БЪЛГАРСКА АКАДЕМИЯ НА НАУКИТЕ ACADÉMIE BULGARE DES SCIENCES
ИЗВЕСТИЯ НА МАТЕМАТИЧЕСКИЯ ИНСТИТУТ
BULLETIN DE L'INSTITUT DE MATHÉMATIQUES
Том (Tome) XV

АЛГОРИТМИ ЗА ТЪРСЕНЕ И ЗАПИС В ТАБЛИЦИ НА ПОСЛЕДОВАТЕЛНО ПОСТЪПВАЩА ИНФОРМАЦИЯ

Атанас А. Раденски

В тази статия са дадени най-съществените резултати от [2], получени под научното ръководство на ст. н. с. Петър Бърнев.

1 ПОСТАНОВКА НА ЗАДАЧАТА

Нека H е линейно наредено множество от краен брой елементи Търси се такъв алгоритъм, който, ако $a_1, a_2, \ldots, a_N$ е произволна рецица от елементи на H, на a_1 да съпоставя число $t(a_1)$, на a_2 — число $t(a_2), \ldots$, на a_N — число $t(a_N)$, като са изпълнени и условията

1) $t(a_1)=1$;

2) $t(a_i)$ е естествено $(i=2, \ldots, N)$;

3) щом $a_m=a_n$, тогава и само тогава $t(a_m)=t(a_n)$;

4) за всяко $i=1, 2, \ldots, N$, ако p_i е кое да е естествено число, за което $1 \leq p_i \leq t(a_i)$, то има поне един номер $j\,(j \leq i)$, такъв, че на a_j е съпоставено точно числото p_i, т. е. $t(a_j)=p_i$;

5) средното време за работа на търсения алгоритъм е минимално.

Условията 1) — 4) означават, че за редица $a_1, \ldots, a_N$ търсеният алгоритъм трябва да изпълни следното: на a_1 съпоставя $t(a_1)=1$, ако $a_2=a_1$, то $t(a_2)=1$, иначе $t(a_2)=2$, ако например a_1, a_2 и a_3 са различни, то $t(a_3)=3$ и т. н. Вижда се, че ако до даден момент от работата си алгоритъмът е съпоставил k различни числа, те ще са точно числата $1, 2, \ldots, k$.

Тази задача се среща често при изработването на транслатори от някои алгоритмични езици. При превода на програмата, написана на такъв език, се иска на всяко срещнато име на проста променлива транслаторът да съпоставя машинен адрес на съдържанието на променливата. При това масивът от клетки, определен от адресите, съпоставени от транслатора на кой да е момент от работата му над дадена програма, трябва да бъде „плътен".

Нека допуснем, че е намерен алгоритъм, удовлетворяващ условията 1) — 4). Ако $a_1, \ldots, a_N$ е редица от елементи на H, алгоритъмът ще съпостави $t(a_1)=1$ на a_1. Когато алгоритъмът търси числото, което трябва да съпостави на a_i $(i=2, 3, \ldots, N)$, той трябва да провери дали няма такъв номер $j\,(1 \leq j < i)$, че $a_j=a_i$. Ако такъв номер има, то на a_i

се съпоставя $t(a_i)$ $t(a_j)$, за да бъде изпълнено условие 3). Ясно е, че трябва различните елементи от редицата, на които алгоритъмът вече е съпоставил числа (а също и самите числа), да бъдат записани по някакъв начин в таблица. И така всеки алгоритъм, който е решение на задачата, ще извършва „запис“ — построяване на таблица, в която ще извършва и „търсене“ — дали даден елемент е записан в таблицата или не.

2. НЯКОИ ИЗВЕСТНИ АЛГОРИТМИ

В зависимост от начина на построяване и търсене в таблиците, които използуват, съществуват различни алгоритми за решаване на поставената в т. 1 задача.

