NĚKTERÉ METODY VYHLEDÁVÁNÍ V POLÍCH (v alg. Fibonacciovské dusledy ...)

Doc.Ing.Jan Honzík, VSC. - katedra počítačů FE VUT v Brně

Za primární vlastnost počítače se velmi často považuje schopnost uchovat velký objem informace a posléze umožnit přístup ke složkám této informace. Na tuto vlastnost se zaměřuje část disciplíny programování nazvaná tradičně "vyhledávání" (ang. searching), nebo výstižněji "uchování informace a její zpřístupňování" (ang. Storage and retreival of information). Z této oblasti čerpá své náměty předkládaný příspěvek, který uvedením několika základních metod vyhledávání v polích chce reagovat na diskusi předchozího semináře, v níž zazněly hlasy po rekapitulaci některých obecně používaných algoritmů.

1. Abstraktní datové struktury, specifikace abstraktních typů dat, abstraktní typ dat "tabulka"

Abstraktní řídicí struktury (sekvence, alternativa, iterace) se objevily již v jazyce Algol 60 a umožnily výraznější odpoutání algoritmizace od strojové úrovně. V oblasti datových struktur byla vazba na stroj silnější. Datové struktury, které se nedaly přímo a snadno representovat v paměti se pro obavy z nízké účinnosti (malá rychlost, velká spotřeba paměti) příliš nepoužívaly. Výsledky výzkumu výpočetní složitosti algoritmů ukázaly, že použití nových, zdánlivě složitých datových struktur vede mnohdy ke kvalitativně účinnějším algoritmům. Od datového typu reprezentujícího množinu datových objektů a množinu operací nad těmito objekty, se zdůrazněním toho co data reprezentují a co s nimi operace provádějí a potlačením hlediska jak jsou data zobrazena a jak se nad nimi operace provádějí dospívá k abstraktním datovým strukturám (ADS) a k abstraktnímu typu dat (ATD), jako představiteli určité třídy ADS. Odsunutí implementační stránky ADS do fáze rozkladu a možnost uzavření její implementace do samostatného programového celku (procedury, modulu) významně zjednodušuje algoritmus pracující s ADS. Systémy podporující použití ATD dovolují práci s ATD výhradně prostřednictvím stanovených abstraktních operací a brání vědomému i náhodnému přímému přístupu ke všem vnitřním objektům implementace ATD.

Specifikace ATD specifikuje objekty vytvářející ATD a operace, které lze nad specifikovaným typem provádět. Tzv. operační specifikace mají tvar algoritmů ve vhodném jazyce, které implicitně popisují vlastnosti operací. Výhodou je blízkost praktickým programátorům,

nevýhodou je explicitní zavedení jisté implementace, která nemusí být vždy nejvýhodnější. Algebraická specifikace sestává ze syntaktické definice a sématické specifikace.

S vyhledáváním je neoddělitelně spojen ATD nazývaný "vyhledávací tabulka" (angl. look-up table), nebo zkráceně jen tabulka. ATD tabulka je homogenní, obecně dynamická datová struktura, jejíž operace jsou nejčastěji odvozeny od mechanizmu, kterému v praxi říkáme "kartotéka". Každá položka tabulky (kartotéky) obsahuje složku, která plní funkci klíče. Hodnotou klíče lze jednoznačně identifikovat každou položku; zásadní vlastností tabulky je, že obsahuje položky, hodnoty, jejichž klíčů jsou navzájem různé. S problematikou vyhledávání úzce souvisí problematika řazení (třídění) a obě oblasti se často prolínají. Lze říci, že prvotním cílem řazení je usnadnění vyhledávání.

