МАТЕМАТИКА И МАТЕМАТИЧЕСКО ОБРАЗОВАНИЕ, 2006
MATHEMATICS AND EDUCATION IN MATHEMATICS, 2006
Proceedings of the Thirty Fifth Spring Conference of
the Union of Bulgarian Mathematicians
Borovets, April 5–8, 2006

ИЗУЧАВАНЕ НА АЛГОРИТМИТЕ ЗА ТЪРСЕНЕ ЧРЕЗ МНОЖЕСТВА

Мария С. Палаханова

Дадено е описание на различни алгоритми за търсене над едномерни и многомерни множества. Описани са алгоритми чрез диаграми с език за програмиране Паскал.

1. Основни дефиниции и забележки, насочени към разрешаване на въпросите за откриване на конкретна информация. Търсенето като проблем може да се представи с три множества:

- Универсално множество U, което съдържа всички подмножества при разрешаването на дадена ситуация.
- Множеството Q на въпросите.
- Множеството A на отговорите.

Дефинираме функция F, която определя отговор на всеки въпрос върху някакво подмножество на универсалното множество U, която ще означим с F: $Q \times 2^U \to A$. За фиксирано, определено подмножество $S \subseteq U$, множеството A_S от всички възможни отговори $A_S = \{F(m,S) | m \in Q\}$ също е определено. Подмножеството $S \subseteq U$ ще представяме с абстрактен тип данни D(S). В практиката съществуват два случая за подмножеството $S \subseteq U$:

1. S е дадено "веднъж за винаги" и не се променя. В този случай имаме търсене в статични типове данни.

2. S не е фиксирано, може да се променя с изтриване на негови елементи или с прибавяне на елементи от U. В този случай имаме търсене в динамични типове данни.

Най-простата и най-често срещана в практиката задача за търсене е така наречената задача за принадлежност. В този случай Q=U е произволно множество, A={true, false} и функцията F, дефинирана като:

$$F(m,S) = \begin{cases} \text{true}, & \text{if } m \in S, \\ \text{false}, & \text{if } m \notin S. \end{cases}$$

Друг основен тип задачи за търсене е задачата за търсене на минимален и максимален елемент в подредено множество. В този случай Q=A=U са произволни подредени множества и F е дефинирана като F(m,S)=min S, за всяко m∈A и S⊆U. В този случай стойностите на F не зависят от m.

По-сложна задача за търсене е така нареченото търсене с частично съответствие. Нека U=E^k, Q = $(E \cup \{*\})^k$, A = 2^U, където E е произволно множество, k $\geq$ 1 е естествено число, $*$ е специален символ, не съдържащ се в E и F е дефинирана като F(a, S) = $\{b \in S : (\forall j)(1 \leq j \leq k)(a_j = b_j$ или $a_j = *\}$. Това означава, че F(a, S) е множеството на всички k-орки, чиито j-ти компонент b_j съвпада с компонент a_j от $a = (a_1, a_2, \ldots, a_k)$ в случая, когато $a_j \neq *$. Този пример принадлежи към задачите за многомерно търсене.

2. Алгоритми за търсене в едномерни множества. В зависимост от начина на реализация на F, различаваме два класа от алгоритми за търсене:

1. Алгоритми, базирани на адресни изчисления, т.е. използват стойността на елемент, за да изчислят неговото място в съответната структура от данни.

2. Алгоритми за асоциативно търсене, които сравняват отделните елементи на U и използват получената информация, за да определят позицията на всеки елемент в съответната структура от данни.

Изборът на подходящ алгоритъм за търсене зависи от типа данни, към който е насочено търсенето и по-късно от специфичните свойства на реализирания абстрактен тип данни D(S). Ако U може да бъде представено изцяло чрез директен достъп в паметта, можем да използваме търсене, базирано на адресни изчисления. В противен случай можем да използваме асоциативно търсене или комбинация от двата вида.

Ако U е определено с |U| = N, тогава подмножеството S $\subseteq$ U може да бъде описано с масив A: array [1...N] of boolean, където A[i] = true, ако i–ят елемент от U е в S. Инициализацията на целия масив отнема време O(N).

В случаите, когато подмножествата S, |S| < m, (m < N) от U са относително малки, използваме хеш таблица, за да намалим размера на масива. Хеш таблицата е съставена от масив A[0...m-1] и така наречена хеш функция h(x): U$\rightarrow$[0,m]. Даден елемент x$\in$U, принадлежащ на S е поставен в хеш таблицата на позиция A[h(x)] или в най-близката до тази позиция, така че да бъде открит бързо. Ако два или повече елемента от U се разпределят върху еднакви позиции от хеш таблицата, се получава колизия, която се разрешава с някои известни методи за разрешаване на колизии – отворено или верижно адресиране [1].