Често авторите, които описват такива алгоритми, посочват като тяхна характеристика само средния брой сравнения при търсене в таблицата. Но средната скорост на алгоритъма съществено зависи и от времето, необходимо за построяване на таблицата. За да се разбере дали една таблица (един начин на запис на информацията) осигурява икономия на време в сравнение с друга таблица, е необходимо да се пресметнат средните скорости на алгоритмите, използуващи едната и другата таблица, като се вземе пред вид както цялото време за търсене в таблицата, така и времето за построяването им.

По-нататък под „редица F“ ще се разбира редицата $a_1, \ldots, a_N$ от елементи на H, измежду които има n различни, $1 \leq n \leq N$.

Оценки на средните скорости бяха направени за основните известни алгоритми, решаващи поставената в т. 1 задача.

2.1. Алгоритъм, при който търсенето се извършва чрез последователни проверки

Една проста таблица, която може да се използува при решаването на задачата, е списък на различните елементи, които се срещат в редицата F до даден момент, като в тази таблица те са в реда, в който са в редицата F. Търсенето в таблицата се извършва чрез последователни сравнения от началото към края ѝ.

Средният брой сравнения при търсене на един елемент в такава таблица, съдържаща n елемента, е голям: $a_1(n)\ \frac{n+1}{2}$. Затова пък построяването на таблицата е много просто. Ако при търсенето в таблицата едно сравняване с елемент, записан в нея, се осъществява за време t_1, а записът на един елемент в таблицата — за време t_2, то средното време за работа на алгоритъма над една редица F е $S^{(1)}_{ср}\ \frac{t_1}{2} N(n+1)+t_2 n$.

При реализация на алгоритъма на АСМ t_1 и t_2 са твърде малки, затова този алгоритъм е за предпочитане при малки N и n.

2.2. Алгоритъм, при който търсенето се извършва чрез разделяне на две равни части

При този начин за построяване на таблица различните елементи, срещнати до даден момент в редицата F, са записани така, че образуват растяща последователност. Търсенето на елемент от редицата F в такава таблица се извършва по известния алгоритъм за търсене в масив от наредени числа чрез разделяне на две части, описан в [1]. Преимущество на този алгоритъм е малкият среден брой $a_2(n)$ разделяния на две части при търсене измежду n наредени елемента: $\log_2 n+1 \leqq a_2(n) < \log_2 n+2$, но не трябва да се забравя за голямото преустройство на таблицата, което се налага при нов запис — един запис в таблица от n елемента предизвиква средно $n/2$ премествания на елементи от таблицата на тези места. При конкретна реализация се оказва, че времето за едно разделяне на две части е около 2 пъти по-голямо от времето за едно сравняване при търсене за алгоритъма, описан в т. 2.1.

Ако t_1 е времето за едно разделяне на две части, а t_2 — времето за прехвърляне на един елемент от таблицата на ново място, то определяща компонента на средното време за работа на алгоритъма $S_{\text{ср}}^{(2)}$ над редицата F е $t_1 N \log_2 n + t_2 n^2/4$. Използването на такава таблица не е за препоръчване поне по две причини: t_1 има сравнително големи стойности и $S_{\text{ср}}^{(2)}$ зависи от n^2.

2.3. Алгоритъм, използващ двоично дърво

За решаването на поставената в т. 1 задача може да се използува таблица, наречена двоично дърво (описано например в [1]). В [3] е показано, че средният брой сравнения при търсене в двоично дърво от n елемента е

$$a_3(n) = \frac{2(n+1)}{n} \sum_{i=1}^{n} \frac{1}{i} - 3.$$

За n големи $a_3(n) \approx 2\ln n + 2c - 3$, където $c = 0{,}5772\ldots$ е константата на Ойлер. Ако t_1 е времето за извършване на сравнение с елемента на двоичното дърво, а t_2 — времето, необходимо за един запис, то средното време на работа над цяла редица може да се приеме $S_{\text{ср}}^{(3)} = t_1 N a_3(n) + t_2 n$. Стойността на t_1 практически не е голяма, $S_{\text{ср}}^{(3)}$ расте като $N \ln n$, така че този алгоритъм работи бързо. От разгледаните досега алгоритми той е най-ефективен по отношение на бързината, защото съчетава бърз запис с бързо търсене.