Pro ATD tabulka jsou v tab.1 - uvedeny charakteristické operace

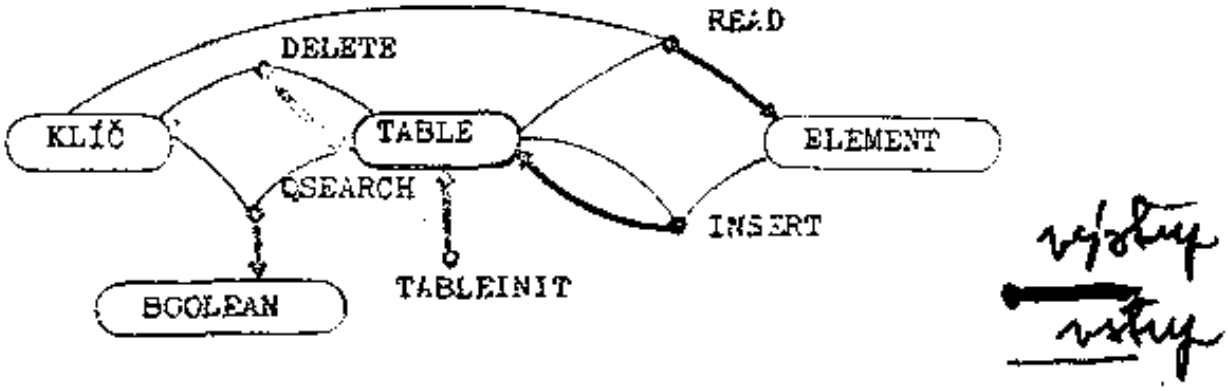
Název operace	stručný význam operace
TABLEINIT	Vytvoř prázdnou tabulku
INSERT	Vlož novou položku s daným klíčem do tabulky; je-li v tabulce položka s tímto klíčem, nahraď ji (přepiš) novou položkou
READ	Získej hodnotu položky, jejíž klíč se rovná zadané hodnotě. Není-li taková položka v tabulce, dochází k chybovému stavu.
DELETE	Vyřaď z tabulky položku s danou hodnotou klíče. Není-li taková položka v tabulce - prázdná operace.
ISSEARCH	Predikát udávající, zda tabulka obsahuje položku, jejíž klíč se rovná zadané hodnotě.

Tab. 1 Operace ATD tabulka

Pro vyjádření syntaktické specifikace se již vžila grafická forma diagramu signatury. Diagram signatury ATD tabulka je uveden na obr.1. Diagram sestává z pojmenovaných typů dat vyjádřených ovály. Tučný ovál představuje specifikovaný datový typ. Pojmenované operace jsou představovány vyplněnými kroužky a argumenty spojnicemi, které spojují kroužek operace s ovály typů. Tenké spojnice představují vstupní a tučné výstupní argumenty. Operace typu "generátor" /tzv.nulární operace) nemá vstupní argument a používá se nejčastěji k inicializaci

datového typu - na obr. 1 je to operace TABLEINIT.

Sémantická specifikace nejčastěji sestává ze systému axiómů. Jeden z možných systémů axiómů pro ATD tabulka je uveden v tab.2.



Obr.1 Signatura ATD tabulka

1.	QSEARCH(KLIC,INITTABLE) = false
2.	QSEARCH(KLIC,DELETE(KLIC,TABLE)) = false
3.	QSEARCH(KLIC,INSERT(KLIC,ELEMENT,TABLE)) = true
4.	READ(KLIC,INITTABLE) = error
5.	READ(KLIC,DELETE(KLIC,TABLE)) = error
6.	READ(KLIC,INSERT(KLIC,ELEMENT,TABLE)) = ELEMENT
7.	READ(KLIC,INSERT(KLIC,$ELEMENT_1$,INSERT(KLIC,$ELEMENT_2$,TABLE))) = $ELEMENT_1$
8.	DELETE(KLIC,TABLEINIT) = TABLEINIT
9.	DELETE(KLIC,INSERT(KLIC,ELEMENT,TABLEINIT)) = TABLEINIT
10.	DELETE(KLIC,INSERT(KLIC,ELEMENT,TABLE)) = TABLE
11.	DELETE(KLIC,TABLE) = if QSEARCH(KLIC,TABLE) then DELETE(KLIC,TABLE) else TABLE

tab.2 Systém axiómů sématické specifikace ATD tabulka

2. Vlastnosti ATD tabulka a metod její implementace

O tabulce říkáme, že je statická, nemění-li se v průběhu zpracování počet jejích položek. Příkladem takové tabulky je telefonní seznam,

kódovací tabulka, číselník ap. Mění-li se při zpracování počet položek, je tabulka dynamická. Dynamika tabulky má jednodušší formu, je-li povoleno pouze vkládání nových položek do tabulky a složitější formu, je-li povoleno také rušení položek tabulky.

