Лекция по курсу «Алгоритмы и структуры данных» / «Технологии и методы программирования»

Алгоритмы поиска:

последовательный, бинарный и интерполяционный поиск
возможности стандартной библиотеки

Мясников Е.В.

Поиск

Пусть имеется набор из N записей, состоящих из одного ключевого поля и и, возможно, множества других полей. Задача состоит в нахождении записи с заданным ключом .

Ключ является аргументом функции поиска. При удачном поиске функция должна найти и вернуть запись с указанным ключом (индекс, указатель, …). При неудачном – вернуть предопределенное значение.

Поиск часто является одним из наиболее емких по времени частей программы. Применение быстрых методов поиска позволяет значительно повысить и эффективность программ в целом.

Классификация методов:

- внутренние и внешние
- статические и динамические
- сравнение непосредственно значений или результатов их преобразований
- …

Поиск

Поиск может осуществляться:
- в **одномерном** пространстве
- в **многомерном** пространстве

Задача поиска может ставиться как:
- Поиск **точно** совпадающего значения
- Поиска значений **в диапазоне** (или окрестности)
- Поиск **ближайшего** (или k ближайших) к заданному

При поиске ближайшего соседа возможны постановки:
- поиск **точного** ближайшего соседа
- **приближенный** поиск ближайшего

В рамках курса изучим

Поиск по ключу в одномерных массивах:
- последовательный
- бинарный
- интерполяционный

Поиск с использованием бинарных деревьев:
- деревья поиска
- АВЛ – деревья

Поиск с использованием хеш-таблиц:
- числовых данных
- символьных последовательностей

Поиск последовательностей с использованием префиксных деревьев

Поиск в многомерных пространствах с использованием BSPT:
- kd – деревья
- ball – деревья
- vp – деревья

…

Последовательный поиск

Синонимы:
- последовательный поиск
- перебор
- exhaustive search
- brute force approcach

Идея: начать с начала и продолжать, пока не будет найден искомый ключ или не будет достигнут конец набора данных

Пути улучшения:
- группировка повторяющихся элементов
  (если ключи повторяются, составные ключи)
- распараллеливание
- вероятностное перераспределение (если запросы не равновероятны, то использование счетчиков и перемещение часто запрашиваемых записей ближе к началу набора)
- упорядочивание по ключу …

Бинарный поиск

Поиск в упорядоченном наборе !

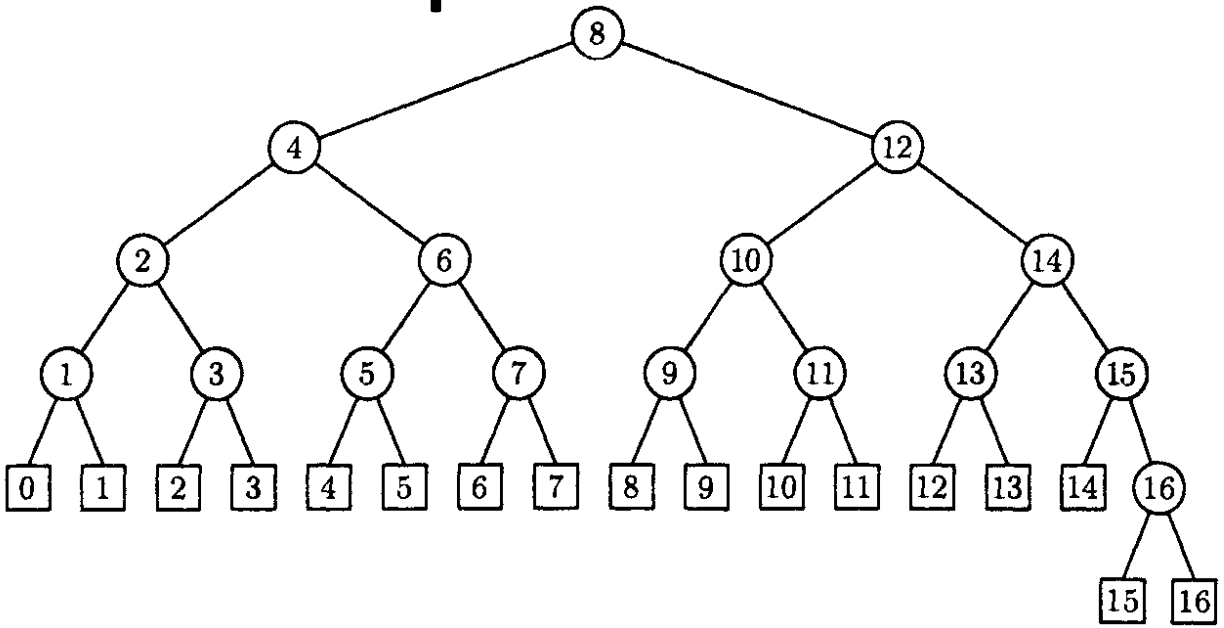
Синонимы:

