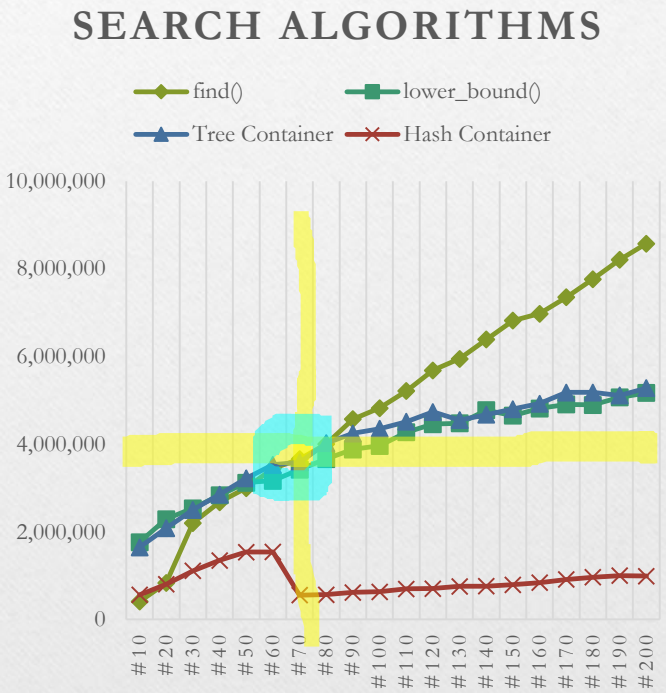
	VS2010 release, i7, 1m iterations	% improvement vs. previous	% improvement vs. category
map<string>	2,307 ms		
map<char*> free function	1,453 ms	59%	59%
map<char*> function object	820 ms	77%	181%
map<char*> lambda	820 ms	0%	181%
std::find()	1,425 ms		
std::equal_range()	1,806 ms		
std::lower_bound	973 ms	53%	86%
find_binary_3way()	771 ms	26%	134%
std::unordered_map()	509 ms		
find_hash()	195 ms	161%	161%

1. Recognitions



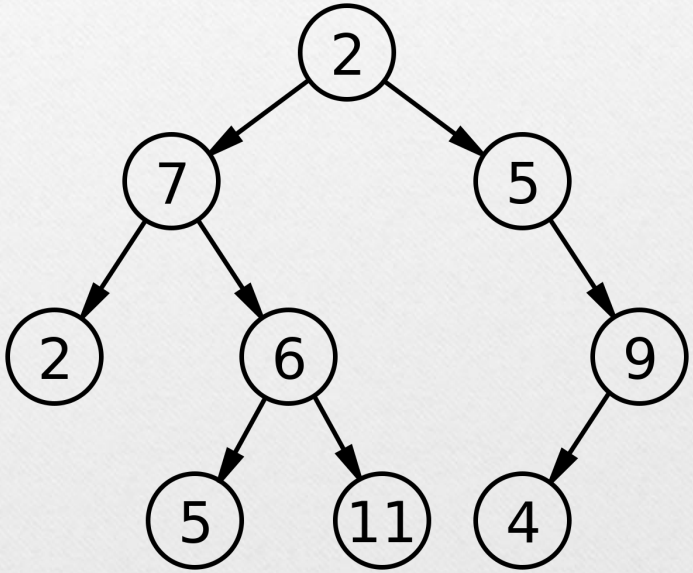
- O(N) < O($\log_2$ N) ?
- 교과서의 측정도표
  - size() := 26 개
- std::map 이
  - Tree Container
- find() 보다 더 느렸다.

1. Recognitions



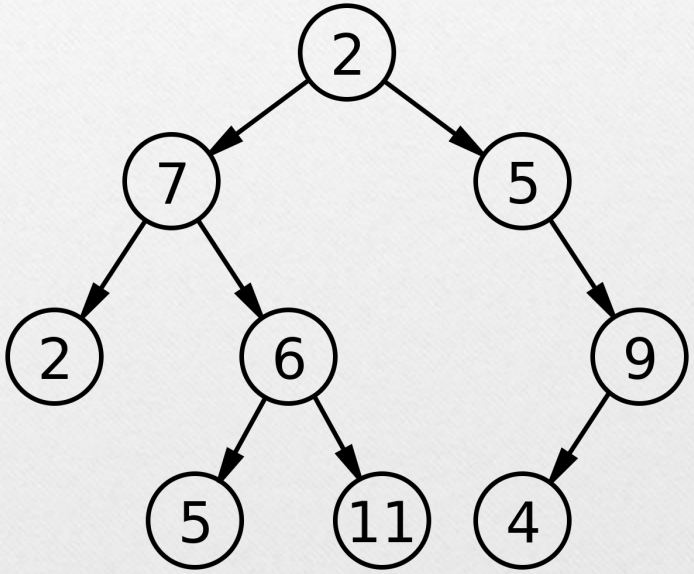
- 자체 측정도표
- size() < **70** 까지는
- find() 가 더 빠르더라.
- 왜 그럴까?
  - 왜 O(N) 이
  - O($\log_2$ N) 보다 빠를까?