Klasifikaci implementačních metod ATD tabulka lze provést na základě různých hledisek. Podle přístupu k prvkům tabulky rozlišujeme metody na sekvenční a na metody s náhodným přístupem. Metody s náhodným přístupem můžeme dále rozdělit podle toho, zda jsou prvky uspořádány lineárně (v poli), nebo nelineárně (např. ve stromové struktuře). Indexsekvenční přístup se využívá v tabulkách s rozptýlenými položkami. Metody se mohou lišit také podle toho, zda je nad klíči položek definována pouze relace ekvivalence, nebo také relace uspořádání. Jiným hlediskem může být skutečnost, zda metoda pracuje s původním klíčem, nebo s transformovaným klíčem. Druhá skupina vede k tabulkám s rozptýlenými položkami (Hash-table). Existují metody, které vyhledávaní realizují na základě částečné shody vyhledávaného klíče s klíčem položky v tabulce. Tyto metody jsou aktuální v systémech, v nichž je klíčem např. jméno osoby, které mohlo vzniknout na základě nesprávně interpretované výslovnosti. Tyto metody mohou v případě neúspěšného vyhledání vyhledat položku, jejíž klíč se částečně shoduje s vyhledávaným klíčem a s jistou pravděpodobností mohl vzniknout chybným zápisem při pořizování dat.

V dalších odstavcích bude uvedeno několik metod, založených na náhodném přístupu. Jádrem každé metody je implementace operace QSEARCH. U každé metody bude uvedena diskuse dynamických vlastností.

3. Sekvenční vyhledávání v poli

Nechť je dán pascalovský typ položky tabulky

```
TYPPOLOZKY = record
               DATA : TYPDATA;
               KLIC : TYPKLIC
             end
```

kde nad typem TYPKLIC je definována relace ekvivalence, a dále

```
TYPPOLE = array [1..MAX] of TYPPOLOZKY
```

kde MAX je konstantní horní mez indexu

Nechť je deklarováno:

```
var  POLE : TYPPOLE;    {Pole implementující tabulku}
     K : TYPKLIC;       {Hodnota vyhledávaného klíče}
```

Pak prosté sekvenční vyhledávání v poli vyjadřuje algoritmus:

```
i:=Ø;  {var  i:Ø..MAX}
Repeat
   i:=succ(i)          {nebo  i:=i+1}
until  (i=n) or(POLE [i] .KLIC=K);   {POLE [n] je poslední
                                      aktivní prvek}
QSEARCH:=POLE  i  .KLIC=K;    {var  QSEARCH:Boolean}
{if  QSEARCH  then  prvek  POLE i   je hledaným prvkem}
```

V příspěvku /2/ sborníku Programování '83 byl uveden důkaz algoritmu, který je jednoduchým zlepšením uvedeného algoritmu. Jeho rychlost se zvýší zavedením tzv. zarážky, pro kterou je zapotřebí prostor jedné položky pole. Zarážka se vkládá za poslední aktuální položku pole. Algoritmus má tvar:

```
POLE [n+1] .KLIC:=K,      {vložení zarážky}
i:=1;
while   POLE i  .  KLIC≠K  do   i:=succ(i);
QSEARCH:= i≠(n+1);
{if  QSEARCH  then  prvek  POLE i    je hledaným prvkem}
```

Jestliže pole obsahuje n položek, pak pro uvedené algoritmy platí:

c	....	konstanta vyjadřující délku jednoho průchodu pole
T_f	....	čas při neúspěšném vyhledání
$T_{t,i}$	....	čas při úspěšném vyhledání i-té položky pole
$ØT_t$	....	průměrný čas pro úspěšné vyhledání při stejné pravděpodobnosti

$$T_f \simeq c.n \qquad T_{t,i} \simeq c.i \qquad ØT_t \simeq c.(n+1)/2$$

Z toho vyplývá, že nejrychleji budou úspěšně vyhledány ty položky, které jsou zařazeny na začátek pole. Je-li známa pravděpodobnost vyhledávání jednotlivých položek, je účelné seřadit je podle této pravděpodobnosti sestupně. Na této myšlence jsou založeny "sebeorganizující" algoritmy, které na základě informace o četnosti vyhledávání jednotlivých položek, získané v průběhu zpracování, umisťují často vyhledávané položky do

čela pole v naději, že budou i v budoucnu vyhledávány častěji, a že úspora času při vyhledávání bude větší, než čas potřebný pro reorganizaci.