- бинарный поиск
- метод деления пополам
- метод дихотомии
- логарифмический поиск

Идея: сравниваем K с ключом в середине набора и определяем, в какой из оставшихся частей может находиться ключ, после чего повторяем процедуру рекурсивно

Время поиска: log(N)

Бинарный поиск



B1. [Инициализация.] Установить $l \leftarrow 1$, $u \leftarrow N$.

B2. [Получение средины.]Если K имеется в таблице, то справедливо $K_l \leq K \leq K_u$. Если $u < l$, алгоритм завершается неудачно;
в противном случае следует установить $i \leftarrow \lfloor (l+u)/2 \rfloor$, чтобы i соответствовало примерно средине рассматриваемой части таблицы.

B3. [Сравнение.] Если $K < K_i$, перейти к шагу B4; если $K > K_i$, перейти к шагу B5; если $K = K_i$, алгоритм успешно завершается.

B4. [Изменение u.] Установить $u \leftarrow i - 1$ и перейти к шагу B2.

B5. [Изменение l.] Установить $l \leftarrow i + 1$ и перейти к шагу B2.

Бинарный поиск. Функция bsearch

```
void* bsearch(
        const void* key,     // указатель на элемент для поиска
        const void* ptr,     // указатель на массив для исследования
        std::size_t count,   // количество элементов в массиве
        std::size_t size,    // размер каждого элемента в массиве в байтах
        comp                 // функция сравнения
      );

int cmp(const void *a, const void *b);   // тип функции сравнения
```

Пример:

```
int cmpint(const void* a, const void* b) {  return *(int*)a - *(int*)b;  }

int arr[] = { 1,2,3,4,5,6,7,8,9,10 };
int val = 7;
void* res = bsearch(&val, arr, sizeof(arr) / sizeof(arr[0]), sizeof(arr[0]), cmpint);
cout      << "\nindex: " << ((int*)(res) - arr)
          << "\nfound :" << *(int*)res;
```

```
index: 6
found :7
```

Бинарный поиск. Функция bsearch

```
void* bsearch(
        const void* key,     // указатель на элемент для поиска
        const void* ptr,     // указатель на массив для исследования
        std::size_t count,   // количество элементов в массиве
        std::size_t size,    // размер каждого элемента в массиве в байтах
        comp                 // функция сравнения
    );
int cmp(const void *a, const void *b);   // тип функции сравнения
```

Пример:

```
int cmp(const void* a, const void* b)
{ return strcmp(*(const char**)a, *(const char**)b); }

const char* names [] = { "anna", "boris", "daniil", "elena", "george", "ivan",
                         "nikolay", "petr", "richard", "vladimir" };
size_t count = sizeof(names)/sizeof(names[0]), elmsz = sizeof(names[0]);
const char* val = "ivan";
void* res = bsearch(&val, names, count, elmsz, cmp);
cout    << "\nindex: " << ((const char**)(res) - names)
        << "\nfound :" << *(const char**)res;
```

```
index: 5
found :ivan
```

Бинарный поиск. Операции бинарного поиска

lower_bound // возвращает итератор на первый элемент
// не меньший заданного

```
template< class ForwardIt, class T >
ForwardIt lower_bound( ForwardIt first, ForwardIt last, const T& value );

template< class ForwardIt, class T, class Compare >
ForwardIt lower_bound( ForwardIt first, ForwardIt last, const T& value, Compare comp );
```

upper_bound // … больший заданного

binary_search // определяет, есть ли заданный элемент в диапазоне

```
template< class ForwardIt, class T >
bool binary_search( ForwardIt first, ForwardIt last, const T& value );

template< class ForwardIt, class T, class Compare >
bool binary_search( ForwardIt first, ForwardIt last,  const T& value, Compare comp );
```

equal_range // возвращает диапазон элементов,
// соответствующих заданному

Бинарный поиск. Операция equal_range

```
template< class ForwardIt, class T >
std::pair<ForwardIt,ForwardIt> equal_range  (
            ForwardIt first,      // диапазон элементов
            ForwardIt last,
            const T& value        // искомое значение
      );

template< class ForwardIt, class T, class Compare >
std::pair<ForwardIt, ForwardIt>
   equal_range (
            ForwardIt first,      // диапазон элементов
            ForwardIt last,
            const T& value,       // искомое значение
            Compare comp          // предикат, возвращающий истину, если
                                  // 1-ый элемент меньше 2-го
       );
```

