

Алгоритмизация и программирование

Лекция 5

Последовательные контейнеры

array<T, size>

fixed-size array

#include <array>

```
std::array<int,6> a {1,2,3,4,5,6};  
cout << a.size(); // 6  
cout << a[2]; // 3  
a[0] = 7; // 1st element => 7
```



contiguous memory; random access; fast linear traversal

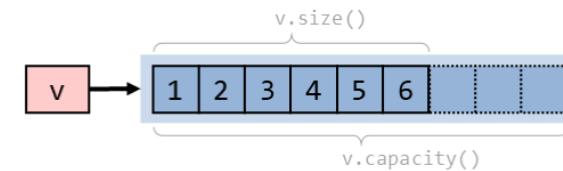
vector<T>

dynamic array

C++'s "default" container

#include <vector>

```
std::vector<int> v {1,2,3,4,5,6};  
v.reserve(9);  
cout << v.capacity(); // 9  
cout << v.size(); // 6  
v.push_back(7); // appends '7'  
v.insert(v.begin(), 0); // prepends '0'  
v.pop_back(); // removes last  
v.erase(v.begin() + 2); // removes 3rd  
v.resize(20, 0); // size => 20
```



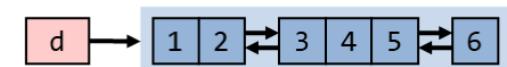
contiguous memory; random access;
fast linear traversal; fast insertion/deletion at the ends

deque<T>

double-ended queue

#include <deque>

```
std::deque<int> d {1,2,3,4,5,6};  
// same operations as vector  
// plus fast growth/deletion at front  
d.push_front(-1); // prepends '-1'  
d.pop_front(); // removes 1st
```



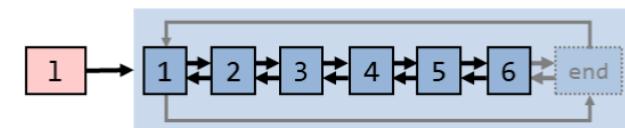
fast insertion/deletion at both ends

list<T>

doubly-linked list

#include <list>

```
std::list<int> l {1,5,6};  
std::list<int> k {2,3,4};  
// O(1) splice of k into l:  
l.splice(l.begin() + 1, std::move(k))  
// some special member function algorithms:  
l.reverse();  
l.sort();
```



fast splicing; many operations without copy/move of elements

forward_list<T>

singly-linked list

#include <forward_list>

```
std::forward_list<int> fl {2,2,4,5,6};  
fl.erase_after(begin(fl));  
fl.insert_after(begin(fl), 3);  
fl.insert_after(before_begin(fl), 1);
```



lower memory overhead than std::list; only forward traversal

deque

Construct A New Deque Object

```
deque<int> d1 {2, 9, 1, 8, 5, 4} → [2 9 1 8 5 4]
deque<int> d2 (begin(d1)+3, end(d1)) → [8 5 4]
deque<int> d3 (5, 3) → [3 3 3 3 3]
deque<int> deep_copy_of_d1 (d1) → [2 9 1 8 5 4]
```

C++17 value type deducible from argument type

```
deque d4 {7, 4, 2}; // deque<int>
```

Typical Memory Layout

Note that the ISO standard only specifies the properties of deque (e.g., constant-time insert at both ends) but not how that should be implemented.

dynamically allocated contiguous chunks

Obtain Iterators

$\mathcal{O}(1)$ Random Incrementing

```
[0 1 2 3].begin() → @first
[0 1 2 3].end() → @one_behind_last
```

Obtain Reverse Iterators

$\mathcal{O}(1)$ Random Incrementing

```
[0 1 2 3].rbegin() → rev@last
[0 1 2 3].rend() → rev@one_before_first
```

d.begin() **d.end()**

d.rbegin() **d.rbegin()**

$@pos = rev@pos.base() - 1$

$rev@pos.base()$

Assign New Content To An Existing Deque

```
deque<int> d1 {8,5,3};
deque<int> d2 {6,8,1,9};
d1 = d2; new state of d1
```

[8 5 3] = [6 8 1 9] → [6 8 1 9]
[8 5 3].assign({4, 1, 3, 5}) → [4 1 3 5]
[8 5 3].assign(2, 1) → [1 1]
[8 5 3].assign(@InBegin, @InEnd) → [2 1 1 2]

source container [3 2 1 1 2 3]

Query Size (= Number of Elements) $\mathcal{O}(1)$

```
[8 5 3].empty() → false
[8 5 3].size() → 3
```

Change Size $\mathcal{O}(|n - newSize|)$

```
[8 5 3].resize(2) → [8 5]
[8 5 3].resize(4, 1) → [8 5 3 1]
[8 5 3].resize(6, 1) → [8 5 3 1 1 1]
[8 5 3].clear() → []
```

Append Elements $\mathcal{O}(1)$

```
[8 5 3].push_back(7) → [8 5 3 7]
```

Prepend Elements $\mathcal{O}(1)$

```
[8 5 3].push_front(7) → [7 8 5 3]
```

Insert Elements at Arbitrary Positions $\mathcal{O}(n)$ Worst Case

```
[8 5 3].insert(begin(d)+1, 7) → [8 7 5 3]
[8 5 3].insert(begin(d)+1, 3, 7) → [8 7 7 7 5 3]
[8 5 3].insert(begin(d)+1, {6,9,7}) → [8 6 9 7 5 3]
[8 5 3].insert(begin(d)+1, @InBegin, @InEnd) → [8 1 8 9 5 3]
```

source container [3 1 8 9 2 3]

Get Element Values $\mathcal{O}(1)$ Random Access

```
[2 8 5 3][1] → 8
[2 8 5 3].front() → 2
[2 8 5 3].back() → 3
```