Vyhradíme-li pro uvedené metody dostatečně velké pole, pak konec cyklu neúspěšného vyhledávání, resp. pozice zarážky je za posledním aktivním prvkem pole. Při vkládání nové položky (INSERT) se tato pozice zvýší o 1 a nový prvek bude posledním aktivním prvkem pole.

Poněkud složitější je rušení položky (operace DELETE). V zásadě lze volit mezi dvěma přístupy:

a) Ruší-li se i-tý prvek (i≠n), posune se část pole od (i+1) prvku do posledního aktivního prvku (n) o jednu pozici vlevo tedy

   for j:=i+1 to n do POLE [j-1]: =POLE [j]

b) klíč rušeného prvku se nastaví na hodnotu, o které je jisto, že nebude nikdy vyhledávána (prvek se "zaslepí").

Nevýhodou prvního přístupu je potenciálně významná časová náročnost posuvu. Nevýhodou druhého přístupu je postupné zužování použitelného prostoru zaslepováním. Lze sice po vyčerpání prostoru "regenerovat" prostor vyhledáváním volných míst mezi "slepými" položkami, zůstává ale fakt, že se při vyhledávání prohledávají zbytečně i "slepé" položky.

4. Sekvenční vyhledávání v seřazeném poli

Je-li nad typem TYPKLIC definována relace uspořádání, lze pole seřadit (např. vzestupně). Sekvenční vyhledávání se zrychlí v případě neúspěšného vyhledávání, protože cyklus lze skončit, je-li klíč testované položky větší než vyhledávaný klíč. Takový algoritmus má tvar:

```
i:=0;
repeat
   i:=succ(i)
until (i=n) or (POLE [i]. KLIC ≥ K);
QSEARCH:=POLE [i]. KLIC=K
{if QSEARCH then prvek POLE [i] je hledaným prvkem}
```

I tento algoritmus lze urychlit zavedením zarážky :

```
i:=Ø;
POLE [n+1] .KLIC:=MAXKLIC;  {MAXKLIC je větší než hodnoty všech
                             vyhledávaných klíčů}
repeat
   i:=succ(i)
until POLE[i] . KLIC≥K;
QSEARCH:=POLE [i] .KLIC=K;  atd.
```

Na rozdíl od neseřazeného pole, je při vkládání do seřazeného pole nutno zachovat seřazenost, s čímž je spojen posun masivu pole od i-tého indexu do pozice posledního aktivního prvku o jedno místo doprava. Jestliže operace INSERT následuje po operaci SEARCH, končí uvedené algoritmy nalezením indexu, na který se po posuvu vloží nový prvek.

Operaci DELETE lze realizovat posunem masivu pole od vyřazovaného prvku do konce aktivní části pole o jednu pozici vlevo. Metoda "zaslepování" lze použít s klíčem, který je menší než všechny možné vyhledávané klíče. Regenerace zaslepených míst však opět nemusí být časově výhodná.

Dynamické vlastnosti sekvenčních algoritmů se významně zlepší, implementuje-li se tabulka <u>zřetězeným seznamem</u> (jednosměrným). Princip algoritmu vyhledávání zůstane zachován, operace INSERT a DELETE jsou však podstatně časově účinnější, za cenu většího paměťového prostoru, potřebného pro konstrukci zřetězeného seznamu.

(5.) Binární vyhledávání

Binární vyhledávání patří do třídy nesekvenčního vyhledávání v seřazené lineární struktuře s náhodným přístupem k jednotlivým prvkům. Metody z této skupiny se podobají numerickým metodám pro hledání kořene funkce jedné proměnné, známe-li interval, v němž je právě jeden kořen.

