Решение задачи поиска элемента по значению средствами стандартной библиотеки C++

Казачкова А.А.[1], Огнева М.В.[2]
[1]*kazachkova.anna@gmail.com*, [2]*ognevamv@gmail.com*,
Саратовский государственный университет имени Н.Г.Чернышевского

Поиск является одной из фундаментальных операций и встречается во многих прикладных задачах. В статье рассмотрены различные инструменты стандартной библиотеки Си++, которые можно использовать для организации поиска. Грамотный выбор подходящего инструмента может значительно повлиять на эффективность программы.

Ключевые слова: алгоритмы поиска, стандартная библиотека Си++, STL, программирование

Поиск является одной из наиболее фундаментальных операций, присущей огромному количеству вычислительных задач. Под поиском обычно понимают нахождение какой-либо конкретной информации в имеющихся данных часто большого объема. Самый простой поиск - это поиск по совпадению, когда целью поиска является нахождение всех записей, подходящих к заданному ключу поиска (ключом может быть какое-то слово, например, фамилия, город, профессия). Кроме поиска по совпадению существует поиск по близости, поиск по интервалу, а также поиск по нескольким условиям, конъюнктивным, дизъюнктивными или смешанной природы. Кроме того, существуют более сложные виды поиска, например, найти все статьи по данной теме.

Будем рассматривать простой вариант поиска элементов из набора по совпадению.

Если мы ничего не знаем о данной последовательности, то единственный способ найти там заданный элемент – это полный перебор всех элементов. Такой поиск называется линейным или последовательным и имеет время выполнения O(n).

В заголовочном файле algorithm стандартной библиотеки C++ есть функция find, принимающая два итератора (начало и конец поиска) и искомый элемент, и возвращающая итератор первого вхождения искомого элемента или итератор конца поиска, если элемент отсутствует.

```
vector<int> v = { 1, 5, -1, 0, 3 };
int x = 0;
auto it = find(v.begin(), v.end(), x);
if (it != v.end())
  cout << it - v.begin();
else
  cout << "no such element";
```

Улучшить это время можно, если мы имеем отсортированную последовательность. В этом случае можно использовать так называемый бинарный поиск (двоичный поиск, метод деления пополам). При бинарном поиске искомый ключ сравнивается с ключом среднего элемента в массиве. Если они равны, то поиск успешен. В противном случае поиск осуществляется

аналогично в левой (если ключ меньше, чем средний элемент) или правой (если ключ больше, чем правый элемент) частях массива. Бинарный поиск выполняется за O(logn)

В C++ существует встроенная функция binary_search, которая проверяет, есть ли в отсортированном диапазоне элемент, равный указанному значению. Функция возвращает значение true если указанный элемент имеется в диапазоне и false – если такой элемент отсутствует. Функция также находится в заголовочном файле algorithm.

```
vector<int> v = { 1, 5, -1, 0, 3 };
int x = 0;
sort(v.begin(), v.end());
cout << boolalpha << binary_search(v.begin(), v.end(), x);
```

Для получения итератора найденного элемента можно использовать функцию lower_bound (возвращает итератор первого элемента, который не меньше искомого), дополнительно проверяя, что найден именно искомый элемент.

```
vector<int> v = { 1, 5, -1, 0, 3 };
int x = 0;
sort(v.begin(), v.end());
for (auto x : v)
  cout << x << ' ';
cout << endl;
auto it = lower_bound(v.begin(), v.end(), x);
if (it != v.end() && *it == x)
  cout << "Элемент " << *it << "\nНомер " << it - v.begin();
else
  cout << "no such element";
```

Метод бинарного поиска довольно эффективен, он работает за O(logn) поскольку каждая итерация вдвое уменьшает число элементов, среди которых нам нужно продолжать поиск. Однако, поскольку данные хранятся в массиве, операции вставки и удаления элементов, необходимые для поддержания отсортированного порядка, не столь эффективны. Для более эффективной работы можно использовать деревья двоичного поиска.

Двоичное дерево - это дерево, каждый узел которого имеет не более двух сыновей. Двоичное дерево называется деревом бинарного поиска, если для каждого его узла выполняется свойство: значения всех узлов в левом поддереве меньше значения данного узла, а значения всех узлов в правом поддереве больше значения данного узла. На этом свойстве основан способ вставки элементов в дерево бинарного поиска: мы сравниваем новое значения со значениями текущих узлов, начиная с корня, и если новое значение меньше текущего, идем в левое поддерево, а если больше, то в правое.