Операция equal_range. Пример

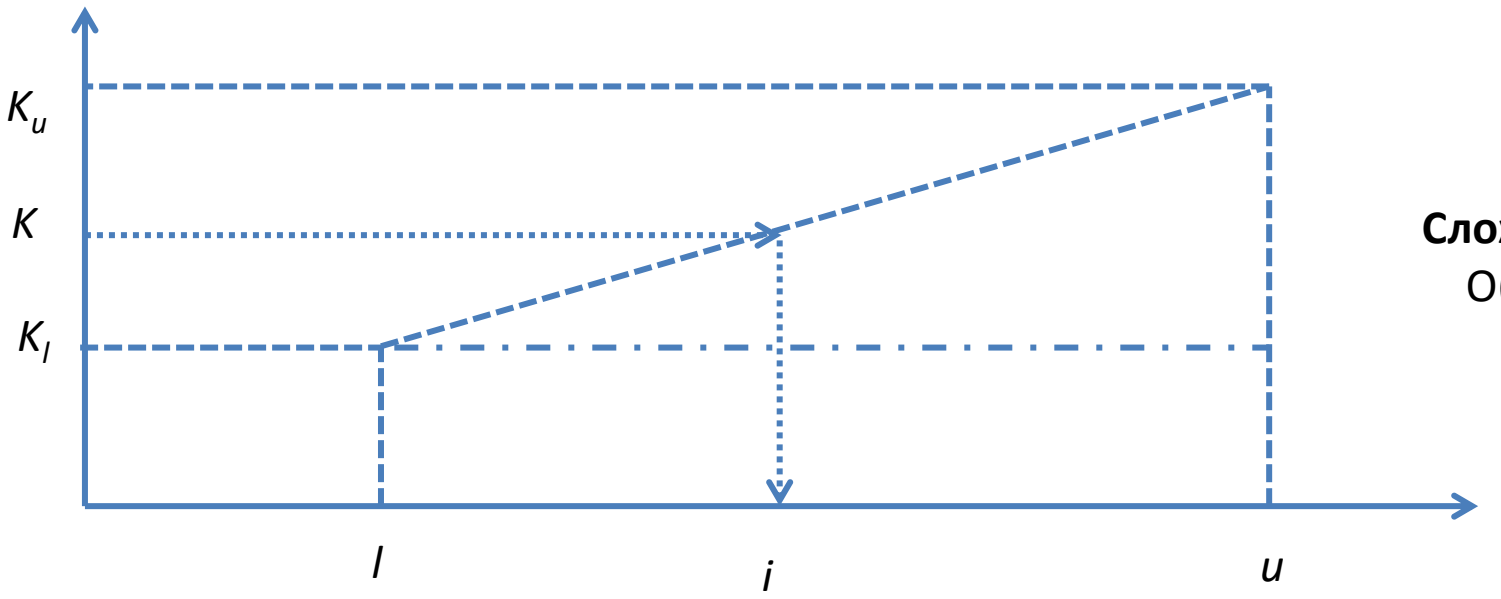
```
struct Stud
{
        string name;
        int mark;
        bool operator< (const Stud& s) const { return mark < s.mark; }
};

vector<Stud> vec = { {"petr",3}, {"ivan",4}, {"anna",4}, {"richard",4}, {"petr",5} };
Stud val = { "", 4 };

auto res = equal_range(vec.begin(), vec.end(), val);

for (auto it = res.first; it != res.second; it++)
        cout << "\n<" << it->name << ":  " << it->mark << ">";
```

```
<ivan:  4>
<anna:  4>
<richard:  4>
```

Интерполяционный поиск

Как ищет человек в телефонной книге?

Предположение:
ключи распределены по закону близкому к арифметической прогрессии



Сложность:
O(log log n)

$$\frac{K_u - Kl}{u - l} = \frac{K - Kl}{i - l}$$

$$i = \frac{K - Kl}{K_u - Kl}(u - l) + l$$

Вопрос:
если ключи распределены по закону, близкому к геометрической прогрессии?