Inside-Out: STL

How to use it wisely?

Jadwiga Pokorska

TietoEvry

26.02.2020

Presentation plan

1. Introduction
   - Time complexity
   - Vector
2. Balanced trees
   - BST
   - AVL
3. Hash tables
   - Idea
   - Properties and limitations
   - Workarounds

Time complexity

Purpose

Time complexity is a tool to measure the efficiency of our algorithm.

Time complexity

Purpose

Time complexity is a tool to measure the efficiency of our algorithm.

Usually defined with *big-O* notation:

- $O(N)$,
- $O(N^2)$,
- $O(\log N)$,
- $O(N \cdot \log N)$,
- $O(\sqrt{N})$,

Task: find not paired item

Task

Given an array of integers, find the only number that does **not** have a pair.

Task: find not paired item

Task

Given an array of integers, find the only number that does **not** have a pair.

Example

For array $[2, 3, 7, 7, 2, 3, 2]$ the answer is 2.

Task: find not paired item

```
int find_not_paired(const vector<int>& t) {
    for (int selected_item : t) {
        int cnt = 0;
        for (int item : t)
            if (item == selected_item)
                cnt++;
        if (cnt % 2 == 1)
            return selected_item;
    }
    return 0;
}
```

What is the time complexity?

Task: find not paired item

```
int find_not_paired(const vector<int>& t) {
    for (int selected_item : t) {
        int cnt = 0;
        for (int item : t)
            if (item == selected_item)
                cnt++;
        if (cnt % 2 == 1)
            return selected_item;
    }
    return 0;
}
```

What is the time complexity? $O(N^2)$

Task: find not paired item

```
int find_not_paired(const vector<int>& t) {
    sort(t.begin(), t.end());
    int cnt = 0, prev = -1;
    for (int i = 0; i < t.size(); ++i) {
        if (prev == t[i]) cnt++;
        else {
            if (cnt % 2 == 1) return prev;
            prev = t[i];
            cnt = 0;
        }
    }
    if (cnt % 2 == 1) return prev;
    return 0;
}
```

What is the time complexity?

Task: find not paired item

```
int find_not_paired(const vector<int>& t) {
    sort(t.begin(), t.end());
    int cnt = 0, prev = -1;
    for (int i = 0; i < t.size(); ++i) {
        if (prev == t[i]) cnt++;
        else {
            if (cnt % 2 == 1) return prev;
            prev = t[i];
            cnt = 0;
        }
    }
    if (cnt % 2 == 1) return prev;
    return 0;
}
```

What is the time complexity? $O(N \cdot \log N)$

Task: find not paired item

```
int find_not_paired(const vector<int>& t) {
    int result = 0;
    for (int item : t)
        result ^= item;
    if (result > 0)
        return result;
    return 0;
}
```

What is the time complexity?

Task: find not paired item

```
int find_not_paired(const vector<int>& t) {
    int result = 0;
    for (int item : t)
        result ^= item;
    if (result > 0)
        return result;
    return 0;
}
```

What is the time complexity? $O(N)$

Task: find not paired item

```
int find_not_paired(const vector<int>& t) {
    map<int,int> m;
    for (int item : t)
        m[item]++;
    for (auto it : m) {
        if (it.second % 2 == 1)
            return it.first;
    }
    return 0;
}
```

What is the time complexity?

Task: find not paired item

```
int find_not_paired(const vector<int>& t) {
    map<int,int> m;
    for (int item : t)
        m[item]++;
    for (auto it : m) {
        if (it.second % 2 == 1)
            return it.first;
    }
    return 0;
}
```

What is the time complexity? $O(N \cdot \log N)$

Task: find not paired item

```
int find_not_paired(const vector<int>& t) {
    unordered_map<int,int> m;
    for (int item : t)
        m[item]++;
    for (auto it : m) {
        if (it.second % 2 == 1)
            return it.first;
    }
    return 0;
}
```

What is the time complexity?