Erase Elements At The Ends $\mathcal{O}(1)$

```
[4 8 5 6].pop_back() → [4 8 5]
[4 8 5 6].pop_front() → [8 5 6]
```

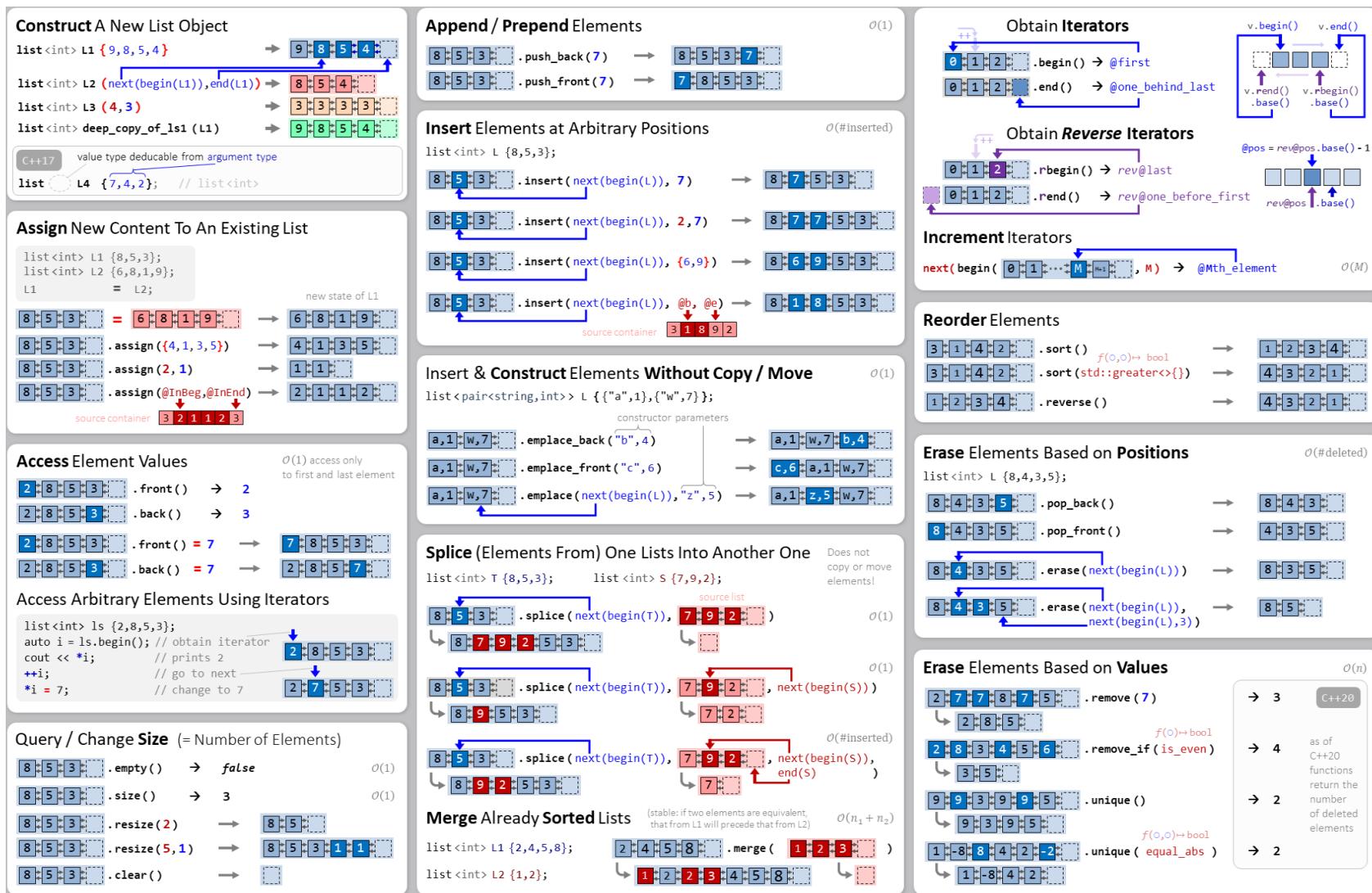
Erase Elements At Arbitrary Positions $\mathcal{O}(n)$ Worst Case Complexity

```
[4 8 5 6].erase(begin(d)+2) → [4 8 6]
[4 8 5 6].erase(begin(d)+1, begin(d)+3) → [4 6]
```

Insert & Construct Elements in Place $\mathcal{O}(n)$ Worst Case

```
deque<pair<string,int>> d {{"a",1}, {"w",7}};
[a,1]{w,7}.emplace_back("b",4) → [a,1]{w,7}{b,4}
[a,1]{w,7}.emplace_front("c",6) → [c,6]{a,1}{w,7}
[a,1]{w,7}.emplace(begin(d)+1, "z", 5) → [a,1]{z,5}{w,7}
```

list



forward_list

Construct A New List Object

```
forward_list<int> L1 { 8,5,4 }
```

Assign New Content To An Existing List

```
forward_list<int> L1 {8,5,3};  
forward_list<int> L2 {6,8,1,9};  
L1 = L2;
```

new state of L1

Access Element Values

```
L1.front() → 2  
L1.front() = 7 → 7 8 5 3
```

○(1) access only to first element

Access Arbitrary Elements Using Iterators

```
forward_list<int> ls {2,8,5,3};  
auto i = ls.begin(); // obtain iterator  
cout << *i; // prints 2  
++i; // go to next  
*i = 7; // change to 7
```

Check Emptiness / Change Size (= Number of Elements)

There's no member function available to determine the size!

```
L1.empty() → false  
L1.resize(2) → 8 5  
L1.resize(5,1) → 8 5 3 1 1  
L1.clear() → 
```

○(1)

Insert Elements at Arbitrary Positions

```
forward_list<int> L { {0,1,2} };
```

Insert & Construct Elements Without Copy / Move

```
forward_list<pair<string,int> > L { {"a",1}, {"w",7} };  
L.emplace_front("c",6);  
L.emplace_after(begin(L), "z",5);
```

constructor parameters

Splice (Elements From) One List Into Another One

```
forward_list<int> T {8,5,3}; forward_list<int> S {7,9,2};
```

Merge Already Sorted Lists

```
forward_list<int> L1 {2,4,5}; forward_list<int> L2 {1,3};
```

Erase Elements Based on Positions

```
forward_list<int> L {0,1,2};
```

Erase Elements Based on Values

```
L.remove(7);  
L.remove_if(is_even);  
L.unique();  
L.unique(equal_abs);
```

Obtain Iterators

A singly-linked list of nodes each holding one value.