2. Tree-Maps



- 왜 std::map 느렸을까?
  - TreeMap
- 그것을 알기 위해
- 트리맵을 파헤쳐보자

2. Tree-Maps




```
template <class Key, class T>
class tree_node
{
    Key key_;
    T value_;

    tree_node *parent_;
    tree_node *left_;
    tree_node *right_;
};
```

2. Tree-Maps

```
template <class Key, class T>
class tree_node
{
    Key key_;
    T value_;

    tree_node *parent_;
    tree_node *left_;
    tree_node *right_;
};
```

- 포인터를 통해
  - 트리노드는
  - 포인터를 통하여
  - 인접 노드로 이동한다.
- 그리고,
  - 포인터 이동은
  - 배열보다 느리다.

2. Tree-Maps

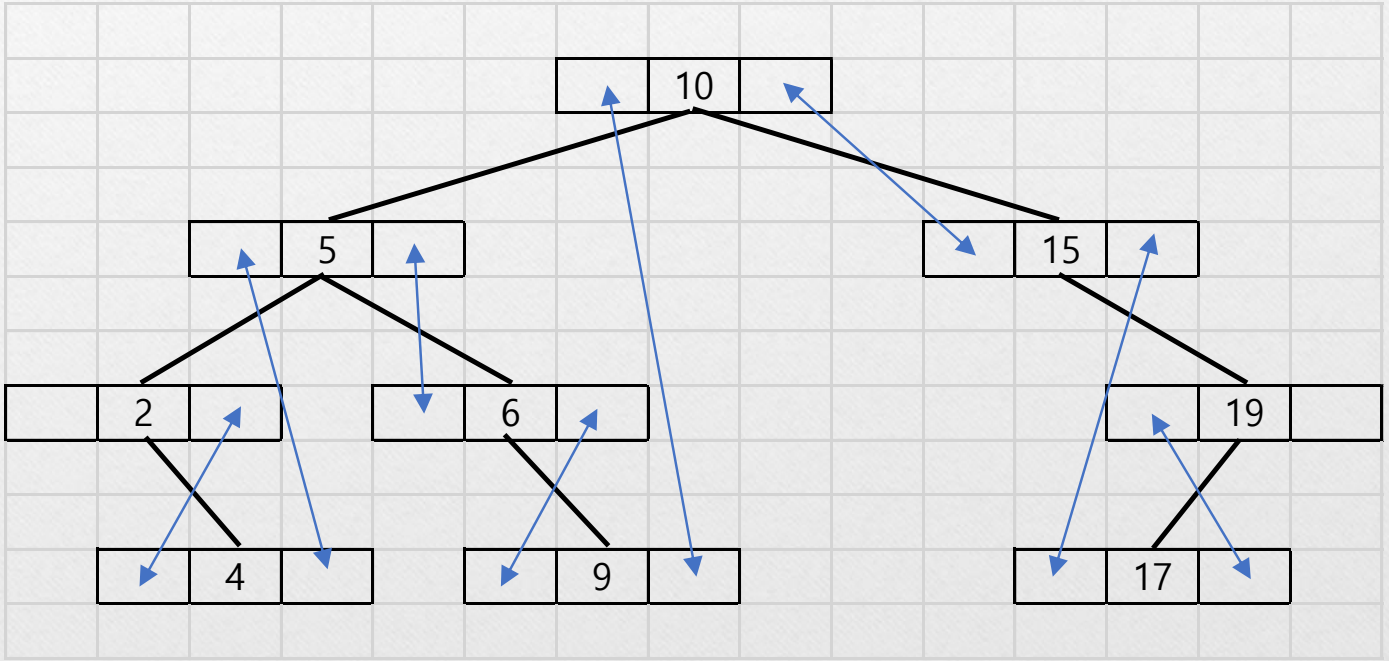
- 실제로 측정해보니
  - 배열이 훨씬 빠르다.
  - 포인터 이동 (리스트)
    - 상상했던 것보다
    - 훨씬 느리더라
- 그래서 std::find
- O(N) 보다 느렸던 것