Ако се прави опит да се вмъкне нов елемент в място от масива, което е вече заето, новият елемент трябва да се съхрани в друг незает елемент от масива. Най-простият алгоритъм е методът на линейните проби с прилагане на хеш-функция за деление с остатък. Търсенето продължава до първия срещнат незает елемент. Този метод използва повече памет от верижния, т.е. таблицата трябва да побере максималния брой елементи плюс 20% резервно пространство, затова е неприемлив за използване.

При верижното адресиране всеки елемент има свързан с него списък за препълване, в който се съхраняват група от елементи, чиито ключове се хешират в един и същ индекс на масива. Той се представя като динамично разпределен свързан списък. И при двата метода средното време за търсене се намалява с увеличаване размера на хеш-таблицата.

Масивът е най-простия пример на структура от данни, поддържаща асоциативно търсене. Всеки елемент от S $\subseteq$ U се означава с една позиция на масива A[1..n]. Последователното търсене в такъв масив се осъществява за време O(n).

Търсенето може да бъде изпълнено много по-ефективно, когато се изпълнява върху подредено множество S, представено чрез сортиран масив A[1...n]. Предварително допускаме, че за търсения елемент x имаме $A[1] \leq x \leq A[n]$.

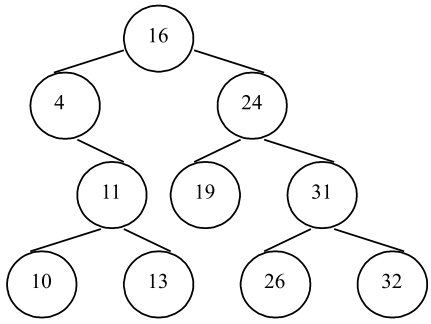
```
function Member (x:element; A:array[1..n] of element; n:integer):boolean;
begin
i:=1; j:=n;
while (j<>i+1) do begin k:=(i+j)div2; if A[k]<=x then i:=k else j:=k; end;
Member:=(A[i]=x);end.
```

Времето за двоичното търсене е от порядъка $O(\log n)$.

Реализацията на операциите по вмъкване и изтриване на елемент от масив е незадоволителна и въвеждаме нова структура от данни двоично дърво за търсене. Нека множеството S={16,4,11,13,10,24,31,26,32,19} представя следното двоично дърво:

Операцията, проверяваща дали елемента x∈U е в множеството S може да се реализира с рекурсивна функция.

```
function Member(x:element;v:node):boolean;
begin
if x=l(v) then Member:=true;
if x<l(v) then
      if v has left son w then
Member(x,w)
      else Member:=false;
if x>l(v) then
      if v has right son w then
Member(x,w)
      else Member:=false; end;
```



Всеки връх на дървото е означен с v и е определен със стойност l(v)∈S.

Ефективността на търсене в двоично дърво се влияе предимно от формата на дървото, т.е. как е балансирано. Балансирано по ширина дърво се дефинира по следния начин:

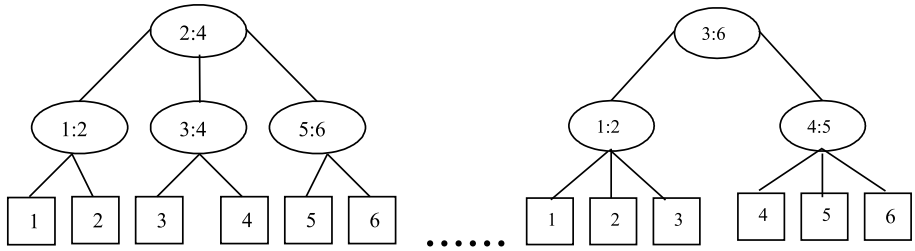
1. Нека T е двоично дърво. Ако T съдържа само един лист, тогава неговият корен е балансиран с $\rho(T)=1/2$. Иначе дефинираме $\rho(T) = (|T_L|+1)/(|T|+1)$, където $|T_L|$ и |T| означават броя на върховете в лявото поддърво T_L и на T съответно.

2. Двоичното дърво T е гранично балансирано със стойност α, ако имаме следните условия: а) $\alpha \leq \rho() \leq 1-\alpha$; б) T е лист или двете негови поддървета са гранично балансирани със стойност α. Тук сложността на търсенето е пропорционална на височината на дървото и е $O(\log |T|)$ [2].