C++11

```
begin() → @first  
before_begin() → @one_before_first  
end() → @one_behind_last
```

○(1)

Increment Iterators

```
next(begin([0 1 2]), M) → @Mth_element
```

○(M)

Prepend Elements

```
L.push_front(7) → 7 8 5 3
```

○(1)

Reorder Elements

```
L.sort();  
L.sort(std::greater<>());  
L.reverse();
```

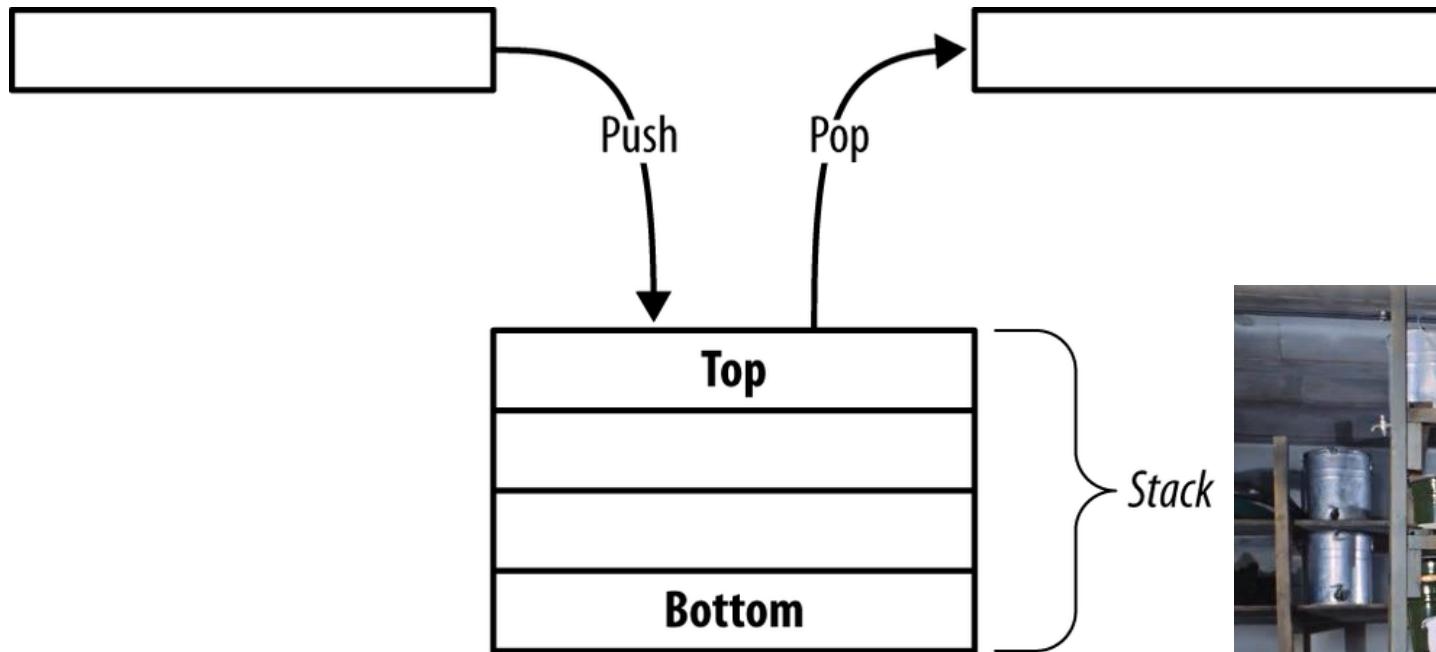
Erase Elements Based on Positions

```
L.pop_front();  
L.erase_after(L.before_begin());  
L.erase_after(begin(L));  
L.erase_after(begin(L), end(L))
```

○(n)

○(#deleted)

Stack



Ассоциативные контейнеры

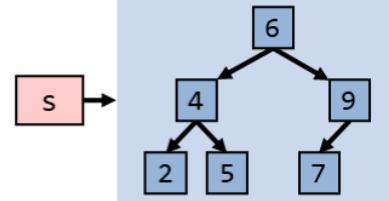
set<Key>

unique, ordered keys

multiset<K>

(non-unique) ordered keys

```
std::set<int> s;  
s.insert(7); ...  
s.insert(5);  
auto i = s.find(7); // → iterator  
if(i != s.end()) // found?  
    cout << *i; // 7  
if(s.contains(7)) {...} C++20
```



usually implemented as balanced binary tree (red-black tree)

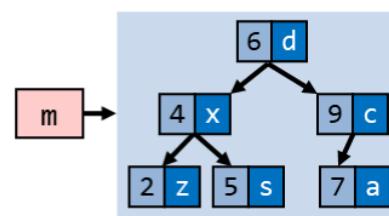
map<Key, Value>

unique key → value-pairs; ordered by keys

multimap<K, V>

(non-unique) key → value-pairs, ordered by keys

```
std::map<int,char> m;  
m.insert({7,'a'}); ...  
m[4] = 'x'; // insert 4 → x  
auto i = s.find(7); // → iterator  
if(i != s.end()) // found?  
    cout << i->first // 7  
        << i->second; // a  
if(s.contains(7)) {...} C++20
```



usually implemented as balanced binary tree (red-black tree)