2. Tree-Maps

- 더 알아보기
  - std::map 에서 쓰는 트리는 약간 다르다.
  - std::map 은 full iteration 이 가능해야 함

```
std::map<int, int> m;

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

2. Tree-Maps



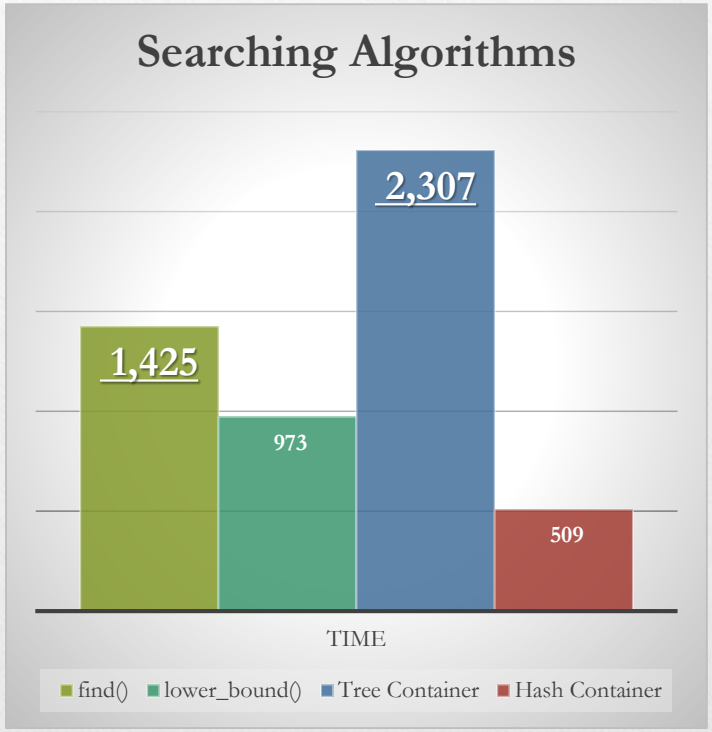
2. Tree-Maps

```
template <class Key, class T>
class map
{
    typedef list<pair<Key, T>>::iterator iterator;

private:
    list<T> data_;
    xtree<Key, *list_node<pair<Key, T>>> tree_;
};
```

2. Tree-Maps

```
template <class Key, class T>
class map_tree_node
{
    Key key_;
    *list_node<pair<Key, T>> value_;

    map_tree_node *parent_;
    map_tree_node *left_;
    map_tree_node *right_;
};
```

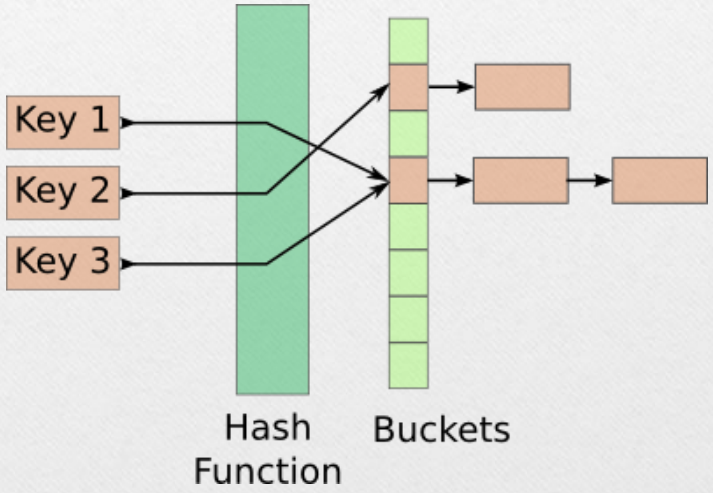
3. Hash-Maps



- 가장 빨랐던
- Hash Container

- 복잡도: **O (1)**
- 원리가 무엇인가?

3. Hash-Maps



- 키값을 입력하면, 어느 버킷에 삽입될 지,
- 그 index를 해쉬 함수를 통해 계산함

- 보통 Buckets은 행렬 형태
- 해쉬 함수를 통한 index 는, 입력된 데이터 (val)을 buckets 의 크기(행)로 나눈 나머지를 취함

- buckets: Vector<List<Entry>>
- index = hash(key) % buckets.size()

3. Hash-Maps

