

Содержание

Data structures and algorithms	2
Data structures	2
Dynamic arrays	2
Stack	3
Queue	4
Deque	5
Sets	6
Упорядоченные и неупорядоченные множества	7
Multisets	7
Maps	9
Priority Queue	10
Algorithms	11

Data structures and algorithms

Data structures

Dynamic arrays

Их идея очень проста, вместо статического массива с фиксированной длиной мы используем массив переменной длины. Это удобно тк не всегда удобно работать со статическим массивом, тем более когда его надо обработать. Наиболее популярный динамический массив в C++ это vector:

```
vector<int> v;  
v.push_back(3); // [3]  
v.push_back(2); // [3, 2]  
v.push_back(5); // [3, 2, 4]
```

Вывод чисел аналогичен привычному:

```
cout << v[0] << "\n"; // 3  
cout << v[1] << "\n"; // 2  
cout << v[2] << "\n"; // 5
```

Также у вектора есть удобный метод size():

```
for (int i=0; i<v.size(); ++i) {  
    cout << v[i] << "\n";  
}
```

Но не забываем про v.at()

Но последние стандарты C++ позволяют упростить вывод чисел:

```
for (auto x : v) {  
    cout << x << "\n";  
}
```

Есть и другие интересные методы у vector, быть может потом напишу про них.

Stack

Он предоставляет нам две $O(1)$ операции:

- `push()`: добавление элемента “наверх”
- `pop()`: удаление элемента “сверху”

А также одну $O(1)$ функцию доступа:

- `top()`: возвращение “верхнего” элемента

Таким образом в стеке у нас есть доступ только к элементу сверху(`top`).

Stack = FILO (first in last out)

```
stack<int> s;
s.push(3); // {3}
s.push(2); // {2, 3}
s.push(5); // {5, 2, 3}
cout << s.top(); // 5
s.pop();
cout << s.top(); // 2
```

Подробнее о стеке можно почитать [здесь](#).

