17 Container ADT II

```
template <typename T>
int set<T>::index_of(T e) const {
  for (size_t i = 0; i < T.size; i++) {
    if (ele[i] == e) {return i;}
  }
  return -1;
}
```

这里 `ele[i]` 和 `e` 的 comparisons 次数：

如果 T.size 为 n，那么 best case 是 1 次（第一个就是）；worst case 是 n 次（直到最后一个才是）

这个 **search algorithm** 的名字就叫 **linear search 或者称 sequential search，因为它的 time complexity 为** $O(n)$

17.1 Measuring the efficiency of an algorithm

1. 确认 problem size: 称为 n.

   比如刚才在 set 中寻找元素，n 就是 set 中的元素数量

2. 选择一个 basic operation：比如 `==` 这个比较

3. 找出 function `C(n)` 用来表示 number of basic operations required for the problem of size **n**

   通常我们选择 worst case. (但是更加 careful 的 algorithm analysis 中 best 和 average case也有其他用处).

$C(n)$ is constant: time complexity is in $O(1)$.

$C(n) = ax + b$: time complexity is in $O(n)$.

$C(n) = log_m n$: time complexity is in $O(log\ n)$

通过我们刚才的分析，**linear search** is in $O(n)$；binary search is in $O(log\ n)$.

17.1.1 binary search

Algorithm idea:

1. Look in the middle position of the array indices under consideration **(err to the left if no perfect middle)**
2. 如果找到了：return the index；否则：**Limit your search to the left or right subarray** as appropriate，然后会回到 step 1.

(如果我们要找的元素不在 array 中？那么就 return -1.)

```
/* Requires: arr is in sorted order (least to greatest)
* size is the size of the array
* Effects: returns the index at which target appears in the
* array, or -1 if not found
*/
int binarySearch(int arr[], int size, int target) {
  int left = 0;
  int right = size - 1;
  while (left <= right) {
    int mid = (left + right) / 2; // binary split!
    if (arr[left] == target) {return mid;}  // found!
    if (arr[mid] < target) {left = mid + 1;}  // not found: move side
    else {right = mid - 1;}
  }
  return -1; // if not found: not in the array
}
```

binary search 的 time complexity 很明显是 $O(log\ n)$，因为我们对 2^10 = 1024 个元素只需要 10 次比较来搜索出想要的元素。

这个 binary search 有一个缺点：必须要 array 是被排序好的才可以运行，否则移动左右侧的步骤没法进行。

17.1.2 binary search 的要求: orderd set

如果我们想使用 binary search，必须要在 orderd set 上。所以原本的 unordered set 并不是理想的 data structure. 现在我们在本来的 unordered set 的基础上加一点代码，做一个 unordered set.

原本我们的 `insert` 只是把新元素加到最后，而现在我们通过查找这个元素按照顺序应该放的位置，并把所有该位置之后的元素后所有元素右移来实现。

这样就可以实现每次我们 `insert` 新元素都是 in order 的，从而 induction 下所有元素都是 in order 的。

```
template <typename T>
void ordered_set<T>::insert(T e) {
  if (contains(e)) return;
  assert (size < CAPACITY);
  int i = size;
  // find supposed order, at the same time
```

```
  // shifting right all elements before finding that
  while (i > 0 && e < ele[i - 1]) {
    ele[i] = ele[i - 1];
    --i;
  }
  // put in the right position (start with nothing so 0)
  ele[i] = e;
  ++size;
}
```

同样的逻辑，我们通过把制定元素位置之后的元素后所有元素左移来实现 `remove`.

```
template <typename T>
void ordered_set<T>::remove(T e) {
  int target = index_of(e);
  if (target == -1) {return;}
  --size;
  while (i < elts_size) {
    elts[i] = elts[i + 1];
    ++i;
  }
}
```