Task: find not paired item

```
int find_not_paired(const vector<int>& t) {
    unordered_map<int,int> m;
    for (int item : t)
        m[item]++;
    for (auto it : m) {
        if (it.second % 2 == 1)
            return it.first;
    }
    return 0;
}
```

What is the time complexity? $O(N)$

Vector – time complexity

Where do I find the information about the time complexity?

Vector – time complexity

Where do I find the information about the time complexity?

Documentation!

cppreference.com Create account Search

Page Discussion View Edit History

C++ / Containers library / std::vector

std::vector

Defined in header <vector>

```
template<
    class T,
    class Allocator = std::allocator<T>
> class vector;
```
(1)

```
namespace pmr {
    template <class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```
(2) (since C++17)

1) `std::vector` is a sequence container that encapsulates dynamic size arrays.

2) `std::pmr::vector` is an alias template that uses a polymorphic allocator

The elements are stored contiguously, which means that elements can be accessed not only through iterators, but also using offsets to regular pointers to elements. This means that a pointer to an element of a vector may be passed to any function that expects a pointer to an element of an array. (since C++03)

The storage of the vector is handled automatically, being expanded and contracted as needed. Vectors usually occupy more space than static arrays, because more memory is allocated to handle future growth. This way a vector does not need to reallocate each time an element is inserted, but only when the additional memory is exhausted. The total amount of allocated memory can be queried using `capacity()` function. Extra memory can be returned to the system via a call to `shrink_to_fit()`. (since C++11)

Reallocations are usually costly operations in terms of performance. The `reserve()` function can be used to eliminate reallocations if the number of elements is known beforehand.

The complexity (efficiency) of common operations on vectors is as follows:

- Random access - constant $O(1)$
- Insertion or removal of elements at the end - amortized constant $O(1)$
- Insertion or removal of elements - linear in the distance to the end of the vector $O(n)$

`std::vector` (for T other than `bool`) meets the requirements of *Container*, *AllocatorAwareContainer*, *SequenceContainer*, *ContiguousContainer* (since C++17) and *ReversibleContainer*.

Vector – time complexity

cppreference.com — Create account — Search

Page | Discussion — View | Edit | History

C++ / Containers library / std::vector

std::vector<T,Allocator>::erase

`iterator erase( iterator pos );`	(1)	(until C++11)
`iterator erase( const_iterator pos );`		(since C++11)
`iterator erase( iterator first, iterator last );`	(2)	(until C++11)
`iterator erase( const_iterator first, const_iterator last );`		(since C++11)

Erases the specified elements from the container.

1) Removes the element at pos.

2) Removes the elements in the range `[first, last)`.

Invalidates iterators and references at or after the point of the erase, including the `end()` iterator.

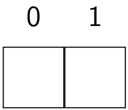
The iterator pos must be valid and dereferenceable. Thus the `end()` iterator (which is valid, but is not dereferencable) cannot be used as a value for pos.

The iterator `first` does not need to be dereferenceable if `first==last`: erasing an empty range is a no-op.

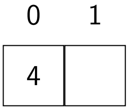
Complexity

Linear: the number of calls to the destructor of T is the same as the number of elements erased, the assignment operator of T is called the number of times equal to the number of elements in the vector after the erased elements

Vector – internal implementation



Vector – internal implementation



Vector – internal implementation

0	1
4	12

Vector – internal implementation

0	1
4	12

0	1	2	3

Vector – internal implementation

0	1
4	12

0	1	2	3
4			

Vector – internal implementation

0	1
4	12

0	1	2	3
4	12		

Vector – internal implementation

0	1	2	3
4	12		

Vector – internal implementation

0	1	2	3
4	12	1	

Vector – internal implementation

0	1	2	3
4	12	1	5

Vector – internal implementation

0	1	2	3	4	5	6	7

0	1	2	3
4	12	1	5

Vector – internal implementation

0	1	2	3	4	5	6	7
4							

0	1	2	3
4	12	1	5

Vector – internal implementation

0	1	2	3	4	5	6	7
4	12						

0	1	2	3
4	12	1	5

Vector – internal implementation

0	1	2	3	4	5	6	7
4	12	1					

0	1	2	3
4	12	1	5

Vector – internal implementation

0	1	2	3	4	5	6	7
4	12	1	5				

0	1	2	3
4	12	1	5

Vector – internal implementation

0	1	2	3	4	5	6	7
4	12	1	5				

Vector – internal implementation

0	1	2	3	4	5	6	7