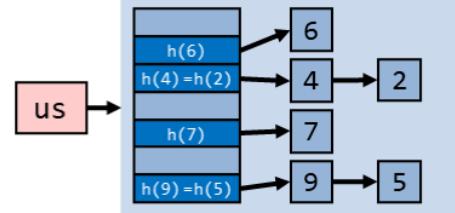
unordered_set<Key>

unique, hashable keys

unordered_multiset<Key>

(non-unique) hashable keys

```
std::unordered_set<int> us;  
us.insert(7); ...  
us.insert(5);  
auto i = us.find(7); // → iterator  
if(i != us.end()) // found?  
    cout << *i; // 7  
if(s.contains(7)) {...} C++20
```



hash table for key lookup, linked nodes for key storage

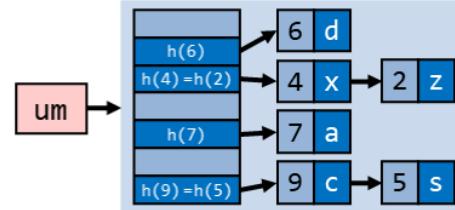
unordered_map<Key, Value>

unique key → value-pairs; hashed by keys

unordered_multimap<Key, Value>

(non-unique) key → value-pairs; hashed by keys

```
unordered_map<int,char> um;  
um.insert({7,'a'}); ...  
um[4] = 'x'; // insert 4 → x  
auto i = um.find(7); // → iterator  
if(i != um.end()) // found?  
    cout << i->first // 7  
        << i->second; // a  
if(s.contains(7)) {...} C++20
```



hash table for key lookup, linked nodes for (key, value) pair storage

set

std::multiset<KeyType, Compare> (multiple equivalent keys)

Construct A New Set Object

```
set<int> s0 {} → []
set<int> s1 {2,1,8,5,4} → [1 2 4 5 8]
set<int> s2 (begin(s1)+2, end(s1)) → [4 5 8]
set<int> deep_copy_of_s1(s1) → [1 2 4 5 8]
set s3 {7,2,4}; // set<int> [2 4 7]
```

C++17 key type deducible from argument type

Assign New Content To An Existing Set

```
set<int> s1 {1,3,5,7};
set<int> s2 {4,6,8};
s1 = s2; new state of s1
[1 3 5 7] = [4 6 8] → [4 6 8]
```

(deep copy from source)

Key Lookup

```
[1 3 5 7].contains(2) → false C++20
[1 3 5 7].contains(5) → true

[1 3 5 7].count(2) → 0 can be > 1
[1 3 5 7].count(5) → 1 only for std::multiset

[1 3 5 7].find(2) → @end (= no match)
[1 3 5 7].find(5) → @match

[1 3 5 7].lower_bound(3) → @first_not_smaller
[1 3 5 7].upper_bound(3) → @first_greater
[1 3 5 7].equal_range(3) → {@Lbound, @Ubound}
```

Insert A Single Key

```
[2 5 8].insert(4) → {@inserted, true} O(log n)
[2 5 8].insert(5) → {@blocking, false} (not inserted)
[2 5 8].insert(@hint, 7) → {@inserted_or_block}
```

potential performance benefit by hinting at probable insert position

Insert Multiple Keys

```
[2 5 8].insert({1,6,8}) → [1 2 5 6 8]
[2 5 8].insert(@inB, @inE) → [2 3 5 6 8]
source container [3 2 6 2] O(#inserted · log n)
```

Insert & Construct A Key in Place

```
set<pair<int,int>> s {{1,3},{5,6}};
[1,3][5,6].emplace(4,7) → {@inserted, true} O(log n)
[1,3][5,6].emplace_hint(@hint, 4,7) → @inserted
pair constructor arguments
```

potential performance benefit by hinting at probable insert position

Erase One Key or A Range of Keys

```
[1 3 5 7].erase(2) → 0 O(log n)
[1 3 5 7].erase(5) → 1 #erased
[1 3 5 7].erase(@pos) → @after_erased O(1) amortized
[1 3 5 7].erase(@beg, @end) → @after_erased O(log n + #erased)
```

Query Size (= Number of Keys)

```
[2 4 5].empty() → false
[2 4 5].size() → 3
```

Erase All Keys

```
[2 4 5].clear()
```

Obtain Iterators

```
v.begin() → @first
v.end() → @one_behind_last
```

Obtain Reverse Iterators

```
v.rbegin() → rev@last
v.rend() → rev@one_before_first
```

Extract Nodes

Allows efficient key modification and transfer of keys between different set objects.

```
[1 5 7].extract(5) → 5 O(log n)
[1 5 7].extract(@pos) → 5 O(1)
```

Merge Two Sets

```
set<int> s1 {1,3,5,7};
set<int> s2 {2,5,8};
[1 3 5 7].merge([2 5 8]) → [1 2 3 5 7 8] O(n2 · log(n1 + n2))
```

Insert Nodes

```
[1 7].insert(5) → {@position.inserted.node} members of the return type
[1 7].insert(1) → {@position.inserted.node}
[1 7].insert(@hint, 5) → @inserted
[1 7].insert(@hint, 1) → @blocking
```

Modify Key

Direct modification not allowed! Instead: extract key, modify its value and re-insert.

```
set<int> s {1,5,7};
auto node = s.extract(5);
node.value() = 8;
s.insert(std::move(node));
```

map

Construct A New Map Object

```
map<int, string> m0 {} → []
map<int, string> m1 {{4,"Z"}, {2,"A"}, {7,"Y"}} → [2 A 4 Z 7 Y]
map<int, string> m2 { begin(m1), end(m1) } → [4 Z 7 Y] []
map<int, string> deep_copy_of_m1(m1) → [2 A 4 Z 7 Y]
C++17 key and mapped types deducible from arguments
map m3 {{2, 3.14}, {5, 6.0}}; // map<int, double>
```