```
template <class T>
size_t hash(const T &obj) { return _Hash((const char *)&obj,
sizeof(T)); }

size_t _Hash(const char *ptr, size_t size)
{
      size_t ret = 2166136261;
      for (const char *it = ptr; it != ptr + size; ++it) {
            ret ^= *it;
            ret *= 16777619;
      }
      return ret;
}
```

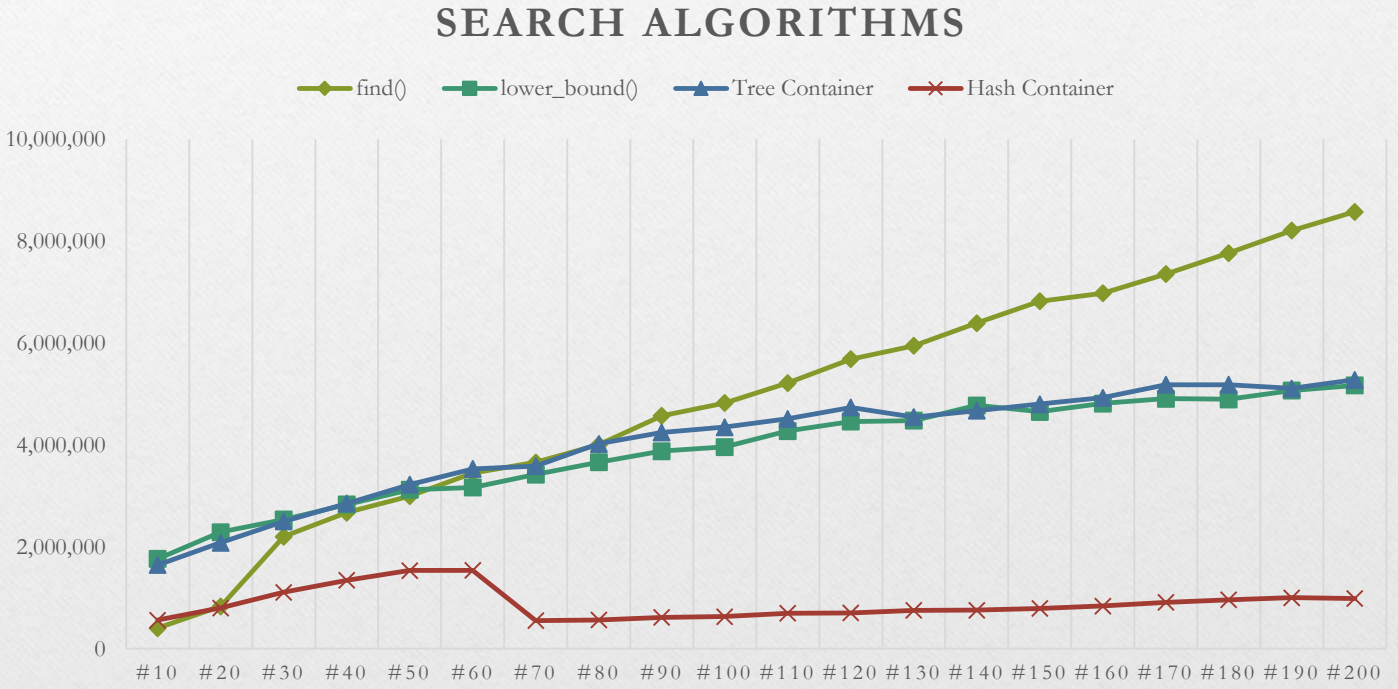
3. Hash-Maps

```
buckets_: vector<list<*list_node<pair<Key, T>>>>;

iterator find(key: Key) const
{
    size_t index = hash(key) % this->buckets_->size();
    auto &bucket = this->buckets_.at(index);

