

1. Strict weak ordering rule

In order to expect correct results from STL C++ algos, set of rules are defined for data-type of sequence and comparators used. <https://medium.com/@shiansu/strict-weak-ordering-and-the-c-stl-f7dcfa4d4e07>

Strict Weak Ordering

Having a non-circular relationship is called non-transitivity for the `<` operator. It's not too hard to realise that if your relationships are circular then you won't be getting reasonable results. In fact there is a very strict set of rules that a data type and its comparators must abide by in order to get correct results from C++ STL algorithms, that is **strict weak ordering**. In general we need only define the `<` operator and we will get `>` and `==` for free because:

- `a > b` is equivalent to `b < a`
- `a == b` is equivalent to `!(a < b) && !(b < a)`

Then for strict weak ordering we must have

- For all `x`: `x < x` is never true, everything should be equal to itself
- If `x < y` then `y < x` cannot be true
- If `x < y` and `y < z` then `x < z`, the ordering should be transitive
- If `x == y` and `y == z` then `x == z`, equality should be transitive

2. Algo functions less efficient in general compared to member methods

Eg

```
unordered_set<int> s = {2,4,1,8,5,9}; // Hash table
unordered_set<int>::iterator itr;
```

```
// Using member function
itr = s.find(4); // O(1)
```

```
// Using Algorithm
itr = find(s.begin(), s.end(), 4); // O(n)
```

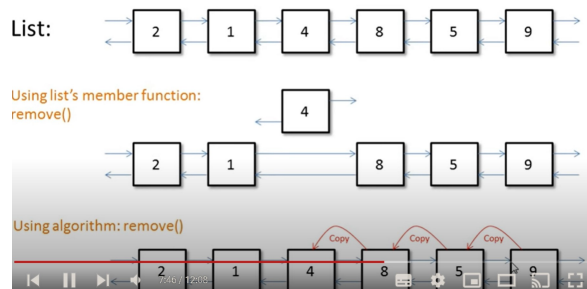
// How about map/multimap?

```
map<char, string> mymap = {{'S', "Sunday"}, {'M', "Monday"}, {'W', "Wednesday"},
..};
```

```
// Using member function
itr = mymap.find('F'); // O(log(n))
```

```
// Using Algorithm
itr = find(mymap.begin(), mymap.end(), make_pair('F', "Friday")); // O(n)
```

Removal of List Items



3. `std::all_of(itr1, itr2, unary_predicate)` -> true if `unary_predicate` returns 1/true for all elements in sequence
`std::any_of` is similar.. Range is `[itr1, itr2)`

`O(n)`

4. `std::binary_search(itr1, itr2, value, comparator)` -> true if value is present else false

Uses `<` (uses relational equation above to find equivalence) or `comparison function`

Expectation is sequence is already sorted using `<` or `comparator`.

`Comparator` can be `function pointer` returning `bool`

`itrmatch = std::find(itr1, itr2, value)`

`itrmatch = std::find_if(itr1, itr2, pred)`

`oitrmatch = std::search(itr1, itr2, oitr1, oitr2, pred)` .. Search for first occurrence of `oitrmatch` in `itr1-itr2`.. Equivalence using `==` or `pred`

5. `oitrmatch = std::copy(itr1, itr2, oitr1)`

`I`

6. `oitrmatch = std::copy_if(itr1, itr2, oitr1, unary_pred)`.. copies elements for which `unary_pred` is true

7. `oitrmatch = std::copy_n(itr1, n, oitr1)` .. Copies `n` elements from `itr1` to `oitrmatch`

`I`

8. `count = std::count(itr1, itr2, val)`.. Uses `==` to compare

`count = std::count_if(itr1, itr2, val, pred)`

9. `bool std::equal(itr1, itr2, jitr1, pred)` .. Compares `itr1-itr2` range to `jitr1-range` of elements using `==` or `predicate`

`I`

10. `std::fill(itr1, itr2, val)`, `std::fill_n(itr1, n, val)`

`I`

11. `std::for_each(itr1, itr2, funcptr)` .. For each element in sequence execute `funcptr`

`I`

12. `std::unique(itr1, itr2)`.. Removes all but first element of equal valued groups

`I`

13. `std::swap(i, j)` .. or `std::swap<type, size>(refofseq1, refofseq2)`

`I`

14. F

`I`

15.

F	
---	--

`I`

16. F

`I`

17. F

18. F

`I`

19. F

`I`

20. F

`I`

21. F

`I`

22. F

`I`

23. F

`I`

- 24. D
- 25. F
- 26. F
- 27. F
- 28. F
- 29. F
- 30. F
- 31. F
- 32. F
- 33. F
- 34. F
- 35. D
- 36. F
- 37. F
- 38. F
- 39. F
- 40. F
- 41. F
- 42. F
- 43. F
- 44. F