Структуры данных

Наумов Д.А., доц. каф. КТ

Алгоритмы и структуры данных, 2021

Содержание лекции

Массивы

- статические
- динамические - массив, размер которого можно изменять в процессе выполнения программы.

```
//пример статического массива
int array[n];

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

Массивы

Доступ к элементам осуществляется так же, как в обычном массиве:

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

Еще один способ создать вектор – перечислить все его элементы:

```
vector<int> v = {2,4,2,5,1};
```

Можно также задать число элементов и их начальное значение:

```
vector<int> a(8); // размер 8, начальное значение 0
vector<int> b(8,2); // размер 8, начальное значение 2
```

Массивы

Функция *size* возвращает число элементов вектора. В следующем коде мы обходим вектор и печатаем его элементы:

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

Обход вектора можно записать и короче:

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

Массивы

Функция **back** возвращает последний элемент вектора, а функция **pop_back** удаляет последний элемент:

```
vector<int> v = {2,4,2,5,1};

cout << v.back() << "\n"; // 1
v.pop_back();
cout << v.back() << "\n"; // 5
```

Векторы реализованы так, что функции *push_back* и *pop_back* в среднем имеют сложность $O(1)$.

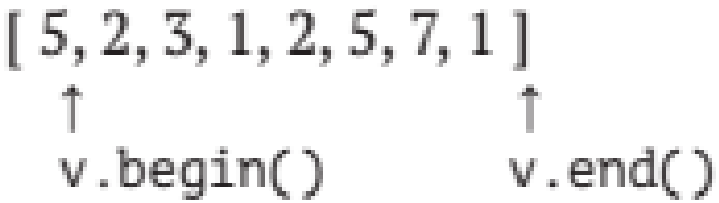
На практике работать с вектором почти так же быстро, как с массивом.

Итераторы

Итератором называется переменная, которая указывает на элемент структуры данных.

Итератор *begin* указывает на первый элемент структуры, а итератор end – на позицию за последним элементом.

Например, в случае вектора **v**, состоящего из восьми элементов, ситуация может выглядеть так:



Обратите внимание на асимметрию итераторов:

- begin() указывает на элемент, принадлежащий структуре данных,
- end() ведет за пределы структуры данных.

Диапазон

Диапазоном называется последовательность соседних элементов структуры данных.

Чаще всего диапазон задается с помощью двух итераторов:

- указывающего на первый элемент.
- указывающего на позицию за последним элементом.

В частности, итераторы *begin()* и *end()* определяют диапазон, содержащий все элементы структуры данных.

Функции из стандартной библиотеки C++ обычно применяются к диапазонам. В следующем фрагменте сначала вектор сортируется, затем порядок его элементов меняется на противоположный, и, наконец, элементы перемешиваются в случайном порядке.

```
sort(v.begin(),v.end());
reverse(v.begin(),v.end());
random_shuffle(v.begin(),v.end());
```

Диапазон

К элементу, на который указывает итератор, можно обратиться, воспользовавшись оператором *. В следующем коде печатается первый элемент вектора:

```
cout << *v.begin() << "\n";
```

Более полезный пример: функция **lower_bound** возвращает итератор на первый элемент отсортированного диапазона, значение которого не меньше x, а функция **upper_bound** – итератор на первый элемент, значение которого не больше x:

```
vector<int> v = {2,3,3,5,7,8,8,8};

auto a = lower_bound(v.begin(),v.end(),5);
auto b = upper_bound(v.begin(),v.end(),5);

cout << *a << " " << *b << "\n"; // 5 7
```

В них применяется двоичный поиск, так что для поиска запрошенного элемента требуется логарифмическое время. Если искомый элемент не найден, то функция возвращает итератор на позицию, следующую за последним элементом диапазона.

Диапазон

В стандартной библиотеке C++ много полезных функций, заслуживающих внимания.

Например, в следующем фрагменте создается вектор, содержащий уникальные элементы исходного вектора в отсортированном порядке:

```
sort(v.begin(),v.end());
v.erase(unique(v.begin(),v.end()),v.end());
```

Двусторонняя очередь

Двусторонней очередью (деком) называется динамический массив, допускающий эффективные операции с обеих сторон.

Как и вектор, двусторонняя очередь предоставляет функции **push_back** и **pop_back**, но вдобавок к ним функции **push_front** и **pop_front**.

Пример использования:

```
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]
```

Операции двусторонней очереди в среднем имеют сложность $O(1)$. Однако постоянные множители для них больше, чем для вектора, поэтому использовать двусторонние очереди имеет смысл, только когда требуется выполнять какието действия на обеих сторонах структуры.

Стек

C++ предоставляет также специализированные структуры данных, по умолчанию основанные на двусторонней очереди.

Для стека определены функции **push** и **pop**, позволяющие вставлять и удалять элементы в конце структуры, а также функция **top**, возвращающая последний элемент без удаления:

```
stack<int> s;

s.push(2); // [2]
s.push(5); // [2,5]

cout << s.top() << "\n"; // 5

s.pop(); // [2]

cout << s.top() << "\n"; // 2
```

Очередь

В случае **очереди** элементы вставляются в начало, а удаляются из конца.

Для доступа к первому и последнему элементам служат функции **front** и **back**.

```
queue<int> q;

q.push(2); // [2]
q.push(5); // [2,5]

cout << q.front() << "\n"; // 2

q.pop(); // [5]

cout << q.back() << "\n"; // 5
```

Множества

Множеством называется структура данных, в которой хранится набор элементов. Основные операции над множествами:

- вставка,
- поиск
- удаление.

Множества реализованы так, что все эти операции эффективны, что часто позволяет улучшить время работы алгоритмов, в которых множества используются.

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

- **set** основана на сбалансированном двоичном дереве поиска, его операции работают за время $O(logn)$,
- **unordered_set** основана на хештаблице и работает в среднем $O(1)$.

Обе структуры эффективны, и во многих случаях годится любая. Поскольку используются они одинаково, в примерах мы ограничимся только структурой *set*.

Множества

В показанном ниже коде создается множество, содержащее целые числа, и демонстрируются некоторые его операции.

- функция **insert** добавляет элемент во множество;
- функция **count** возвращает количество вхождений элемента во множество;
- функция **erase** удаляет элемент из множества.

```
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
```

Множества

Важным свойством множеств является тот факт, что все их элементы различны. Следовательно, функция **count** всегда возвращает 0 (если элемент не принадлежит множеству) или 1 (если принадлежит), а функция insert никогда не добавляет элемент во множество, если он в нем уже присутствует. Это демонстрируется в следующем фрагменте:

```
set<int> s;

s.insert(3);
s.insert(3);
s.insert(3);

cout << s.count(3) << "\n"; // 1
```

Множества

Множество в основном можно использовать как вектор, однако доступ к элементам с помощью оператора [] невозможен. В следующем коде печатается количество элементов во множестве, а затем эти элементы перебираются:

```
cout << s.size() << "\n";

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

Множества

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 множества

Основное различие между двумя структурами множества в C++ – то, что **set** упорядочено, а **unordered_set** не упорядочено. Поэтому если порядок элементов важен, то следует пользоваться структурой *set*.

Рассмотрим задачу о нахождении наименьшего и наибольшего значений во множестве. Чтобы сделать это эффективно, необходимо использовать структуру *set*.

Поскольку элементы отсортированы, найти наименьшее и наибольшее значения можно следующим образом:

```
auto first = s.begin();
auto last = s.end(); last--;

cout << *first << " " << *last << "\n";
```

Отметим, что поскольку *end()* указывает на позицию, следующую за последним элементом, то необходимо уменьшить итератор на единицу.

Упорядоченные множества

В структуре **set** имеются также функции **lower_bound(x)** и **upper_bound(x)**, которые возвращают итератор на наименьший элемент множества, значение которого не меньше x или больше x соответственно. Если искомого элемента не существует, то обе функции возвращают **end()**.

```
cout << *s.lower_bound(x) << "\n";
cout << *s.upper_bound(x) << "\n";
```

Мультимножества

В отличие от множества, в **мультимножество** один и тот же элемент может входить несколько раз.

В C++ имеются структуры **multiset** и **unordered_multiset**, похожие на **set** и **unordered**$_s et$.