3. АЛГОРИТМИ, ИЗПОЛЗУВАЩИ КЛЮЧОВИ ФУНКЦИИ

Ключова функция с f стойности се нарича оператор l, дефиниран в H и приемащ стойностите си с едни и същи или близки вероятности измежду $1, 2, \ldots, f$.

Алгоритмите, използуващи ключова функция, имат по принцип тази обща черта, че си служат с допълнителна таблица от f клетки (които могат да се означат с $M(1), M(2), \ldots, M(f)$. Нека е зададена конкретна редица $a_1, a_2, \ldots, a_w$. При търсенето на $t(a_i)$ ($i = 1, 2, \ldots, w$) първо се намира $p = l(a_i)$ ($1 \leqq p \leqq f$). Клетката $M(p)$ съдържа някаква информация (или указание къде се намира такава информация), по която може да се намери $t(a_i)$ съгласно условията на задачата от т. 1.

Ако броят f на стойностите на ключовата функция е много голям, необходима е и голяма допълнителна таблица, при това може да се случи така, че много от нейните клетки да остават неизползувани. От друга страна, при малки f е трудно да се осигури стойностите на ключовата функция да се приемат с близки вероятности.

3.1. Алгоритъм, използуващ f-списък

Разпространен начин за използуване на ключова функция е следният: Построява се таблица, която се състои от f подсписъка. В даден момент подсписъкът с номер k се състои от различните елементи, които са срещнати до този момент в редицата F при намирането на $l(a_i)$, $i=1, 2, \dots$, (взети в реда, в който са в редицата F) и за които ключовата функция има стойност k, $k=1, 2, \dots, f$ (такава таблица ще наричаме f-списък). Проверката, дали даден елемент a_i е записан в f-списъка, започва с намирането на $k=l(a_i)$ и продължава с търсене чрез последователни сравнения в подсписъка с номер k.

Комбинаторно беше намерен средният брой сравнения $a_4(n)$ при търсене на елемент в f-списък при следните предположения: броят на елементите на H е T, частното $s=T/f$ е цяло, ключовата функция, която се използува, приема стойност k точно за s елемента на H $(k=1, 2, \dots, f)$. В резултат се получи

$$a_4(n)=\frac{f(T-s)!\,s!}{2nT!}\sum_{\substack{l=1\\ l\leq n}}^{s}\binom{T-n}{s-l}\binom{n}{l}l(l+1).$$

В случая, когато $n\leq s$, се получи $\sum_{\substack{l=1\\ l\leq n}}^{s}\binom{T-n}{s-l}\binom{n}{l}l(l+1)=2n\binom{T-2}{s-1}$ $+n(n+1)\binom{T-2}{s-2}$, откъдето следва, че $a_4(n)=(2(T-s)+(n+1)(s-1))/(2(T-1))$. За големи стойности на T $a_4(n)\approx 1+(n-1)/2f$.

Ако с t_1 е означено времето за извършване на едно сравнение при търсене в f-списък, а с t_2 — времето за извършване на запис, след като при търсенето е достигнат краят на подсписъка, то средното време за работа над цялата редица F е $S'^{(4)}_{\text{ср}}=t_1Na_4(n)+t_2(n)$. Стойността на t_1 тук се оказва приблизително равна на стойностите на t_1 в изразите за $S^{(1)}_{\text{ср}}$ и $S^{(2)}_{\text{ср}}$.