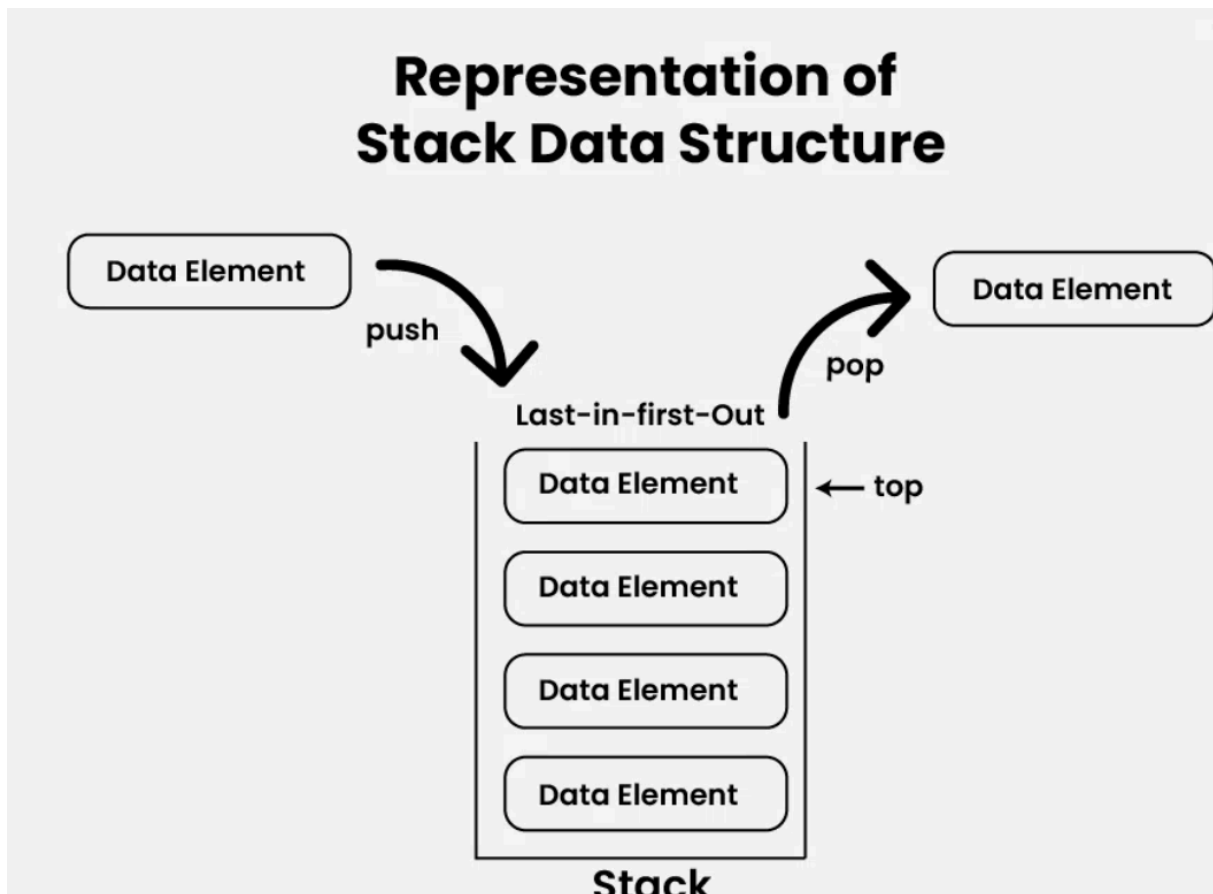


Figure 1: Схема работы стека

Queue

Очередь тоже предоставляет две $O(1)$ операции:

- `push()`: добавление элемента в “конец” очереди
- `pop()`: удаление первого элемента из очереди

А также одну $O(1)$ операцию доступа:

- `front()`: возвращает первый элемент очереди

Таким образом в очереди у нас есть доступ к первому и последнему элементам, а сама queue это FIFO(first in first out).

```
queue<int> q;
q.push(3); // {3}
q.push(2); // {3, 2}
q.push(5); // {3, 2, 5}
cout << q.front(); // 3
q.pop();
cout << q.front() // 2
```

Подробнее про очередь можно почитать [здесь](#).

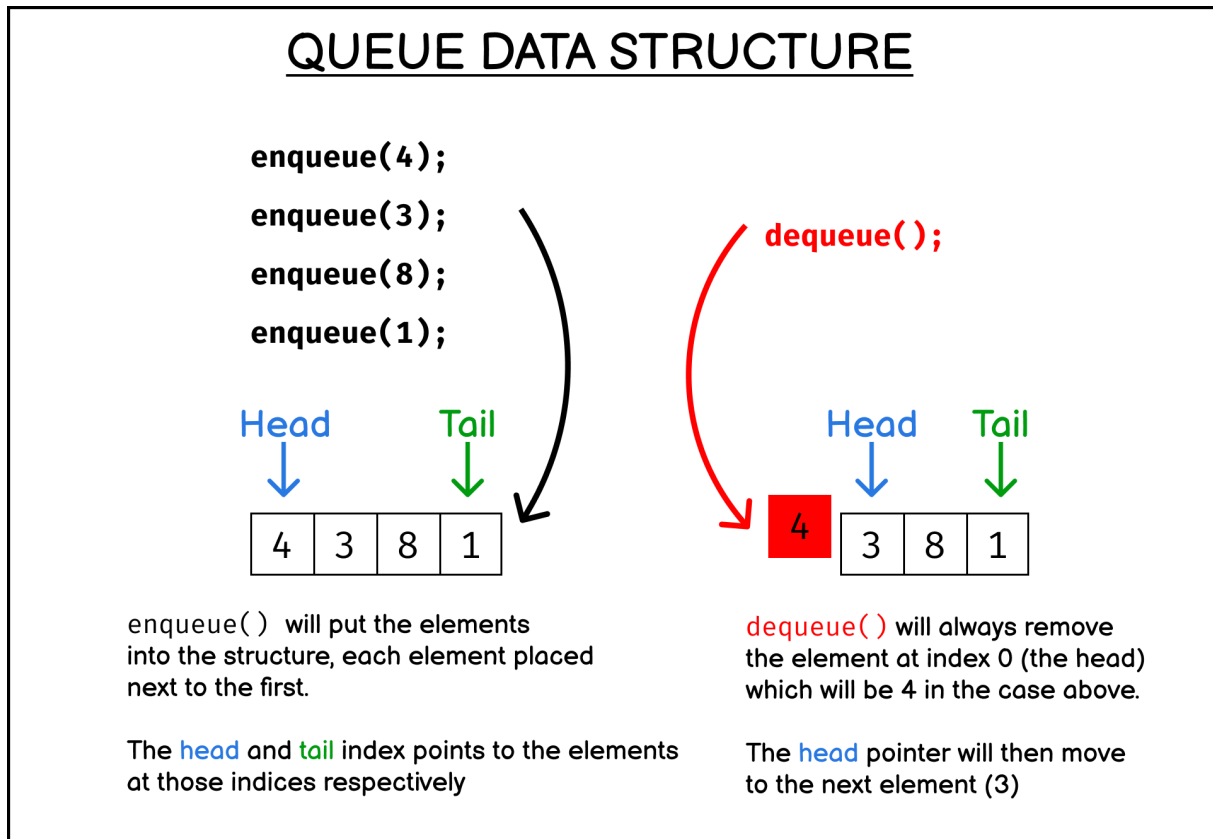


Figure 2: Схема работы очереди (да, тут другие названия, но суть та же)