Групата на балансираните по дължина дървета е по-голяма. Най-простият клас балансирани по дължина дървета са така наречените 2-3 дървета. Дървото T е 2-3 дърво, ако всеки връх на T има два или три наследника и всички листа са на едно ниво. Дефиницията гарантира, че 2-3дървото с височина h има най-малко 2^h и най-много 3^h листа. Подреденото множество S от n елемента е представено с 2-3 дърво T с n листа по следния начин: листата от T са означени с елементите от S по начина едно към едно в нарастващ ред. Всеки вътрешен връх означава две групи стойности L(v) и M(v). Тук L(v) означава най-голямата стойност на листата от лявото поддърво на

v, а M(v) означава аналогичната стойност от средното поддърво на v. Предполагаме, че десният наследник липсва.

Нека множеството S={1,2,3,4,5,6} е представено с две различни 2-3 дървета



Функцията за търсене е следната:

```
   function Member (x:element; v:node): boolean;
begin
if v is a leaf then Member:=(l(v)=x);
if v is not a leaf then begin if x<=L(v) then Member(x, left son of v)
   else if (L(v)<x) and (x<=M(v)) then Member(x,
   middle son of v) else begin
               if v has 3 sons then Member(x,
               right son of v);
               else Member:=false;
               end; end; end.
```

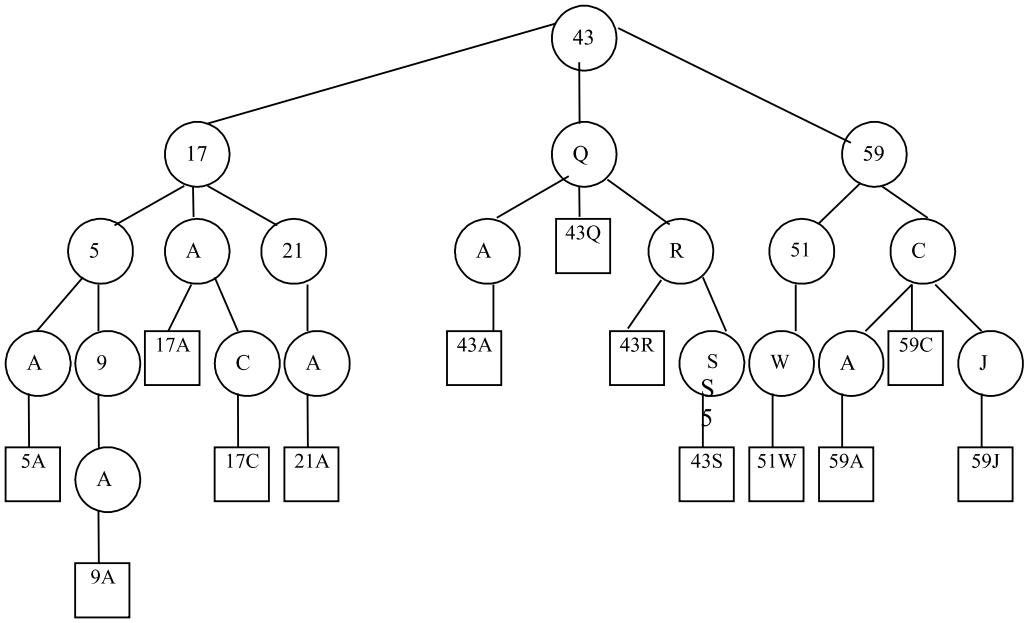
3. Алгоритми за търсене в многомерни множества. Търсенето в множества, чиито елементи са структурирани (съставени от няколко компоненти) се нарича многомерно търсене. Ще конструираме k-мерно подмножество V на Декартовото произведение $U = U_1 \times U_2 \times \cdots \times U_k$ от k множества, за k >= 1. Елементите на U са k-орки от вида $v = (c_1(v), c_2(v), \ldots, c_k(v))$, където $c_i(v)$ е i-та компонента на v i=1, 2, ..., k. В зависимост от интерпретациите на U можем да получим различни варианти на приложения. Например в базите от данни подмножествата на U съответстват на релации, индивидуалните k-орки са записи, елементите от k-орките са атрибути. В геометричната интерпретация индивидуалните k-орки съответстват на точки или вектори в к-мерното пространство. В друго приложение k-орката може да представя дума с дължина k, където i-тия компонент съответства на i-тия символ. В практиката размерността на U_i може да варира – дву-елементни булеви множества, ограничени множества (азбука), неограничени множества (естествените числа).

Ако k-орките във V са сортирани лексикографски, може да ги представим с масив A с размер k × n, където |U| = n и прилагаме последователно или двоично търсене. Двоичното търсене в този масив изисква O(log n) сравнения. Тъй като при всяко от тези сравнения участват k компоненти, общата сложност на търсене се измерва с O(klog n).