    for (auto node : bucket)
        if (equal_to(node->first, key))
            return iterator(node);
    return this->end();
}
```

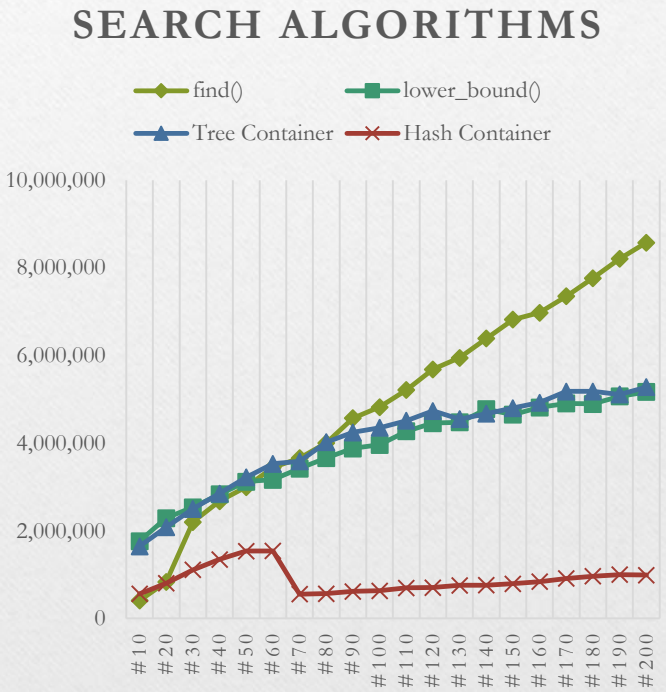
Measurements

1. find() vs. others
2. lower_bound() vs. Tree Container
3. Tree vs. Hash Containers

1. find() vs. others

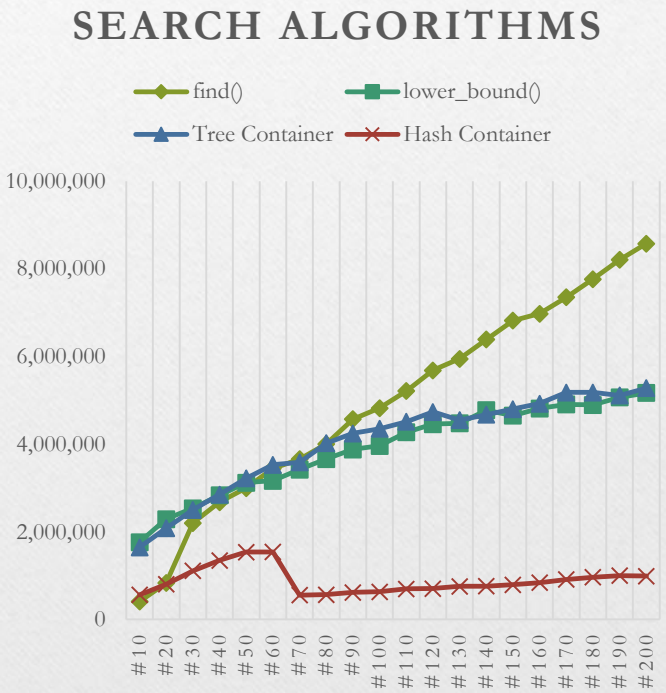


1. find() vs. others



- O(N)
  - std::find()
- O($\log_2$ N)
  - std::lower_bound()
  - Tree Container
- O(1)
  - Hash Container

1. find() vs. others

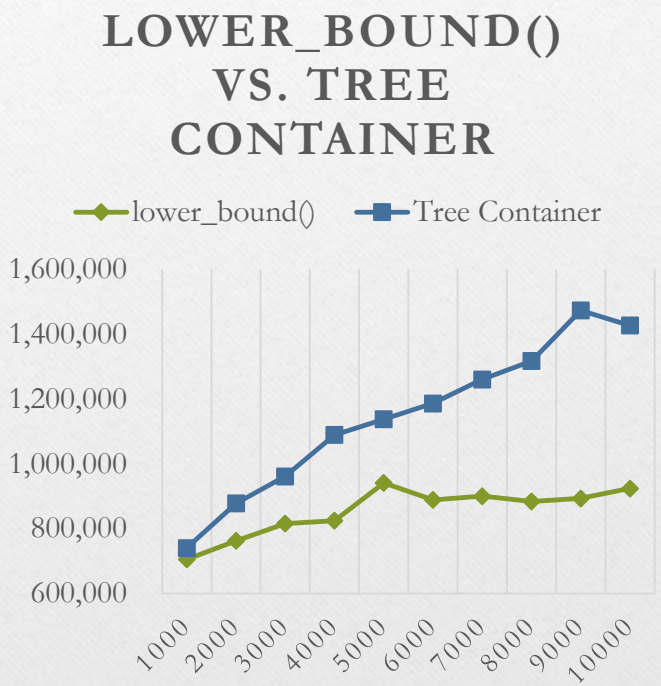


- 일정 수를 기준으로
  - std::find() 보다