Assign New Content To An Existing Map

```
map<int, string> m1 {{2,"X"}};
map<int, string> m2 {{1,"A"}, {4,"G"}};
m1 = m2; new state of m1
[2 X] = [1 A 4 G] → [1 A 4 G]
```

Lookup Using Keys as Input

```
map<int, string> m {{3,"A"}, {5,"X"}, {1,"F"}};
[1 F 3 A 5 X].contains(2) → false C++20
[1 F 3 A 5 X].contains(5) → true
[1 F 3 A 5 X].count(2) → 0 can be > 1 only for std::multimap
[1 F 3 A 5 X].count(5) → 1
[1 F 3 A 5 X].find(2) → @end (=no match)
[1 F 3 A 5 X].find(5) → @match
[1 F 3 A 5 X].lower_bound(3) → @1st_not_smaller
[1 F 3 A 5 X].upper_bound(3) → @1st_greater
[1 F 3 A 5 X].equal_range(3) → @Lower, @Upper
[1 F 3 A 5 X].at(3) → "A"
[1 F 3 A 5 X].at(2) → Throws Exception std::out_of_range
Query Size (= number of key-value pairs)
[1 F 3 A].empty() → false
[1 F 3 A].size() → 2
```

Insert A Single Key-Value Pair

 $\mathcal{O}(\log n)$

```
[1 F 3 A].insert({2,"W"}) → {@inserted, true}
[1 F 3 A].insert({3,"X"}) → {@blocking, false}
potential performance benefit by hinting at probable insert position
[1 F 3 A].insert(@hint, {2,"W"}) → @ins@block
```

Obtain Iterators

```
v.begin() → @first
v.end() → @one_behind_last
v.rend() → @rev@last
v.rbegin() → rev@one_before_1st
```

Obtain Reverse Iterators

```
v.begin() → @first
v.end() → @one_behind_last
v.rend() → @rev@last
v.rbegin() → rev@one_before_1st
```

Extract Nodes

Allows efficient transfer of key-value pairs.

```
[1 F 2 R 3 A].extract(2) → [2 R] O(log n)
[1 F 2 R 3 A].extract(@pos) → [2 R] O(1)
```

Access / Modify Value

 $\mathcal{O}(\log n)$

```
map<int, string> m {{1,"F"}, {3,"A"}};
[1 F 3 A][3] → "A"
[1 F 3 A][3] = "X" → [1 F 3 X]
Attention: [k] inserts new pair if key k is not present!
```

Insert & Construct Key-Value Pair

 $\mathcal{O}(\log n)$

```
[1 F 3 A].emplace(2,"W") → {@inserted, true}
[1 F 3 A][2] → ""
potential performance benefit by hinting at probable insert position
[1 F 3 A].emplace_hint(@int, 2,"W") → @inserted
[1 F 3 A].try_emplace(2,"W") → {@inserted, true}
C++17 advantage: does not move from rvalue input parameters if not inserted
```

Insert or Assign Value

 $\mathcal{O}(\log n)$

```
[1 F 3 B].insert_or_assign(3,"X") → {@as, false}
[1 F 3 B].insert_or_assign(5,"R") → {@ins, true}
[1 F 3 B].insert_or_assign(@hint, 3,"W") → @as
[1 F 3 B].insert_or_assign(@hint, 2,"G") → @ins
O(1) amortized
```

Erase Key-Value-Pair(s)

 $\mathcal{O}(\log n)$

```
[1 F 3 A 5 X].erase(2) → 0 #erased
[1 F 3 A 5 X].erase(3) → 1 #erased
[1 F 3 A 5 X].erase(@pos) → @after_erased
O(log n + #deleted)
[1 F 3 A 5 X].erase(@b, @e) → @after_erased
```

Merge Two Maps

 $\mathcal{O}(n_2 \cdot \log(n_1 + n_2))$

```
map<int, string> m1 {{1,"F"}, {3,"S"}, {5,"T"}};
map<int, string> m2 {{2,"A"}, {5,"X"}};
[1 F 3 S 5 T].merge([2 A 5 X]) → [1 F 2 A 3 S 5 T] [5 X]
```

Modify Key

```
map<int, string> m {{1,"F"}, {3,"A"}};
auto node = m.extract(3);
[3 A] → [1 F 3 A]
if (node) { // if key existed
    node.key() = 8;
    m.insert(move(node));
}
[1 F 8 A]
```

Direct key modification not allowed!

Instead:

- extract
- modify
- re-insert

unordered_set

Construct A New Set Object

```
unordered_set<int> s0 {}
```

Assign New Content To An Existing Set

```
unordered_set<int> s1 {1,5,3,7};  
unordered_set<int> s2 {8,4,6};  
s1 = s2;
```

Key Lookup

```
1 5 3 7 .contains(2) → false  
1 5 3 7 .contains(3) → true  
1 5 3 7 .count(2) → 0 can be > 1 only for std::unordered_multiset  
1 5 3 7 .count(3) → 1  
1 5 3 7 .find(2) → @end (= no match)  
1 5 3 7 .find(3) → @match  
1 5 3 7 .equal_range(5) → {@first_equal, @after}
```

Query Size (= Number of Keys)

```
1 5 3 .empty() → false  
1 5 3 .size() → 3
```

Obtain Iterators (to keys)

```
3 1 5 .begin() → @first  
3 1 5 .end() → @one_behind_last
```