Множеството V={5A,9A,17A,17C,21A,43A,43Q,43R,43S,51W,59A,59J,59C} ⊆{ естествените числа} ×{ английската азбука} може да се представи чрез дървовидна

структура със следния алгоритъм:

```
procedure build (V:set;r:node;k:const;i:integer);
begin
if V≠∅ then C(i) is the set of all i-th components of k-tuples of V;
1. find the median m of C(i) and place it into the node r;
2. create the sets V1 = {v ∈ V|ci(v) < m}, V2 = {v ∈ V|ci(v) = m}, V3 =
{v ∈ V|ci(v) > m}
3. build(V1, left son of r, k, i)
4. if i<k then build(V2, middle son of r, k, i+1) else if i=k then cre-
ate a leaf, make it a middle son of r and place V2 in it; build(∅,r,k,k)
5. build(V3, right son of r, k, i)
```



Сравняваме първата компонента $c_1(x)$ на k-орката x със стойността в корена на дървото. Ако резултата от това сравнение е ("<")(">"), сравняваме същата компонета $c_1(x)$ със стойността в ляв (десен) наследник на r. Ако резултатът е "=", сравняваме следващата компонента $c_2(x)$ със стойността в средния наследник на r. Търсенето е успешно, ако всички компоненти на x са открити в дървото и съответният лист е достигнат.

4. Обобщаване на задачите за търсене в конкретна "задача за принадлежност". Проста "задача за принадлежност" е следната: за дадено подмножество B на множество A и за даден елемент a∈A, трябва да решим дали a∈B. Алтернативна формулировка на тази задача е следната: дадено е разделяне на множество A на два класа – множество B и неговото допълнение A\B и елемент a∈A, да се открие на кой от двата разделящи класа принадлежи a. Обобщаваме: за даден определен дял $\bar{A}$ от A и за даден елемент a∈A, трябва да открием такъв клас $\bar{a} \in \bar{A}$,

за който имаме $a \in \bar{a}$. В началото означихме $A_S = \{F(m,S) | m \in Q\}$ да е определено. Дефинираме функция $F_S : Q \to A_S$ за всяко $m \in Q$, като $F(m,S) = F_S(m)$. Създава се клас $\bar{Q} = \{\bar{m}_a : a \in A_S\}$, $\bar{m}_a = \{m \in Q : F_S(m) = a\}$. Дефинираме $\bar{F}_S : \bar{Q} \to A_S$ като определяме, че $\bar{F}_S(\bar{m}) = F_S(m)$.

Пример. Нека Q=U е множеството от всички цели числа, нека A е множеството от неотрицателните числа. Тогава за дадено множество $S = \{m_1 < m_2 < \cdots < m_n\} \subseteq U$ изображението, определено като F(m,S)=i, ако и само ако $m_i \leq m \leq m_{i+1}$, за i=0, 1, ..., n, където $m_0 = \inf Q$, $m_{n+1} = \sup Q$. Очевидно $A_S = \{0, 1, \ldots, n\}$ и $\bar{Q}$ се състои от класовете от вида $\bar{m}_i = \{m \in Q: m_i \leq m \leq m_{i+1}\}$, i=0, 1, ..., n. $\bar{F}_s$ е определено като $\bar{F}_s(\bar{m}_i) = i$ ако и само ако $m_i \leq m \leq m_{i+1}$. Съответният клас от "задачи за принадлежност" е представен с функция Φ: $Q \to \bar{Q}$, дефинирана като $\Phi(m) = \bar{m}_i$, ако и само ако $m_i \leq m \leq m_{i+1}$.

Тази "задача за принадлежност" може да се разреши с двоично търсене върху сортирана последователност $m_0, m_1, \ldots, m_{n+1}$.

ЛИТЕРАТУРА

[1] Т. М. Смит. Програмиране на PASCAL – приципи и методи, 1996.
[2] D. E. Knuth. The art of computer programming, 1973.
[3] C. F. Chabris. Artificial intelligence & turbo pascal, 1982.

Мария Славчева Палаханова
2700 Благоевград
ж.к. Еленово бл. 53, ап. 16
e-mail: palahanova@hotmail.com

LEARNING THE SEARCHING ALGORITHMS THROUGH SETS

Mariya S. Palahanova

The description of various searching algorithms over one- and multidimensional sets is given. Different methods for teaching of searching algorithms in freely choosing preparation of informatics are proposed. Some algorithms are described diagrammatically with the language programming PASCAL.