В следующем коде в мультимножество три раза добавляется значение 5.

```
multiset<int> s;

s.insert(5);
s.insert(5);
s.insert(5);

cout << s.count(5) << "\n"; // 3
```

Мультимножества

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"; // 2
```

Отметим, что во временной сложности функций **count** и **erase** имеется дополнительный множитель $O(k)$, где k -– количество подсчитываемых (удаляемых) элементов. В частности, подсчитывать количество копий значения в мультимножестве с помощью функции count неэффективно.

Отображения

Отображением называется множество, состоящее из пар ключ-значение. Отображение можно также рассматривать как обобщение массива.

Если в обыкновенном массиве ключами служат последовательные целые числа 0, 1,..., $n1$, где n – размер массива, то в отображении ключи могут иметь любой тип и необязательно должны быть последовательными.

В стандартной библиотеке C++ есть две структуры отображений, соответствующие структурам множеств: в основе map лежит сбалансированное двоичное дерево со временем доступа к элементам $O(logn)$, а в основе **unordered_map** – техника хеширования со средним временем доступа к элементам O(1).

Отображения

В следующем фрагменте создается отображение, ключами которого являются строки, а значениями – целые числа:

```
map<string,int> m;

m["monkey"] = 4;
m["banana"] = 3;
m["harpsichord"] = 9;

cout << m["banana"] << "\n"; // 3
```

Если в отображении нет запрошенного ключа, то он автоматически добавляется, и ему сопоставляется значение по умолчанию. Например, в следующем коде в отображение добавляется ключ «aybabtu» со значением 0.

```
map<string,int> m;

cout << m["aybabtu"] << "\n"; // 0
```

Отображения

Функция **count** проверяет, существует ли ключ в отображении.

```
if (m.count("aybabtu")) {
    //ключ "aybabtu" есть в отображении
}
```

В следующем коде печатаются все имеющиеся в отображении ключи и значения:

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

Очереди с приоритетом

Очередь с приоритетом – это мультимножество, которое поддерживает вставку, а также извлечение и удаление минимального или максимального элемента (в зависимости от типа очереди). Вставка и удаление занимают время $O(logn)$, а извлечение – время $O(1)$.

Очередь с приоритетом обычно основана на структуре пирамиды (**heap**), представляющей собой двоичное дерево специального вида. Структура **multiset** и так предоставляет все операции, которые определены в очереди с приоритетом, и даже больше, но у очереди с приоритетом есть достоинство – меньшие постоянные множители в оценке временной сложности.

Поэтому если требуется только найти минимальный или максимальный элемент, то лучше использовать очередь с приоритетом, а не множество или мультимножество.

Очереди с приоритетом

По умолчанию элементы очереди с приоритетом в C++ отсортированы в порядке убывания, так что поддерживаются поиск и удаление наибольшего элемента, что и продемонстрировано в следующем коде:

```
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();
```

Множества, основанные на политиках

Компилятор g++ предоставляет также несколько структур данных, не входящих в стандартную библиотеку C++. Они называются **структурами, основанными на политиках** (policy based structure).

Для их использования в программу нужно включить такие строки:

```
#include <ext/pb_ds/assoc_container.hpp>

using namespace __gnu_pbds;
```

После этого можно определить структуру данных **indexed_set**, которая похожа на множество, но допускает индексирование как массив. Для значений типа *int* определение выглядит так:

```
typedef tree <int, null_type, less<int>, rb_tree_tag,
    tree_order_statistics_node_update> indexed_set;
```

Множества, основанные на политиках

А создается множество так:

```
indexed_set s;

s.insert(2);
s.insert(3);
s.insert(7);
s.insert(9);
```

Особенность этого множества состоит в том, что доступ можно осуществлять по индексу, который элемент имел бы в отсортированном массиве.

Функция **find_by_order** возвращает итератор, указывающий на элемент в заданной позиции:

```
auto x = s.find_by_order(2);
cout << *x << "\n"; // 7
```

Множества, основанные на политиках

Функция order_of_key возвращает позицию заданного элемента:

```
cout << s.order_of_key(7) << "\n"; // 2
```

Если элемент отсутствует во множестве, то мы получим позицию, в которой он находился бы, если бы присутствовал:

```
cout << s.order_of_key(6) << "\n"; // 2
cout << s.order_of_key(8) << "\n"; // 3
```

Время работы обеих функций логарифмическое.

Эксперименты

Приведем некоторые результаты, касающиеся практической эффективности описанных выше структур данных.

Хотя временная сложность – отличный инструмент, она не всегда сообщает всю правду об эффективности, поэтому имеет смысл провести эксперименты с настоящими реализациями и наборами данных.

Многие задачи можно решить, применяя как множества, так и сортировку. Важно понимать, что алгоритмы на основе сортировки обычно гораздо быстрее, даже если это не очевидно из одного лишь анализа временной сложности

Эксперименты

В качестве примера рассмотрим **задачу о вычислении количества уникальных элементов вектора**.

- Одно из возможных решений – поместить все элементы во множество и вернуть размер этого множества. Поскольку порядок элементов не важен, можно использовать как *set*, так и *unordered_set*.
- Можно решить задачу и подругому: сначала отсортировать вектор, а затем обойти его элементы. Подсчитать количество уникальных элементов отсортированного вектора просто.

Эксперименты

В табл. приведены результаты эксперимента, в котором оба алгоритма тестировались на случайных векторах чисел типа *int*.

Размер входных данных	set (с)	unordered_set (с)	Сортировка (с)
10^6	0.65	0.34	0.11
$2 \cdot 10^6$	1.50	0.76	0.18
$4 \cdot 10^6$	3.38	1.63	0.33
$8 \cdot 10^6$	7.57	3.45	0.68
$16 \cdot 10^6$	17.35	7.18	1.38

- Оказалось, что алгоритм на основе *unordered_set* примерно в два раза быстрее алгоритма на основе *set*, а алгоритм на основе сортировки быстрее алгоритма на основе *set* более чем в 10 раз.
- Отметим, что временная сложность обоих алгоритмов равна $O(nlogn)$, и тем не менее алгоритм на основе сортировки работает гораздо быстрее.
- Причина в том, что сортировка – простая операция, тогда как сбалансированное двоичное дерево поиска, применяемое в реализации set, – сложная структура данных.

Сравнение отображения и массива

Отображения -– удобные структуры данных, по сравнению с массивами, поскольку позволяют использовать индексы любого типа, но и постоянные множители велики.

В следующем эксперименте мы создали вектор, содержащий n случайных целых чисел от 1 до 106, а затем искали самое часто встречающееся значение путем подсчета числа вхождений каждого элемента.

Сначала мы использовали отображения, но поскольку число 106 достаточно мало, то можно использовать и массивы.

Сравнение отображения и массива

Результаты эксперимента сведены в табл.

Размер входных данных	map (с)	unordered_map (с)	Массив (с)
10^6	0.55	0.23	0.01
$2 \cdot 10^6$	1.14	0.39	0.02
$4 \cdot 10^6$	2.34	0.73	0.03
$8 \cdot 10^6$	4.68	1.46	0.06
$16 \cdot 10^6$	9.57	2.83	0.11

Хотя *unordered_map* примерно в три раза быстрее *map*, массив все равно почти в 100 раз быстрее.

Таким образом, по возможности следует пользоваться массивами, а не отображениями. Особо отметим, что хотя временная сложность операций *unordered_map* равна $O(1)$, скрытые постоянные множители, характерные для этой структуры данных, довольно велики.

Сравнение очереди с приоритетом и мультимножества

Верно ли, что очереди с приоритетом действительно быстрее мультимножеств? Чтобы выяснить это, мы провели еще один эксперимент.

Мы создали два вектора, содержащие n случайных чисел типа *int*. Сначала мы добавили все элементы первого вектора в структуру данных, а затем обошли второй вектор и на каждом шаге удаляли наименьший элемент из структуры данных и добавляли в нее новый элемент.

Сравнение очереди с приоритетом и мультимножества

Результаты эксперимента представлены в табл.

Размер входных данных	multiset (с)	priority_queue (с)
10^6	1.17	0.19
$2 \cdot 10^6$	2.77	0.41
$4 \cdot 10^6$	6.10	1.05
$8 \cdot 10^6$	13.96	2.52
$16 \cdot 10^6$	30.93	5.95

Оказалось, что с помощью очереди с приоритетом эта задача решается примерно в пять раз быстрее, чем с помощью мультимножества.