Deque

это двусторонняя очередь. Она предоставляет нам следующие $O(1)$ методы:

- `push_back()`
- `pop_back()`
- `push_front()`
- `pop_front()`

А также $O(1)$ операции доступа:

- `back()`
- `front()`

```
deque<int> d;
d.push_back(5); // {5}
d.push_back(2); // {5, 2}
d.push_front(3); // {3, 5, 2}
d.pop_back(); // {3, 5}
d.pop_front(); // {5}
```

Внутреннее устройство сложнее чем у вектора, поэтому deque медленнее, но тем не менее $O(1)$ есть. Подробнее о deque можно почитать [здесь](#).

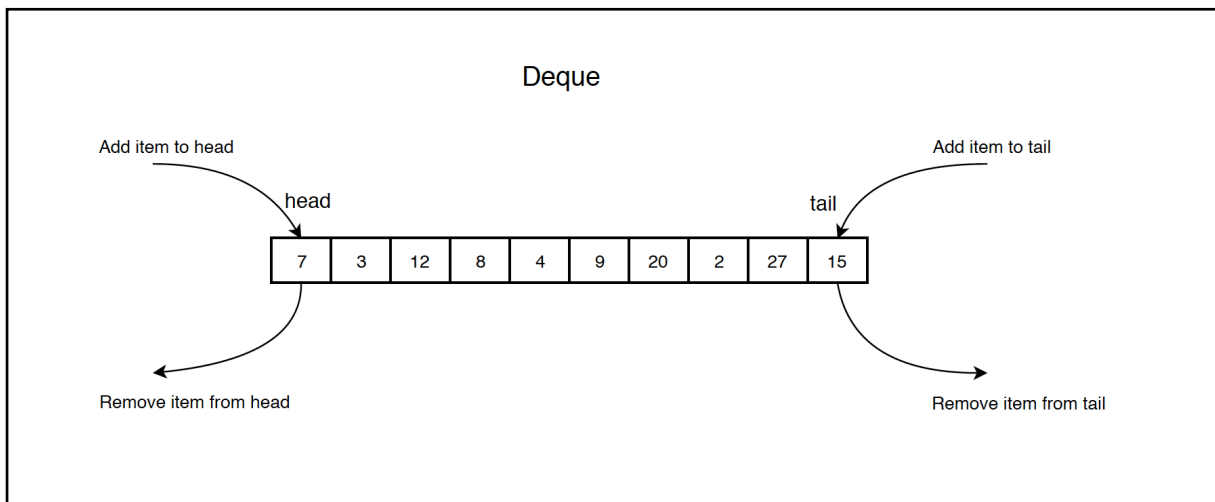


Figure 3: Схема работы дека

Sets

Множеством называется структура данных, в которой хранится набор элементов. Основные операции над множествами - вставка, поиск и удаление. Множества реализованы так, что все эти операции эффективны, что часто позволяет улучшить время работы алгоритмов.

Множества и мультимножества

В стандартной библиотеке C++ имеются две структуры, относящиеся к множествам:

- `set` основана на сбалансированном двоичном дереве поиска, его операции работают за время $O(\log n)$
- `unordered_set` основана на хэш-таблице и работает в среднем за $O(1)$. (да, есть крайне малая вероятность выполнения работы алгоритма за $O(n)$)

(Позже мы обсудим, что за деревья и хэш-таблицы такие).