4	12	1	5	87			

Vector – time complexity

- *insert (back)*

Vector – time complexity

- *insert (back)* – $O(1)$ (expected),

Vector – time complexity

- *insert (back)* – $O(1)$ (expected),
- *delete (back)*

Vector – time complexity

- *insert (back)* – $O(1)$ (expected),
- *delete (back)* – $O(1)$,

Vector – time complexity

- *insert (back)* – $O(1)$ (expected),
- *delete (back)* – $O(1)$,
- *lookup (index)*

Vector – time complexity

- *insert (back)* – $O(1)$ (expected),
- *delete (back)* – $O(1)$,
- *lookup (index)* – $O(1)$,

Vector – time complexity

- *insert (back)* – $O(1)$ (expected),
- *delete (back)* – $O(1)$,
- *lookup (index)* – $O(1)$,
- *insert (middle)* – $O(N)$,
- *delete (middle)* – $O(N)$,
- *find (value)* – $O(N)$.

Vector – time complexity

- *insert (back)* – $O(1)$ (expected),
- *delete (back)* – $O(1)$,
- *lookup (index)* – $O(1)$,
- *insert (middle)* – $O(N)$,
- *delete (middle)* – $O(N)$,
- *find (value)* – $O(N)$.

Note: vector does not shrink by itself.

Insert complexity proof.

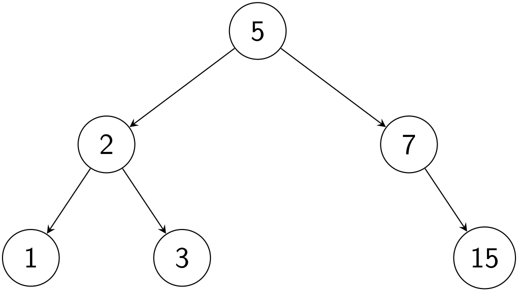
Proof!

Binary search tree

Long long ago, before c++11 . . .

Binary search tree

Long long ago, before c++11 ... all sets were based on the binary search trees.



AVL vs. BST

AVL is just a regular BST with rotations that guarantee reasonable time complexities.

AVL vs. BST

AVL is just a regular BST with rotations that guarantee reasonable time complexities.

AVL vs. BST – worst case time complexity

BST

- *insert* – $O(N)$,
- *delete* – $O(N)$,
- *lookup* – $O(N)$.

AVL vs. BST – worst case time complexity

BST

- *insert* – $O(N)$,
- *delete* – $O(N)$,
- *lookup* – $O(N)$.

AVL

- *insert* – $O(\log N)$,
- *delete* – $O(\log N)$,
- *lookup* – $O(\log N)$.

AVL vs. BST – worst case time complexity

BST

- *insert* – $O(N)$,
- *delete* – $O(N)$,
- *lookup* – $O(N)$.

AVL

- *insert* – $O(\log N)$,
- *delete* – $O(\log N)$,
- *lookup* – $O(\log N)$.

Note

`std::set` and `std::map` are internally using Red-Black Trees which have the same time complexity as AVL, but different internal constraints to ensure balancing.

Hashing – why do we need it?

Hashing – why do we need it?

U – universe of numbers that may appear in the data.

What if $|U|$ is small (i.e. $1\,000\,000$)?

Hashing – why do we need it?

U – universe of numbers that may appear in the data.

What if $|U|$ is small (i.e. 1 000 000)?

What if $|U|$ is large (i.e. 10^{18})?

Hashing – why do we need it?

U – universe of numbers that may appear in the data.

What if $|U|$ is small (i.e. $1\,000\,000$)?

What if $|U|$ is large (i.e. 10^{18})?

...then we need *hashing*!

Hashing – why do we need it?

U – universe of numbers that may appear in the data.
A – array that we actually have.
M – size of the array A.

Hashing – why do we need it?

U – universe of numbers that may appear in the data.
A – array that we actually have.
M – size of the array A.

If we had a (fast) function $h : U \to \{0, \ldots, M-1\}$, then we could store each element x within $A[f(x)]$.

Hashing – why do we need it?

U – universe of numbers that may appear in the data.
A – array that we actually have.
M – size of the array A.

If we had a (fast) function $h : U \to \{0, \ldots, M - 1\}$, then we could store each element x within $A[f(x)]$.

Note: If $|U|$ is small, then the identity function $f(x) = x$ would do.

Hashing – why do we need it?

U – universe of numbers that may appear in the data.
A – array that we actually have.
M – size of the array A.

If we had a (fast) function $h : U \to \{0, \ldots, M-1\}$, then we could store each element x within $A[f(x)]$.

Note: If $|U|$ is small, then the identity function $f(x) = x$ would do.

What could possibly go wrong?

Hashing – why do we need it?

U – universe of numbers that may appear in the data.
A – array that we actually have.
M – size of the array A.

If we had a (fast) function $h : U \to \{0, \dots, M-1\}$, then we could store each element x within $A[f(x)]$.
Note: If $|U|$ is small, then the identity function $f(x) = x$ would do.
What could possibly go wrong?

- what if we have a *collision* $(f(x) = f(y))$?

Hashing – why do we need it?

U – universe of numbers that may appear in the data.
A – array that we actually have.
M – size of the array A.

If we had a (fast) function $h : U \rightarrow \{0, \ldots, M-1\}$, then we could store each element x within $A[f(x)]$.
Note: If $|U|$ is small, then the identity function $f(x) = x$ would do.
What could possibly go wrong?

- what if we have a *collision* $(f(x) = f(y))$?
- what if f does **not** distribute elements uniformly over the available cells?

Hashing – properties

- f is an arithmetic expression, i.e. $f(x) = (5 \cdot x) \bmod M$,