Если такое дерево является идеально сбалансированным, то есть для каждого узла количество элементов в левом и правом поддеревьях отличается не больше чем на единицу, то время выполнения такого поиска есть O(logn). Однако в худшем случае, когда, например, мы добавляем в него упорядоченную последовательность данных, оценка времени будет O(n).

Красно-чёрным называется дерево бинарного поиска, у которого каждому узлу сопоставлен дополнительный атрибут – цвет и для которого выполняются следующие свойства:

1. Каждая вершина – либо красная, либо черная
2. Каждый лист – черный
3. Если вершина красная, оба ее ребенка черные
4. Все пути, идущие от корня к листьям, содержат одинаковое количество черных вершин

В стандартной библиотеке C++ есть коллекция, реализующая абстрактную структуру данных множество с помощью красно-чёрных деревьев, это set. Для использования необходимо подключить одноимённый заголовочный файл.

```
set<int> st = { 1, 5, -1, 0, 3 };
int x = 0;
auto it = st.find(x);
if (it != st.end())
  cout << *it;
else
  cout << "no such element";
```

Для поиска итератора можно использовать методы lower_bound и upper_bound. Но это должны быть именно методы объекта-множества, т.к. если применить функции lower_bound или upper_bound из библиотеки algorithm, время работы станет линейным. Это связано с тем, что алгоритмы не могут использовать особенности внутреннего устройства множества, для логарифмического времени работы им требуется итератор произвольного доступа, недоступный для set.

```
set<int> st = { 1, 5, -1, 0, 3 };
int x = 2;
auto it = st.lower_bound(x);
// auto it = lower_bound(st.begin(), st.end(), x); // МЕДЛЕННО
if (it != st.end())
  cout << *it;
else
  cout << "no such element";
```

Предположим однако, что мы хотим работать с неотсортированным массивом, а время O(n) нас не устраивает? Самым очевидным кажется провести предварительно сортировку, однако ее время только в самом лучшем частном случае окажется O(n), что в целом даст нам то же самое. Другой вариант - хранить элементы так, чтобы потом было легко их найти. Самый эффективный поиск получится, если использовать так называемый метод прямой адресации. Для использования этого метода необходимо, чтобы одно из полей было уникальным числом, которое при записи в массив будет считаться номером ячейки. То есть мы сразу же кладем наши данные на определенное место и можем найти их за O(1). Однако на практике это не всегда возможно осуществить. Во-первых, не во всех случаях может существовать такое поле. Во-вторых, обычно числовые идентификаторы бывают многозначными и во много десятков раз больше, чем общее число записей. То есть, в качестве индексов мы

собираемся использовать числа в районе миллиона, а значит, иметь по крайней мере миллион ячеек, в то время как реальных записей будет около тысячи.

Неким компромиссом в данном случае могут служить так называемые хеш-таблицы. Хеш-таблица – это таблица, в которой адресация задается с помощью хеш-функции. Хеш-функция - это функция, которая переводит ключ в индекс в таблице с учетом нужного количества элементов. В этом случае решается вопрос неэффективного использования памяти, однако появляется следующая проблема. Предположим, например, что у нас есть хеш-функция h(key)=key mod 100, и мы имеем два ключевых значения – 34562 и 98162. Нетрудно заметить, что оба они хешируются в одно и то же значение – 62. Такая проблема называется коллизией. Хорошая хеш-функция должна минимизировать коллизии и равномерно распределять данные по таблице.

В стандартной библиотеке реализация абстрактной структуры данных множество с помощью хэш-таблиц появилась в стандарте C++11 и называется unordered_set. Находится в одноимённом заголовочном файле.

```
int main() {
  unordered_set<int> st = { 1, 5, -1, 0, 3 };
  int x = 0;
  auto it = st.find(x);
  if (it != st.end())
    cout << *it;
  else
    cout << "no such element";
```

Когда количество элементов с одинаковыми значениями хэш-функции в каком-то блоке (load_factor) превышает максимально допустимое (max_load_factor), происходит автоматическое увеличение количества блоков (bucket_count) с пересчётом хэшей. Пересчёт выполняется за линейное время, но позволяет дальнейшие операции вставки и поиска снова осуществлять за константное время.

Таким образом, существует целый ряд инструментов, грамотное применение которых позволяет писать эффективные программы.