Insert A Single Key

$O(1)$ average, $O(n)$ worst case

```
2 8 5 .insert(4) → .first {@inserted, true}  
2 8 5 .insert(8) → {@blocking, false} (not inserted)  
2 8 5 .insert(@hint, 7) → @inserted_or_block
```

potential performance benefit by hinting at probable insert position

Insert Multiple Keys

$O(\#ins)$ avg., $O(n \cdot \#ins + \#ins)$ worst

```
2 8 5 .insert({1,6,8}) → 6 2 5 1 8  
2 8 5 .insert(@inB, @inE) → 2 4 5 3 8  
source container 3 3 2 4 2
```

Query & Control Hash Table Properties

hash function: $h(key) \rightarrow \text{bucket index}$

```
unordered_set<char> us {'E','X','Z','A','F','B'};  
US → 6 h(A) → A → .begin(4) → @first_in_bucket  
bucket index 5 h(E)=h(Z) → E → Z → .end(4) → @one_behind_last_in_bucket  
4 h(B) → B  
3 h(X)=h(F) → X → F → .bucket_size(4) → 2  
2 .bucket_count() → 7  
1 0 h(X)=h(F) → X → F → .reserve(min_capacity) → (make table large enough to handle min_capacity elements)  
hash bucket .rehash(#hash_buckets) → (can allocate even more buckets if max. load factor demands it)
```

Extract Nodes

$O(1)$ avg., $O(n)$ worst

```
1 5 7 .extract(5) → 5  
1 5 7 .extract(@pos) → 5
```

Allows efficient key modification and transfer of keys between different set objects.

Merge Two Sets

$O(n_2)$ average, $O(n_1 \cdot n_2 + n_2)$ worst case

```
unordered_set<int> s1 {1,5,3};  
unordered_set<int> s2 {8,1,2};  
s1.merge(s2);
```

Insert & Construct A Key in Place

$O(1)$ avg., $O(n)$ worst

```
1,3 5,6 .emplace(8,7) → {@inserted, true} (if inserted)  
pair constructor arguments
```

potential performance benefit by hinting at probable insert position

Insert Nodes

$O(1)$ avg., $O(n)$ worst

```
1,3 5,6 .emplace_hint(@hint, 8,7) → @inserted
```

members of the return type

Erase One Key or A Range of Keys

$O(\#erased)$ avg., $O(n)$ worst case

```
1 5 3 7 .erase(2) → 0 #erased  
1 5 3 7 .erase(3) → 1  
1 5 3 7 .erase(@pos) → @after_erased  
1 5 3 7 .erase(@beg, @end) → @after_erased
```

Modify Key

Direct modification not allowed!
Instead: extract key, modify its value and re-insert.

```
unordered_set s {1,7,5};  
auto node = s.extract(5);  
if (node) { // if key existed  
    node.value() = 8;  
    s.insert(move(node));  
}
```

unordered_map

Construct A New Map Object

```
unordered_map<string,int> m0 {} → [ ]
```

```
unordered_map<string,int> m1 {{"A",2}, {"Z",4}, {"Y",7}} → [Z 4 A 2 Y 7]
```

```
unordered_map<string,int> m2 (begin(m1)+1, end(m1)) → [A 2 Y 7]
```

```
unordered_map<string,int> deep_copy_of_m1(m1) → [Z 4 A 2 Y 7]
```

C++17 key and mapped types deducible from arguments

```
unordered_map<double,double> m3 {{2, 3.14}, {5, 6.0}}; // unordered_map<int,double>
```

Assign New Content To An Existing Map

```
unordered_map<string,int> m1 {"X",2}; (deep copy from source)
```

```
unordered_map<string,int> m2 {"A",1}, {"G",4}; new state of m1
```

```
X 2 = A 1 G 4 → [A 1 G 4]
```

Lookup Using Keys as Input $O(1)$ average, $O(n)$ worst case

```
unordered_map<string,int> m {"S",3}, {"X",5}, {"F",1};
```

```
S 3 F 1 X 5 .contains("W") → false C++20
```

```
S 3 F 1 X 5 .contains("X") → true
```

```
S 3 F 1 X 5 .count("W") → 0 can be > 1 only for unordered_multimap
```

```
S 3 F 1 X 5 .count("X") → 1
```

```
S 3 F 1 X 5 .find("W") → @end (=no match)
```

```
S 3 F 1 X 5 .find("X") → @match
```

```
S 3 F 1 X 5 .equal_range("F") → {@1st_equal, @after}
```

```
S 3 F 1 X 5 .at("F") → 1
```

```
S 3 F 1 X 5 .at("B") → Throws std::out_of_range
```

Insert A Single Key-Value Pair $O(1)$ avg., $O(n)$ worst (inserted yes/no)

```
F 1 A 3 .insert("W",2) → {@inserted, true}
```

```
F 1 A 3 .insert("A",9) → {@blocking, false}
```

potential performance benefit by hinting at probable insert position

```
F 1 A 3 .insert(@hint, "W",2) → @instd/blockng
```

Insert Multiple Key-Value Pairs $O(\#ins)$ avg., $O(n \cdot \#ins + \#ins)$ worst

```
A 3 .insert({{"G",4}, {"K",9}, {"A",7}}) → [K 9 A 3 G 4]
```

```
A 3 .insert(@inBegin, @inEnd) → [K 9 A 3 G 4]
```

source container G 4 K 9 A 7 X 2

Construct Key-Value Pair $O(1)$ avg., $O(n)$ worst (inserted yes/no)

```
F 1 A 3 .emplace("W",2) → {@inserted, true}
```

```
F 1 A 3 ["W"] = 2 → [F 1 W 2 A 3]
```

potential performance benefit by hinting at probable insert position

```
F 1 A 3 .emplace_hint(@hint, "W",2) → @inserted
```

```
F 1 A 3 ["W"] = 0 newly created mapped values are value-initialized (e.g. 0 for int)
```

advantage: does not move from rvalue input parameters if not inserted

```
F 1 A 3 .try_emplace("W",2) → {@inserted, true}
```