Hashing – properties

- f is an arithmetic expression, i.e. $f(x) = (5 \cdot x) \bmod M$,
- f is randomly chosen from some set of functions (family),

Hashing – properties

- f is an arithmetic expression, i.e. $f(x) = (5 \cdot x) \bmod M$,
- f is randomly chosen from some set of functions (family),
- *collision* – we store all the key-value pairs within a linked list,

Hashing – properties

- f is an arithmetic expression, i.e. $f(x) = (5 \cdot x) \mod M$,
- f is randomly chosen from some set of functions (family),
- *collision* – we store all the key-value pairs within a linked list,
- a good family of functions ensures the correct distrubution (in expectation),

Hashing – properties

- f is an arithmetic expression, i.e. $f(x) = (5 \cdot x) \bmod M$,
- f is randomly chosen from some set of functions (family),
- *collision* – we store all the key-value pairs within a linked list,
- a good family of functions ensures the correct distrubution (in expectation),
- if the array gets full then all elements are rehashed into a bigger one.

Hashing – properties

- f is an arithmetic expression, i.e. $f(x) = (5 \cdot x) \bmod M$,
- f is randomly chosen from some set of functions (family),
- *collision* – we store all the key-value pairs within a linked list,
- a good family of functions ensures the correct distrubution (in expectation),
- if the array gets full then all elements are rehashed into a bigger one.

What are the time complexities of the operations?

- *insert (no lookup)* –

Hashing – properties

- f is an arithmetic expression, i.e. $f(x) = (5 \cdot x) \mod M$,
- f is randomly chosen from some set of functions (family),
- *collision* – we store all the key-value pairs within a linked list,
- a good family of functions ensures the correct distrubution (in expectation),
- if the array gets full then all elements are rehashed into a bigger one.

What are the time complexities of the operations?

- *insert (no lookup)* – $O(1)$,

Hashing – properties

- f is an arithmetic expression, i.e. $f(x) = (5 \cdot x) \bmod M$,
- f is randomly chosen from some set of functions (family),
- *collision* – we store all the key-value pairs within a linked list,
- a good family of functions ensures the correct distrubution (in expectation),
- if the array gets full then all elements are rehashed into a bigger one.

What are the time complexities of the operations?

- *insert (no lookup)* – $O(1)$,
- *delete* –

Hashing – properties

- f is an arithmetic expression, i.e. $f(x) = (5 \cdot x) \bmod M$,
- f is randomly chosen from some set of functions (family),
- *collision* – we store all the key-value pairs within a linked list,
- a good family of functions ensures the correct distrubution (in expectation),
- if the array gets full then all elements are rehashed into a bigger one.

What are the time complexities of the operations?

- *insert (no lookup)* – $O(1)$,
- *delete* – $O(1)$ (expected),

Hashing – properties

- f is an arithmetic expression, i.e. $f(x) = (5 \cdot x) \bmod M$,
- f is randomly chosen from some set of functions (family),
- *collision* – we store all the key-value pairs within a linked list,
- a good family of functions ensures the correct distrubution (in expectation),
- if the array gets full then all elements are rehashed into a bigger one.

What are the time complexities of the operations?

- *insert (no lookup)* – $O(1)$,
- *delete* – $O(1)$ (expected),
- *lookup* –

Hashing – properties

- f is an arithmetic expression, i.e. $f(x) = (5 \cdot x) \bmod M$,
- f is randomly chosen from some set of functions (family),
- *collision* – we store all the key-value pairs within a linked list,
- a good family of functions ensures the correct distrubution (in expectation),
- if the array gets full then all elements are rehashed into a bigger one.

What are the time complexities of the operations?

- *insert (no lookup)* – $O(1)$,
- *delete* – $O(1)$ (expected),
- *lookup* – $O(1)$ (expected).

Other data types

If a more complex data type needs to be stored, then it is necessary to define an own hash function and an equality operator.

Other data types

If a more complex data type needs to be stored, then it is necessary to define an own hash function and an equality operator. What is more, `std::pair` is such a type, so STL does not provide any hash function for us (only the equality operator).

Other data types

If a more complex data type needs to be stored, then it is necessary to define an own hash function and an equality operator. What is more, `std::pair` is such a type, so STL does not provide any hash function for us (only the equality operator).

Example hash function for `std::pair<T1,T2>`:

```
struct pair_hash
{
    template <class T1, class T2>
    std::size_t operator()
        (const std::pair<T1, T2> &pair) const
    {
        return std::hash<T1>()(pair.first)
                ^ std::hash<T2>()(pair.second);
    }
};
```

Other data types

If a more complex data type needs to be stored, then it is necessary to define an own hash function and an equality operator. What is more, `std::pair` is such a type, so STL does not provide any hash function for us (only the equality operator).

Example hash function for `std::pair<T1,T2>`:

```
struct pair_hash
{
    template <class T1, class T2>
    std::size_t operator()
        (const std::pair<T1, T2> &pair) const
    {
        return std::hash<T1>()(pair.first)
               ^ std::hash<T2>()(pair.second);
    }
};
```

Other data types

If a more complex data type needs to be stored, then it is necessary to define an own hash function and an equality operator. What is more, `std::pair` is such a type, so STL does not provide any hash function for us (only the equality operator).

Example hash function for `std::pair<T1,T2>`:

```
struct pair_hash
{
    template <class T1, class T2>
    std::size_t operator()
        (const std::pair<T1, T2> &pair) const
    {
        return std::hash<T1>()(pair.first)
                ^ (std::hash<T2>()(pair.second) << 1);
    }
};
```

Hashing – limitations

Hashing – limitations

- for small sets it's extremely inefficient due to rehashing,

Hashing – limitations

- for small sets it's extremely inefficient due to rehashing,
- for stored N elements likely there will be a list of size $\Omega(\log \log N)$,

Hashing – limitations

- for small sets it's extremely inefficient due to rehashing,
- for stored N elements likely there will be a list of size $\Omega(\log \log N)$,
- hashing function needs to be defined for the stored object (alternatively `operator<` in tree-based map),

Hashing – limitations

- for small sets it's extremely inefficient due to rehashing,
- for stored N elements likely there will be a list of size $\Omega(\log \log N)$,
- hashing function needs to be defined for the stored object (alternatively `operator<` in tree-based map),
- for known hash function it is possible to prepare the data that will all fall into one place (linked list) causing the $\Omega(N)$ blow-up per single operation.

Hashing – limitations

- for small sets it's extremely inefficient due to rehashing,
- for stored N elements likely there will be a list of size $\Omega(\log\log N)$,
- hashing function needs to be defined for the stored object (alternatively `operator<` in tree-based map),
- for known hash function it is possible to prepare the data that will all fall into one place (linked list) causing the $\Omega(N)$ blow-up per single operation.

Code!

Hashing – limitations

- for small sets it's extremely inefficient due to rehashing,
- for stored N elements likely there will be a list of size $\Omega(\log\log N)$,
- hashing function needs to be defined for the stored object (alternatively `operator<` in tree-based map),
- for known hash function it is possible to prepare the data that will all fall into one place (linked list) causing the $\Omega(N)$ blow-up per single operation.

Code!

