ТЪРСЕНЕ НА ДАННИ. ПОДРЕДЕНИ И НЕПОДРЕДЕНИ ДАННИ. МЕТОДИ ЗА ТЪРСЕНЕ

ВТУ, 16.10.2008 г.

ТЪРСЕНЕ - СЪЩНОСТ И СВОЙСТВА

Търсене – фундаментална дейност, извършвана ежедневно и ежечасно от всеки индивид

Примери:

- Търсим телефонен номер;
- Търсим изгубени ключове;
- Търсим дадено телевизионно предаване;
- Търсим нещо за ядене (сложност – експоненциална :));
- Търсим си белята (сложност – константна :)).

Добре позната дейност, но видовете търсене се класифицират доста трудно.

ВИДОВЕ ТЪРСЕНЕ

В процеса на търсене често използваме натрупания опит, а в непозната ситуация най-често използваме метода на пробите и грешките. Съществуват достатъчно мощни методи за търсене, които зависят най-вече от конкретните условия.

- Некоректно търсене – да търсим под вола теле;
- Неясно дефинирано търсене – търсене на интересно предаване по телевизията;
- Търсене, ограничено по време – търсене на бомба с часовников механизъм;
- Търсене на движеща се цел – търсене на самолет с радар;
- Търсене в безкрайно множество – търсене на планети в звездни системи.

ОПРОСТЕН МОДЕЛ ЗА ТЪРСЕНЕ

Основни свойства:

- Целта е фиксирана и добре дефинирана;
- Неограничено време;
- Крайно и статично множество;
- Всички сравнения имат еднаква цена;
- Няма грешка при кое да е сравнение;
- Информацията от предходни сравнения се запазва.

За съжаление, този модел невинаги е налице. Често се случва множеството за търсене да е динамично или някои ресурси да са ограничени. Затова е по-добре алгоритмите за търсене да се разглеждат не като независими действия, а като *част от общ пакет фундаментални операции над конкретно множество.*

СЪВКУПНОСТ ОТ ДЕЙСТВИЯ

Удобно е търсенето да се разглежда като елемент на следния комплект операции:

- Инициализиране;
- Търсене;
- Вмъкване;
- Изтриване;
- Обединяване на две множества;
- Сортиране.

Между някои от тези операции има пряка връзка и често се изпълняват едновременно (например търсене–вмъкване, търсене–изтриване, сортиране–търсене и др.)

ПОДРЕДЕНИ ДАННИ

Множество от данни наричаме ***подредено (сортирано)***, ако подреждането на елементите на множеството удовлетворява някаква предварително зададена наредба.

Примери:

1. Множеството на естествените числа – $1 < 2 < 3 < \ldots < n < n + 1 < \ldots$;

2. Буквите в латиницата – $A < B < \ldots < Y < Z$;

3. Дните на седмицата – *понеделник* $<$ *вторник* $< \ldots <$ *неделя.*

Важен момент е, че всеки елемент на множеството трябва да има различна „стойност“ от останалите спрямо зададената наредба. Ако дадено множество е сортирано, върху него могат да се изпълняват доста по-бързо някои от допустимите операции. От друга страна, сортирането също изисква времеви ресурс.

ПОСЛЕДОВАТЕЛНО ТЪРСЕНЕ

Два варианта – търсене в несортирано и сортирано множество.

Нека елементите на множеството са записани в едномерен масив A. Преглеждат се елементите, докато се открие търсения или се обходят всички елементи.

Търсене в несортирано множество:

int Search(int $A[\,]$, int n, int x) {
    for(int $i = 0; i < n; i++$)
        if($A[i] == x$)
            return 1;
    return 0;
}

Сложност при успешно търсене – $O(n)$, при неуспешно – $\Theta(n)$

ПОСЛЕДОВАТЕЛНО ТЪРСЕНЕ

Търсене в сортирано множество:

```
int SearchSort(int A[ ], int n, int x) {
    int i = 0;
    while(i < n && A[i] < x)
        i++;
    if(A[i] == x)
        return 1;
    return 0;
}
```

Сложност при успешно търсене – $O(n)$, при неуспешно – $O(n)$

ТЪРСЕНЕ СЪС СТЪПКА

Търсенето е в сортирано множество. Избираме стъпка k и сравняваме с x елементите $A[k], A[2k], \ldots, A[mk]$, докато $mk \leq n$.

```
int SearchStep(int A[ ], int n, int x, int k) {
    int i = 0;
    while(i < n && A[i] < x)
        i+ = k;
    if(A[i] <= x)
        for(int j = i − k + 1; j ≤ i; j + +)
            if(A[j] == x)
                return 1;
    return 0;
}
```

Сложност при успешно търсене – $O(\sqrt{n})$, при неуспешно – $O(\sqrt{n})$

ДВОИЧНО ТЪРСЕНЕ

Търсим елемента x в сортиран масив. С l и r означаваме левия и десния край на масива.

1. Определяме елемента $y = A[(l+r)/2]$;
2. Сравняваме y и x. Ако $x == y$ – край (да);
3. Ако $x < y$, то $r = (l+r-1)/2$. В противен случай $l = (l+r+1)/2$;
4. Ако l != r – стъпка 1. В противен случай проверяваме дали $A[l] == x$. Ако да – елементът е в масива. В противен случай не е.

ДВОИЧНО ТЪРСЕНЕ

```
int BinSearch(int A[ ], int n, int x) {
    int l = 0, r = n − 1;
    while(l != r) {
        int y = A[(l + r)/2];
        if(x == y) return 1;
        else
            if(x < y)
                r = (l + r − 1)/2;
            else l = (l + r + 1)/2;
    }
    if(A[l] == x)
        return 1;
    return 0;
}
```

Сложност при търсене – $O(\log_2 n)$

ФИБОНАЧИЕВО ТЪРСЕНЕ

Числа на Фибоначи: $F_0 = 0, F_1 = 1, F_{n+1} = F_n + F_{n-1}$

Получаваме редицата $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots\ldots$. За нея е изпълнено

$$\lim_{n\to\infty} \frac{F_{n+1}}{F_n} = \frac{\sqrt{5}+1}{2} \approx 1,618$$

Наподобява двоичното търсене – масивът се разделя на две части, от които едната се отхвърля. Разликата е в елемента, който избираме на всяка стъпка. Ако масивът е с дължина $F_k - 1$, то на всяка стъпка избираме $y = A[F_{k-1}]$. Ако $x < y$, търсим измежду елементите с индекси $1, 2, \ldots, F_{k-1} - 1$. Ако $x > y$, търсим измежду елементите с индекси $F_{k-1} + 1, \ldots, F_k - 1$.

На практика, фибоначиевото търсене строи балансирано двоично дърво, известно като *Фибоначиево дърво*. Дълбочината на това дърво е с 45 % по-голяма от тази на дърветата, които се изграждат при двоичното търсене $\Rightarrow$ можем да очакваме 45 % забавяне спрямо двоичното търсене.