Забележка. Разгледан бе един конкретен случай, когато H се състои от 419 403 645 петсимволни думи, всяка от които може да бъде записана в оперативната памет на АСМ „Минск-32“; $s=9\,320\,081$, $f=45$; предположено бе, че алгоритмите се реализират на АСМ „Минск-32“ и така бяха определени стойностите на t_1 и t_2. В този случай бяха пресметнати при $n=100$, $N=1000$ стойностите на $S^{(i)}_{p}$ $(i=1, 2, 3, 4)$. Получи се приблизително $S^{(4)}_{\text{ср}}=S^{(1)}_{\text{ср}}/22=S^{(2)}_{\text{ср}}/7=S^{(3)}_{\text{ср}}/4$.

За $n\approx 1000$ f-списъкът все още може да осигури по-голяма бързина, отколкото дава двоичното дърво, въпреки че $a_4(n)$ расте (с нарастването на n) като n, а $a_3(n)$ — като $\log n$. В практическите случаи на задачата от т. 1 алгоритъмът, използуващ f-списък, може да се окаже най-бързото и едновременно с това просто решение.

3.2. Алгоритъм, използуващ f-дърво

Тук се предлага таблица, наречена f-дърво, подобна на описаната в т. 3.1, с тази разлика, че подсписъците са двоични дървета. Тъй като двоичното дърво осигурява по-голяма бързина, отколкото таблицата, която е подсписък в f-списъка, може да се очаква, че алгоритъмът, използуващ f-дърво, е по-бърз от този, използуващ f-списък.

Беше пресметнат средният брой $a_5(n)$ на сравненията, които трябва да се извършват за намиране на елемент, записан в f-дърво от n елемента (при същите предложения, както при пресмятането на $a_4(n)$):

$$a_5(n)=\frac{f(T-s)!\,s!}{n\,T!}\sum_{l=1}^{n}\binom{n}{l}\binom{T-n}{s-l}\,l\,a_3(l).$$

За големи T и s този вид на $a_5(n)$ не е удобен за пресмятане поради големите биномни коефициенти и факториели. В случая $n<s$ беше извършена преработка на формулата за $a_5(n)$, като в крайна сметка се получи

$$a_5(n)=b\left[1+\sum_{l=1}^{n-1}\binom{n-1}{l}a_3(l+1)b_l+\dots+b_\cdot\right],$$

където

$$b=\left(1-\frac{s}{T}\right)\left(1-\frac{s}{T-1}\right)\left(1-\frac{s}{T-2}\right)\cdots\left(1-\frac{s}{T-n+2}\right)\frac{1}{1-\frac{n-1}{T}},$$

$$b_l=\frac{1}{\prod_{j=1}^{l}\frac{n-1}{s-l}},\qquad l=1,2,\dots,n-1.$$

Ако t_1 е времето, необходимо за едно сравнение и преминаване към следващия елемент от дървото — подсписък при търсене, t_2 — времето за запис след достигане края на подсписъка, може да се приеме за средното време за работа на алгоритъма над цяла редица F:

$$S^{(5)}_{\text{ср}}(n)=t_1 N a_5(n)+t_2 n.$$

За случая, споменат в забележката към т. 3.1, бяха пресметнати някои стойности на $S^{(5)}_{\text{ср}}$ и $S^{(4)}_{\text{ср}}$, като бе прието $N=5n$. Оказа се, че стойностите на $S^{(5)}_{\text{ср}}$ и $S^{(4)}_{\text{ср}}$ са приблизително равни, когато n е от порядъка на 100, което означава, че за такива стойности на n използуването на алгоритъма от т. 3.1 е по-уместно (той е по-прост). За големи n обаче $S^{(4)}_{\text{ср}}\gg S^{(5)}_{\text{ср}}$ (като се има пред вид какви са подсписъците на f-списъка и f-дървото). Когато n достига по-големи стойности, алгоритъмът, използуващ f-дърво, е най-бърз.