```
pokorska@thinkpad:~/wrocpp/wrocpp$ make blow
g++     blow.cpp   -o blow
pokorska@thinkpad:~/wrocpp/wrocpp$ ./blow
x = 107897: 361.175 seconds, sum = 2666686666700000
x = 126271: 0.078 seconds, sum = 2666686666700000
pokorska@thinkpad:~/wrocpp/wrocpp$ 
```

Set vs. unordered set in practice

Code!

Set vs. unordered set in practice

Code!

```
pokorjad:~/wrocpp$ ./set_vs_unordered_set
set: 66.966 seconds
set lookup: 49.879 seconds, elems found: 10005884
unordered set: 43.651 seconds
set lookup: 15.091 seconds, elems found: 10005884
pokorjad:~/wrocpp$
```

Open addressing

Hash function: $hash(x) = x \bmod 7$.

0	1	2	3	4	5	6

Open addressing

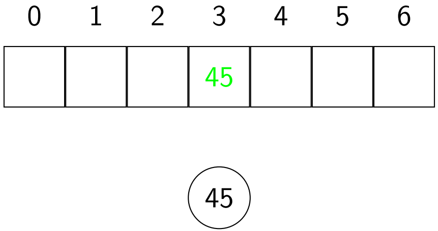
Hash function: $hash(x) = x \bmod 7$.

0	1	2	3	4	5	6

45

Open addressing

Hash function: $hash(x) = x \bmod 7$.



Open addressing

Hash function: $hash(x) = x \bmod 7$.

0	1	2	3	4	5	6
			45			

Open addressing

Hash function: $hash(x) = x \bmod 7$.

0	1	2	3	4	5	6
			45			

84

Open addressing

Hash function: $hash(x) = x \bmod 7$.

0	1	2	3	4	5	6
84			45			

84

Open addressing

Hash function: $hash(x) = x \bmod 7$.

0	1	2	3	4	5	6
84			45			

Open addressing

Hash function: $hash(x) = x \bmod 7$.

0	1	2	3	4	5	6
84			45			

17

Open addressing

Hash function: $hash(x) = x \bmod 7$.

0	1	2	3	4	5	6
84			45			

17

Open addressing

Hash function: $hash(x) = x \bmod 7$.

0	1	2	3	4	5	6
84			45	17		

17

Open addressing

Hash function: $hash(x) = x \bmod 7$.

0	1	2	3	4	5	6
84			45	17		

Open addressing

Hash function: $hash(x) = x \bmod 7$.

0	1	2	3	4	5	6
84			45	17		

3

Open addressing

Hash function: $hash(x) = x \bmod 7$.

0	1	2	3	4	5	6
84			45	17		

3

Open addressing

Hash function: $hash(x) = x \bmod 7$.

0	1	2	3	4	5	6
84			45	17		

3

Open addressing

Hash function: $hash(x) = x \bmod 7$.

0	1	2	3	4	5	6
84			45	17	3	

3

Open addressing

Hash function: $hash(x) = x \bmod 7$.

0	1	2	3	4	5	6
84			45	17	3	

Cuckoo-hashing (optional)

Idea of how to make the lookup in $O(1)$ in worst case (not expected).