  - Tree 가 빨라짐
- 단, HashMap 은
  - 늘 빠르더라

2. lower_bound() vs. Tree Container

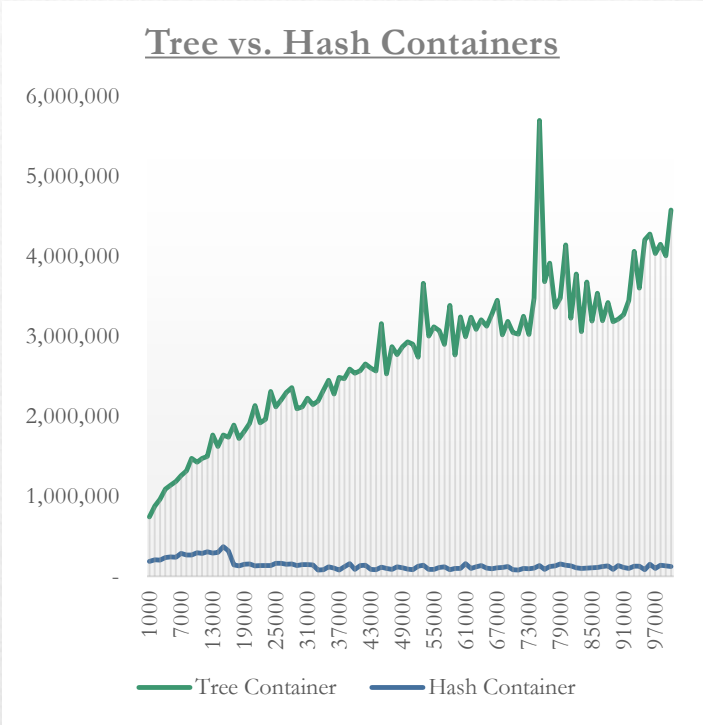
- 원소가 많을수록
- 배열을 사용하면
  - 단, 정렬되어있다.
  - std::lower_bound()
- 유리하지 않을까?

2. lower_bound() vs. Tree Container



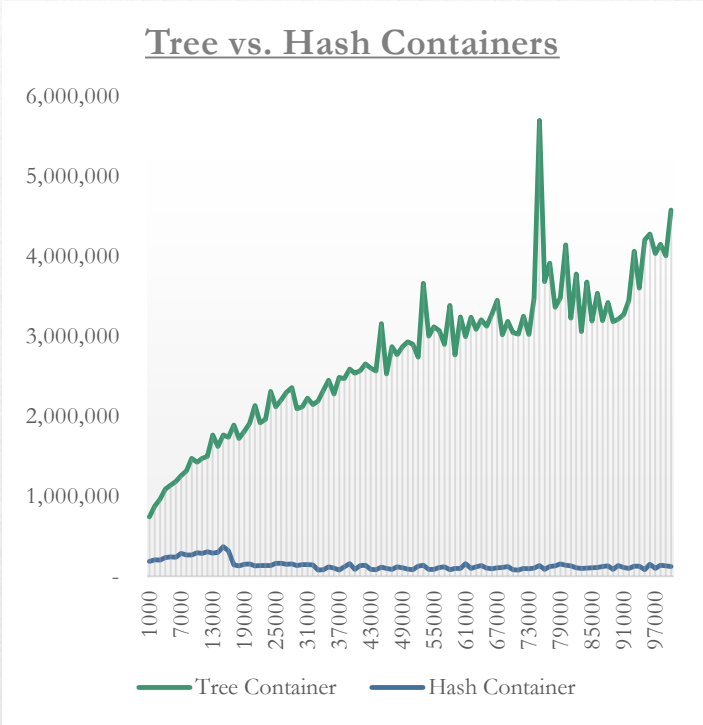
- 그렇다.
- 원소의 수가 많아져
- 트리의 깊이가 깊어질수록
  - 배열은 큰 영향 없음
  - 트리는 순회에 걸리는 시간이 증가함.

3. Tree vs. Hash Containers



- $O(\log_2 N)$ vs. $O(1)$
- 대수의 법칙
  - 단위가 커질수록
  - 본래의 복잡도에 가까운 곡선이 그려진다.

3. Tree vs. Hash Containers



- $O(\log_2 N)$ vs. $O(1)$
- 그리고,
- 단위가 커질수록
  - HashMap 이 강하다.

Q & A

2017-07-01

Jeongho Nam