3.3. Уравновесено дърво

Нека Q е линейно наредено множество, състоящо се от нечетен брой различни елементи: $a_1<a_2<\dots<a_{2m+1}$. Нека k е $m+2\leq k\leq 2m+1$. Тогава дефинираме $Q_k=\{\bar{a}_1, \bar{a}_2, \dots, a_{m+1}, \dots, a_{2m+1}\}$, като

$\bar{a}_i = a_k - m + i - 1$ за $i = 1, 2, \ldots, m, m+1, \ldots, 3m-k+2$,

$\bar{a}_i = a_k - 3m + i - 2$ за $i = 3m-k+3, \ldots, 2m+1$.

Смятаме Q_k наредено, като $\bar{a}_i < \bar{a}_j$ тогава и само тогава, когато $i<j$. Множеството Q_k може да бъде наречено уравновесено относно a_k (или относно k), защото в него има m елемента с номера, по-малки от номера на $a_k = \bar{a}_{m+1}$, и m елемента с номера, по-големи от номера на a_k.

Аналогично се дефинира Q_k, когато $1 \leq k \leq m$; с равенството $Q_{m+1} = Q$ се дефинира Q_{m+1}.

Пример. Ако $Q = \{a_1, \ldots, a_9\}$, то

$$Q_8 = \{a_4, a_5, a_6, a_7, a_8, a_9, a_1, a_2, a_3\},$$

$$Q_2 = \{a_7, a_8, a_9, a_1, a_2, a_3, a_4, a_5, a_6\}.$$

Нека е зададена редица от n различни елемента на $Q: a_1^*, a_2^*, \ldots, a_n^*$. Членовете на тази редица могат да бъдат записани в уравновесено дърво по следния алгоритъм (той е и определение за уравновесено дърво):

1) записва се a_1^* (a_1^* се нарича корен на уравновесеното дърво); на p се дава стойност 2, на l — стойност 1; преминава се на 2);

2) ако a_l^* е елементът a_k от Q, построява се Q_k; преминава се на 3);

3) нека a_p^* е елементът $\bar{a}_s$ от Q_k; ако $s > m-1$, проверява се дали има елемент, записан надясно от a_l^* (т. е. дали a_l^* има така наречения десен наследник), ако $s < m+1$, проверява се дали има елемент, записан наляво от a_l^* (ляв наследник); ако десен (респективно ляв) наследник няма, записва се като такъв a_p^*, на l се дава стойност 1, на p се дава стойност $p+1$ и ако $p>k$, край; в противен случай се преминава на 2); ако десен (респективно ляв) наследник има и той е a_r^*, то на l се дава стойност r и се преминава на 2).

Описаният по-горе алгоритъм се опростява. Под лява (респективно дясна) половина на Q_k се разбира множеството от елементите на Q_k, които като елементи на Q_k имат номера, по-малки (респективно по-големи) от m.

Нека например a_2^* е записан като ляв наследник на a_1^* в процеса на построяване на уравновесено дърво от елементите на редицата $a_1^*, a_2^*, \ldots, a_n^*$. Показва се, че всички елементи и само те, които в лявата половина на Q_k са наляво от a_2^* (т. е. с номера в Q_k, по-малки от номера на a_2^* в Q_k), могат да бъдат леви наследници на a_2^* и аналогично всички елементи (и само те), които са надясно от a_2^*, но са в лявата половина на Q_k, могат да бъдат десни наследници на a_2^*. Следователно левият и десният наследник на a_2^* ще бъдат записани в уравновесеното дърво по правилата за построяване на двоично дърво, като се използува наредбата на лявата половина на Q_k. Това аналогично е в сила за двата наследника на a_2^*, за техните наследници (ако има такива) и т. н. Следователно няма да има нужда от построяване на всички уравновесени относно елемент на редицата $a_1^*, a_2^*, \ldots, a_n^*$ множества, а необходимо ще бъде само Q_k (тъй като a_1^* е a_k от Q). Тъй като съвкупността от всички уравновесени дървета не се изменя, ако се използува не наредбата на Q_k, а произволна наредба, то при определянето на наследниците на левия и десния наследник на a_1^* може да се използува наредбата, дадена в Q_k. Следователно построяването на уравновесено дърво от елементите на редицата $a_1^*, a_2^*, \ldots, a_n^*$ е равносилно с построяването на 2-дърво за редицата a_1^*, $a_2^*, \ldots, a_n^*$ чрез специална ключова функция φ_k, зависеща от a_1. Като се има пред вид, че a_1^* е a_k от Q, φ_k се дефинира така: ако a_i от $Q(i=1, 2, \ldots, k-1, k+1, \ldots, 2m+1)$ е a_j от Q_k, то $\varphi_k(a_i)=1$, щом $j<m+1$, и $\varphi_k(a_i)=2$, щом $j>m+1$. Тогава средният брой сравнения $a_6(n)$ за търсене на елемент, записан в уравновесено дърво, е $a_6(n)-a_5(n-1)+1$, като във формулата за $a_5(n-1)$ е заместено $f=2$.