Insert or Assign Value $O(1)$ avg., $O(n)$ worst (C++17 inserted yes/no)

```
F 1 B 3 .insert_or_assign("B",5) → {@as, false}
```

```
F 1 B 3 .insert_or_assign("R",6) → {@ins, true}
```

potential performance benefit by hinting at probable insert position

```
F 1 B 3 .insert_or_assign(@hint, "B",5) → @as
```

```
F 1 B 3 .insert_or_assign(@hint, "G",2) → @ins
```

Query & Control Hash Table Properties

hash function: $h(key) \mapsto \text{bucket index}$

```
unordered_map<string,int> um {{"E",6}, {"X",4}, {"Z",1}, {"A",3}, {"F",2}, {"B",2}}
```

```
um .bucket("E") → 4 (key → hash bucket index)
```

```
um .begin(4) → @first_in_bucket
```

```
um .end(4) → @one_behind_last_in_bucket
```

```
um .bucket_size(4) → 2
```

```
um .bucket_count() → 7
```

hash bucket

um .load_factor() → 4/7 = 0.57

um .max_load_factor(0.8) (set)

um .max_load_factor() → 0.8 (get)

(make table large enough to handle min_capacity elements)

```
um .reserve(min_capacity) → [ ]
```

```
um .rehash(#hash_buckets) → [ ]
```

(can allocate even more buckets if max. load factor demands it)

Access / Modify Value $O(1)$ avg., $O(n)$ worst

```
unordered_map<string,int> m {"F",1}, {"A",3};
```

```
F 1 A 3 ["A"] = 3
```

```
F 1 A 3 ["A"] = 4 → [F 1 A 4]
```

Attention: [k] inserts new pair if key k is not present!

```
F 1 A 3 ["W"] = 2 → [F 1 W 2 A 3]
```

```
F 1 A 3 ["W"] = 0 newly created mapped values are value-initialized (e.g. 0 for int)
```

Erase Key-Value-Pair(s) $O(\#\text{erased})$ avg., $O(n)$ worst case

```
F 1 A 3 X 5 .erase("W") → 0
```

```
F 1 A 3 X 5 .erase("A") → 1
```

```
F 1 A 3 X 5 .erase(@pos) → @after
```

```
F 1 A 3 X 5 .erase(@beg, @end) → @after
```

Modify Key

```
unordered_map<string,int> m {"F",1}, {"A",3}; F 1 A 3
```

```
auto node = m.extract("A"); A 3
```

```
if (node) { node.key() = "X"; X 3 }
```

```
m.insert(move(node));
```

Extract Nodes to efficiently transfer key-value pairs C++17

```
F 1 R 2 A 3 .extract("R") → R 2
```

```
F 1 R 2 A 3 .extract(@pos) → R 2
```

(C++17 avg., O(n) worst)

(Re-)Insert Nodes

```
F 1 A 3 .insert(N 5) → {@position.inserted.node} members of the return type
```

```
F 1 A 3 N 5 true (empty)
```

```
F 1 A 3 .insert(A 6) → {@position.inserted.node} false Z 4
```

```
F 1 A 3 .insert(@) → {@position.inserted.node} (empty node) false
```

```
F 1 A 3 .insert(@hint, X 5) → @inserted
```

```
F 1 A 3 .insert(@hint, F 6) → @blocking
```

Merge Two Maps C++17

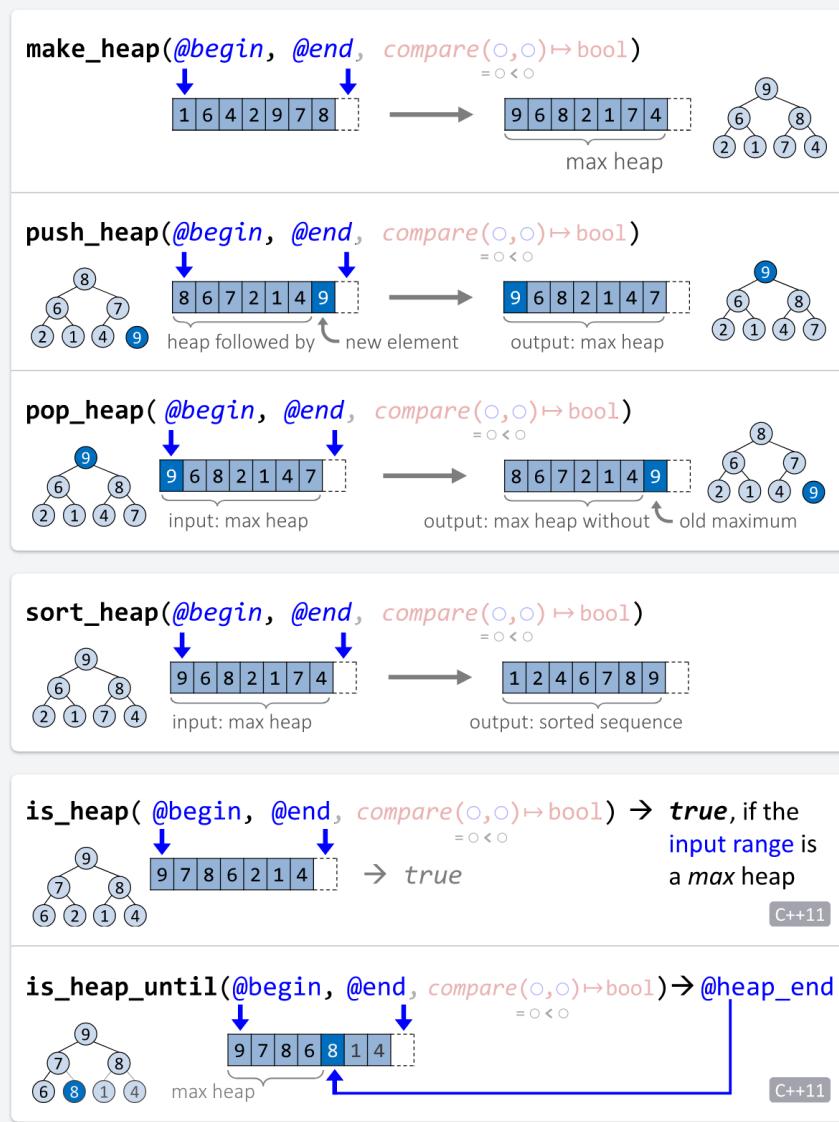
```
unordered_map<string,int> m1 {"F",1}, {"S",3}, {"X",5};
```

```
unordered_map<string,int> m2 {"A",2}, {"X",7};
```