Пример работы из коробки:

```
set<int> s;
s.insert(3);
s.insert(2);
s.insert(5);
cout << s.count(3) << '\n'; // 1
cout << s.count(4) << '\n'; // 0
s.erase(3);
s.insert(4);
cout << s.count(3) << "\n"; // 0
cout << s.count(4) << "\n"; // 1
```

)

Идея множеств в том, что в ней нет дубликатов, только уникальные элементы. Если элемент уже есть во множестве, его невозможно добавить дважды.

```
set<int> s;
s.insert(3);
s.insert(3);
cout << s.count(3) << "\n"; // 1
```

Множества можно использовать как вектор, однако доступ к элементам с помощью оператора `[]` невозможен. В коде ниже выводится количество элементов, а затем эти элементы перебираются:

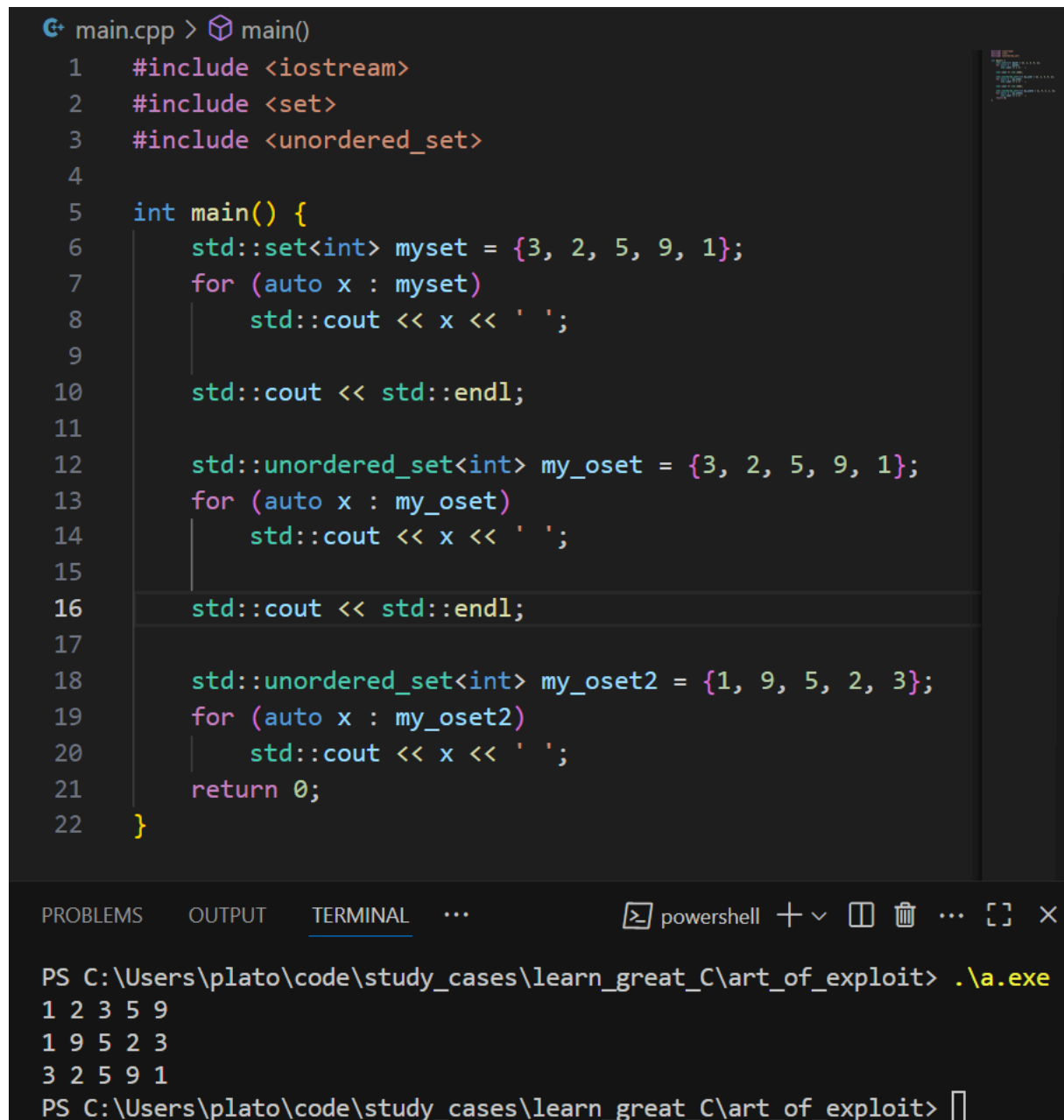
```
cout << s.size() << "\n";
for (auto x : s) {
    cout << x << "\n";
}
```

Функция `find(x)` возвращает итератор, указывающий на элемент со значением `x`. Если же множество не содержит `x`, то возвращается итератор `end()`.

```
auto it = s.find(x);
if (it == s.end()) {
    // x не найден
}
```