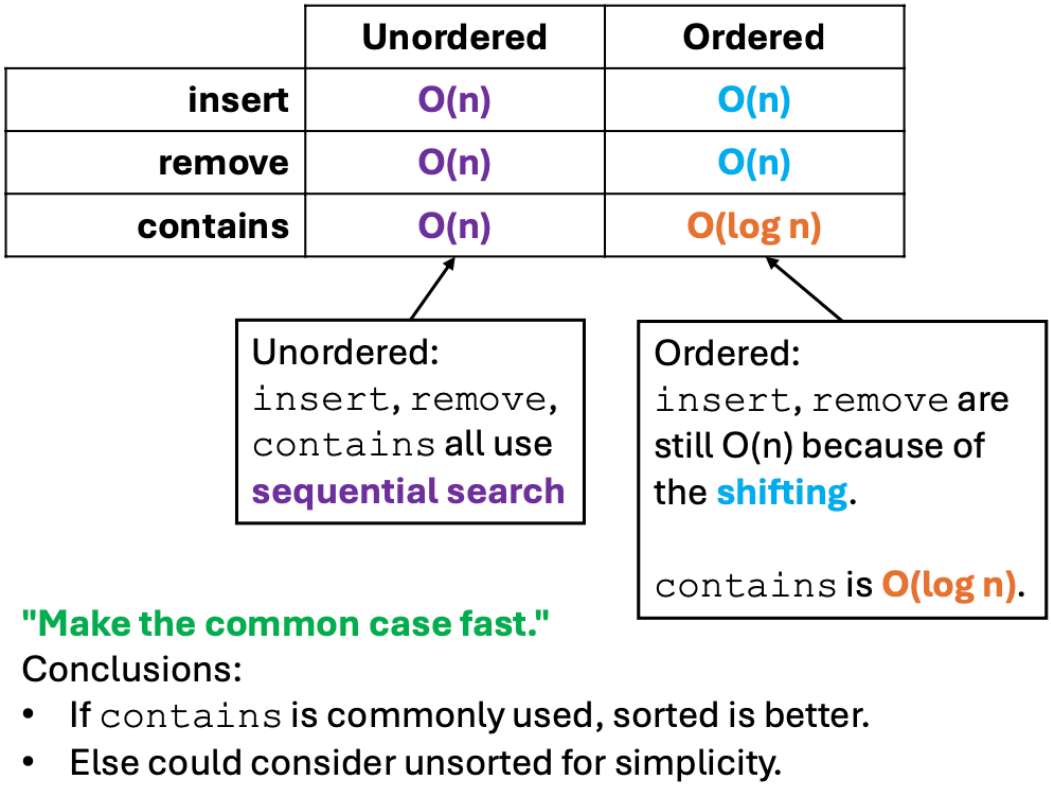
17.1.3 题外话: c++ 的 short-circuited `&&`

这里注意，我们可能会担心 `while (i > 0 && e < elts[i - 1])` 因为 `i-1` out of boundary 而出现 compiler error，但其实不用。因为 c++ 使用 short-circuited `&&`，意思是说当第一个 expression 为 `false` 时，第二个表达式将不被考虑。所以不会有 compiler error.

17.1.4 Efficiency Comparison

我们在 `insert` 和 `remove` 这两个 member function 中加入了一个循环，所以看起来 `ordered set` 的复杂度好像增加了，但其实并没有。

在 unordered set 中，我们 `insert` 和 `remove` 本身就要先使用 `contain` 来 linear search，$C(n) = n$，因而复杂度是 $O(n)$；ordered set 中，我们 `insert` 和 `remove` 的复杂度是 $O(log\ n) + O(n) = O(n)$. 虽然增加了一个循环使得 $C(n) = log_2 n + n$，但是复杂度（worst case）不变. 而 `contain` 的复杂度则大幅下降. 这在处理很大的集合时极大提高了效率。



17.2 Operator Overloading

我们在 search 中，用到了 `>`， `==` 等 comparison operator.

我们可以随便 declare `ordered_set<int>`， `ordered_set<string>` 等，但是我们发现我们如果 declare 一个 `ordered_set<person>`，其中 `person` 是另外一个 class，那么则会出现 compiler error。因为我们并没有定义 `person` objects 的 `==`, `<`, `>` 等符号.

因而我们需要在 `person` 这个 class 中声明 `operator<`, `operator==` 等 functions 为friend fucntion，并在外面对这些 functions 也进行参数是 `person` 的overload. （也可以在 class 中作为 member functions 而不是 friend functions 来定义。）

这些 comparison operator 都是 **binary operator（二元运算符）**，也就是说 operator function 的参数有两个，第一个是 operator 左侧的，第二个是 operator 右侧的。

```
class Person {
private:
  std::string name;
public:
...
  friend bool operator<(const Person &p1, const Person &p2);
  friend bool operator==(const Person &p1, const Person &p2);
};

bool operator<(const Person &p1, const Person &p2) {
  return p1.name < p2.name;
```

```
}
bool operator==(const Person &p1, const Person &p2) {
  return p1.name == p2.name;
}
```

我们可以发现和 return stream 的 operators 不一样的是，这些运算符就不用返回引用值了。因为不用连续运算。

17.3 `main.cpp` using `using`

我们写了两个 set class，如何简便地选择 `main.cpp` 使用哪一个：

选择 unordered_set:

```
#include "Unordered_set.hpp"
#include "Ordered_set.hpp"
//Select one implementation
template <typename T>
using Set = unordered_set<T>;

int main() {
  Set<int> is;
  //...
  Set<string> desserts;
  //...
}
```

选择 ordered_set:

```
#include "Unordered_set.hpp"
#include "Ordered_set.hpp"
//Select one implementation
template <typename T>
using Set = ordered_set<T>;

int main() {
  Set<int> is;
  //...
  Set<string> desserts;
  //...
}
```

17.4 其他事项

17.4.1 Static v.s. dynamic polymorphism

Subtype polymorphism 也就是 dynamic polymorphism. 发生在 run time,

Generic programming with templates 也就是 static polymorphism. 发生于 compile time.

我们在 set 的 implementation 中使用 static polymorphism 是因为 type 不在 runtime 变化.

17.4.2 `check_invariant`

Representation invariants for sets

Unordered_set	Ordered_set
The first `elts_size` members of `elts` contain the integers comprising the set,	The first `elts_size` members of `elts` contain the integers comprising the set,
	from lowest to highest,
with no duplicates.	with no duplicates.

For ordered_set:

```
template <typename T>
bool Ordered_set<T>::check_invariant() const {
  for (int i = 0; i < elts_size - 1; ++i) {
  if (elts[i] >= elts[i+1])
    return false;
  }
  return true;
}
```

For unordered_set:

```
template <typename T>
bool Unordered_set<T>::check_invariant() {
  for (int i = 0; i < elts_size; ++i) {
    for (int j = i+1; j < elts_size; ++j) {
      if (elts[i] == elts[j])
        return false;
    }
  }
  return true;
}
```

我们发现个得个在 `check_invariant()` 中便用 nested loop. 这便得桯序变得很慢. 这在testing 中是值得的, 但是在 final product 中不应该用。

我们可以在程序中禁用这些 asserts，就是碰到这些代码就忽略。方法是 **compiling with the `NDEBUG` preprocessor.**

variable defined.

有两个办法做到：

1. 在 include `<cassert>` 之前加上 `NDEBUG` 的宏。

   ```
   #define NDEBUG
   #include <cassert>
   ```

2. 在 command line 写明我们包括 `NDEBUG` 进行编译.

   ```
   g++ -DNDEBUG...
   ```