Уравновесеното дърво може да се използува аналогично на двоичното дърво за решаване на задачата от т. 1.

По-съществена разлика между скоростите на алгоритъма, използуващ уравновесено дърво, и алгоритъма, използуващ двоично дърво, се получава при по-големи стойности на n.

Аналогично на таблиците, описани в т. 3.1 и 3.2, може да се използува таблица, на която подсписъците са уравновесени дървета.

4. ЕДИН ПРАКТИЧЕСКИ СЛУЧАЙ НА ЗАДАЧАТА ОТ Т. 1 И ДВЕ ОСОБЕНОСТИ НА ЕЗИКА

При изработването на транслатор от FORTRAN за АСМ Минск-32 може да бъде поставена следната задача: да се намери алгоритъм, който да съпоставя на всяка проста променлива, срещната от транслатора при прегледа на програмата, машинен адрес на съдържанието ѝ (ако една проста променлива се среща в програмата няколко пъти, на нея всеки път трябва да бъде съпоставен един и същ адрес). Съпоставените на простите променливи адреси трябва да определят плътен масив от клетки на оперативната памет. Търсеният алгоритъм трябва да бъде максимално бърз и икономично да използува оперативната памет.

Алгоритмите, посочени в т. 2 и 3, са решения на горната задача. Но работата и на най-бързите от тях може да бъде ускорена значително, ако се използуват две особености на програмите на FORTRAN. Тези особености бяха установени с изследване на 34 програми на FORTRAN, случайно избрани от издания на Канзаския университет. Може да се очаква, че подобни резултати ще бъдат получени и за други, близки до FORTRAN алгоритмични езици.

4.1. Първа особеност

Оказа се, че в програмите на FORTRAN се срещат много еднобуквени имена на прости променливи. В изследваните 34 програми прости променливи са употребени общо 7900 пъти, като еднобуквени имена на прости променливи — 3513 пъти, т е. в 44,45% от случаите. Тогава независимо от алгоритъма, по който се съпоставят адреси на съдържанието на простите променливи, може на всички променливи с еднобуквени имена предварително да бъдат съпоставени адреси на съдържанието, които да се получават направо от кодовете на тези имена. Основният алгоритъм, който съпоставя адреси на простите променливи, ще работи по-малко време, тъй като в около 44% от случаите той ще бъде освободен от задължението да съпоставя адреси на съдържанието — това са случаите, в които трябва да бъде съпоставен адрес на променлива с еднобуквено име.