Упорядоченные и неупорядоченные множества

Как можно догадаться из названия либо множество упорядочено, либо нет, но что же это значит? Если порядок неважен, берем `set`, иначе `unordered_set`:



```

main.cpp > main()
1  #include <iostream>
2  #include <set>
3  #include <unordered_set>
4
5  int main() {
6      std::set<int> myset = {3, 2, 5, 9, 1};
7      for (auto x : myset)
8          std::cout << x << ' ';
9
10     std::cout << std::endl;
11
12     std::unordered_set<int> my_aset = {3, 2, 5, 9, 1};
13     for (auto x : my_aset)
14         std::cout << x << ' ';
15
16     std::cout << std::endl;
17
18     std::unordered_set<int> my_aset2 = {1, 9, 5, 2, 3};
19     for (auto x : my_aset2)
20         std::cout << x << ' ';
21     return 0;
22 }

```

PROBLEMS OUTPUT TERMINAL ... powershell + - [] [x]

```

PS C:\Users\plato\code\study_cases\learn_great_C\art_of_exploit> .\a.exe
1 2 3 5 9
1 9 5 2 3
3 2 5 9 1
PS C:\Users\plato\code\study_cases\learn_great_C\art_of_exploit>

```

Figure 4: Пример работы `set` и `unordered set`

Multisets

Да, есть еще и мультимножества и они тоже либо упорядочены либо нет, отличие от множеств в том, что они допускают повторения.

```

multiset<int> s;
s.insert(5);
s.insert(5);
s.insert(5);
cout << s.count(5) << "\n"; // 3

```

Функция `erase` удаляет все копии значения из мультимножества.

```
s.erase(5);  
cout << s.count(5) << "\n"; // 0
```

Если требуется удалить только одно значение, то можно поступить так:

```
s.erase(s.find(5)); cout << s.count(5) << "n";
```

Но count и erase работают за $O(k)$, где k количество подсчитываемых или удаляемых элементов

Maps

А вот и наш любимчик `map`(отображения), идея проста, есть ключ, тогда есть и значение(как логин и пароль, или как индекс и значение по данному индексу).

Но в `map` ключом может быть и строка, и число, и любой другой тип, при этом необязательно последовательный.

Устройство Отображений таково:

1. У нас есть Хэш-функция
2. Передав в Хэш-функцию ключ, мы получаем число, взяв остаток от числа(некоторого n , это либо `saracity`, либо `size`) получим номер бакета (ячейки памяти, куда в дальнейшем и будем складывать наши ключи со значениями, если бакет переполнится, то произойдет рехэширование)

В STL имеется две структуры как и в случае со множествами:

- `map`, основан на сбалансированном двоичном дереве со временем доступа $O(\log n)$.
- `unordered_map`, в основе которого лежит техника хэширования со средним временем доступа к элементам $O(1)$.

Приступим к примерам:

```
map<string, int> m;
m["monkey"] = 4;
m["banana"] = 3;
m["harpsichord"] = 9;
cout << m["banana"] << "\n"; // 3
```

Если в `map` такого ключа нет, то он создается и в него запишется заданное значение.

При простом обращении к элементу по несуществующему ключу, помимо создания ключа, будет присвоено значение по умолчанию. Например, в следующем коде в отображение добавляется ключ "aybabt" со значением 0.

```
map<string, int> m;
cout << m["aybabt"] << "\n"; // 0
```

)

Функция `count` проверяет, существует ли ключ в отображении:

```
if (m.count("aybabt")) {
    // работаем с ключом
}
```

А так можно напечатать все ключи и значения отображения:

```
for (auto x : m) {
    cout << x.first << " " << x.second << "\n";
}
```

Priority Queue

Очередь с приоритетом это мультимножество(или множество), которое поддерживает вставку, а также извлечение и удаление минимального или максимального элемента(в зависимости от очереди). Вставка и удаление занимают $O(\log n)$, а извлечение $O(1)$.

Очередь с приоритетом обычно основана на heap structure, а не сбалансированном двоичном дереве, что в разы проще.

По умолчанию, элементы в C++ PQ отсортированы по убыванию, и все же это реально найти и удалить наибольший элемент в очереди. Код ниже это демонстрирует:

```
priority_queue<int> q;
q.push(3);
q.push(5);
q.push(7);
q.push(2);
cout << q.top() << "\n"; // 7
q.pop();
cout << q.top() << "\n"; // 5
q.pop();
q.push(6);
cout << q.top() << "\n"; // 6
q.pop();
```

Для того, чтобы PQ поддерживал поиск и удаление наименьшего элемента, можно сделать следующее:

```
priority_queue<int, vector<int>, greater<int>> q;
```

Algorithms