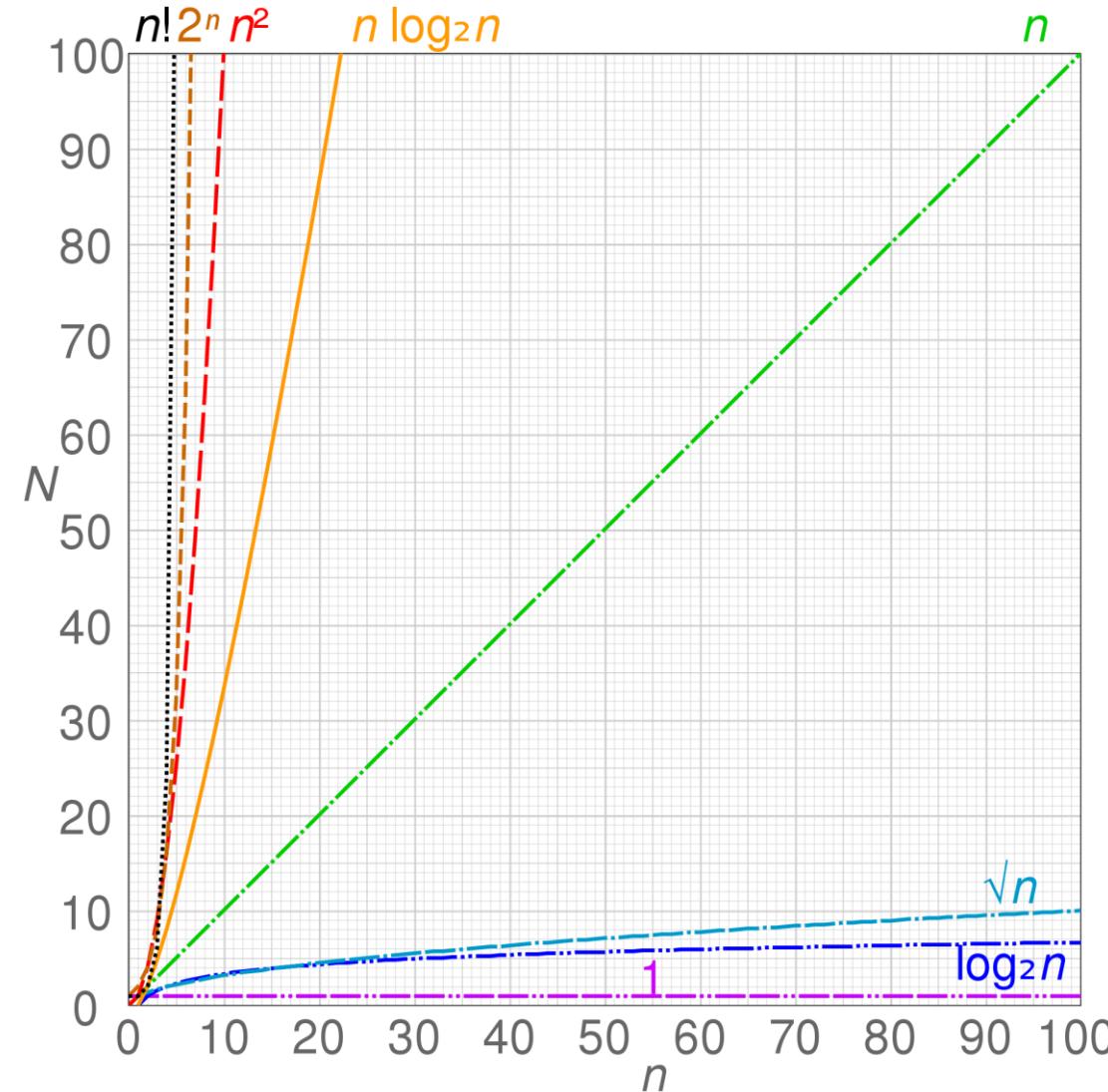
```
F 1 S 3 X 5 .merge(A 2 X 7) O(n2) average, O(n1 · n2 + n2) worst case
```

```
F 1 A 2 S 3 X 5 X 7
```

Heap operations



Оценка сложности Big-O



Оценка сложности для контейнеров STL

Container	Insert Head	Insert Tail	Insert	Remove Head	Remove Tail	Remove	Index Search	Find
vector	n/a	O(1)	O(n)	O(1)	O(1)	O(n)	O(1)	O(log n)
list	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	n/a	O(n)
deque	O(1)	O(1)	n/a	O(1)	O(1)	O(n)	n/a	n/a
queue	n/a	O(1)	n/a	O(1)	n/a	n/a	O(1)	O(log n)
stack	O(1)	n/a	n/a	O(1)	n/a	n/a	n/a	n/a
map	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(1)	O(log n)
multimap	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(1)*	O(log n)
set	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(1)	O(log n)
multiset	n/a	n/a	O(log n)	n/a	n/a	O(log n)	O(1)*	O(log n)

<https://users.cs.northwestern.edu/~riesbeck/programming/c++/stl-summary.html>