4.2. Втора особеност

Нека с M_k е означен броят на случаите, в които транслаторът, след като е срещнал име на проста променлива в някоя от изследваните програми и е съпоставил адрес на съдържанието ѝ, ще срещне същото име в същата програма най-много, след като съпостави адреси на още k на брой не непременно различни прости променливи, $k=0, 1, 2, \ldots, 7$. С M_γ е означено число, което показва колко пъти в изследваните 34 програми се срещат имена на прости променливи, т. е. колко пъти при транслацията на изследваните програми транслаторът ще трябва да съпоставя адрес на съдържанието на проста променлива. Резултатите от изследването са:

	k	0	1	2	3	4	5	6	7
	M_k	1330	2367	3123	3635	4051	4331	4550	4750
В %:	$100\,M_k/M_\gamma$	16,8	30	39,2	46	51,3	54,8	57,8	60,1.

Оттук се вижда, че в 39,2% от случаите име на проста променлива, срещнато в някоя от изследваните програми, се е срещнало (в същата програма) най-много след две имена на прости променливи.

Тази особеност може да бъде използувана по следния начин: Независимо от основния алгоритъм, избран за съпоставяне адреси на съдържанието на простите променливи, организира се динамична таблица, в която във всеки момент се записват имената на последните k прости променливи, на които са съпоставени адреси, както и самите адреси (например $k=3$). Ако в следващия момент от транслацията бъде срещната проста променлива, първо се проверява дали името ѝ е записано в динамичната таблица и ако е записано, от таблицата се прочита и съпоставеният ѝ адрес. От получените статистически данни се вижда, че за изследваните програми при $k=4$ в 46% от случаите търсенето в динамичната таблица би завършило с успех. Ако търсенето е завършило с неуспех, на променливата се съпоставя адрес по основния алгоритъм (който може да е някой от описаните в т. 2 и 3). Конкретната стойност на k се определя след оценка на скоростта на основния алгоритъм, така че да се получи най-голяма икономия на време.

ЛИТЕРАТУРА

1. Лавров, С. С., Л. И. Гончарова. Автоматическая обработка данных. Хранение информации в памяти ЭВМ. М., 1971.
2. Радenски, А. А. Алгоритми за търсене и запис на информация в таблици (Дипломна работа — Мат. фак. СУ, 1972.)
3. Hibbard, Th. N. Some combinatorial properties of certain trees with application to searching and sorting. — J. Assoc. Comput. Machinery, 9, 1962, No. 1, 13—28.

Постъпила на 30. IX. 1972 г.

АЛГОРИТМЫ ПОИСКА И ЗАНЕСЕНИЯ В ТАБЛИЦЫ ПОСЛЕДОВАТЕЛЬНО ПОСТУПАЮЩЕЙ ИНФОРМАЦИИ

Атанас Радeнски

(Резюме)

В статье рассматривается задача о построении алгоритмов, осуществляющих поиск и занесение в таблицы последовательно поступающей информации.

Оцениваются средние скорости четырех известных алгоритмов (в т. 3.1 указывается среднее число сравнений при поиске в таблице, состоящейся из f подсписков, названной автором f-списком). Предлагаются в т. 3.2 и 3.3 два новых способа занесения информации в таблицы f-дерево и сбалансированное дерево, и оцениваются скорости поиска в f-дереве и сбалансированном дереве. В т. 4 рассматривается частный случай задачи из т. 1, возникающий при разработке транслятора с ФОРТРАН-а, в связи с которой указаны две особенности этого языка.

ALGORITHMS FOR SEARCHING AND TABULATION OF SUCCESSIVE IN-COMING INFORMATION

Atanas Radenski

(Summary)

In the paper the problem of finding algorithms performing the searching and tabulation of successive in-coming information is considered.

In 2. and 3.1. estimates of the average speed of four known algorithms are given (in 3.1. is given the average number of comparisons for searching in a table, consisting of sublists and called the f-list). In 3.2. and 3.3. two new methods of organization of the information into tables are proposed: f-tree and balanced tree, the searching speed in them being given. In 4. a particular case of the problem from 1. arising at the working out of FORTRAN's translator is considered and, in connection with the problem under consideration, two singularities of this language are pointed out.