Nechť pro prvky pole POLE platí:

$$POLE[1].KLIC \leq POLE[2].KLIC \leq \ldots \leq POLE[n].KLIC$$

a nechť dále platí tvrzení

$$(K \geq POLE[1].KLIC) \text{ and } (K \leq POLE[n].KLIC)$$

Pak lze princip vyhledávání stručně slovně popsat takto: Hledaný klíč se porovná s klíčem položky v polovině prohledávaného intervalu pole. Dojde-li ke shodě, končí vyhledávání úspěšně. Je-li hledaný klíč menší, opakuje se proces pro levou a je-li větší pro pravou polovinu intervalu pole. Vyhledávání končí neúspěchem v případě, že prohledávaná část pole je prázdná (tzn., že levý index je větší než pravý). Algoritmus má tvar:

```
el:=1;*  {var  el:1..MAX;  levá hranice intervalu pole}
er:=n;   {var  er:1..MAX;  pravá hranice intervalu pole}
repeat
   i:=(el+er)div 2;    {var  i:1..MAX; index poloviny pole}
   if K POLE i .KLIC
      then  er:=i-1    {hledaný klíč může být v levé polovině}
      else  el:=i+1    {hledaný klíč může být v pravé polovině}
until  (K=POLE[i] .KLIC)or(er<el);
QSEARCH  :=K=POLE[i] .KLIC;
{if  QSEARCH  then  prvek  POLE[i]  je hledaným prvkem}
```

Dijkstrova varianta binárního vyhledávání vychází z předpokladu, že pole může obsahovat více položek se shodným klíčem. Tento předpoklad je sice v rozporu s vlastnostmi ATD tabulka, algoritmus však může být užitečný pro řadu aplikací. V případě úspěšného vyhledání se nalezne nejlevější ze skupiny položek se shodnými klíči. (Algoritmus pochopitelně pracuje i v poli se všemi různými klíči. Úspěšné i neúspěšné vyhledání je však stejně dlouhé, což je nevýhoda oproti předchozímu algoritmu).

*) Znak "1" představuje celočíselnou hodnotu "jedna", místo "ℓ" je "el".

Nechť platí:

$$POLE[1].KLIC \leq POLE[2].KLIC \leq \ldots \leq POLE[n-1].KLIC < POLE[n].KLIC$$

a také tvrzení

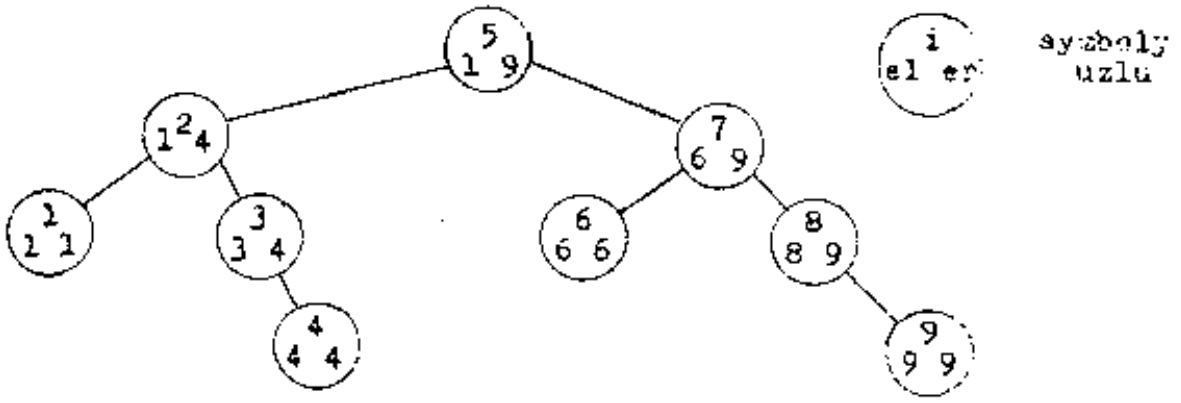
$$(POLE[1].KLIC \leq K) \text{ and } (K < POLE[n].KLIC)$$

Pak má algoritmus Dijkstrovy varianty tvar:

```
el:=1;
er:=n;
while er  ≠  (el + 1)  do
  begin
     i:=(el+er)div  2;
```

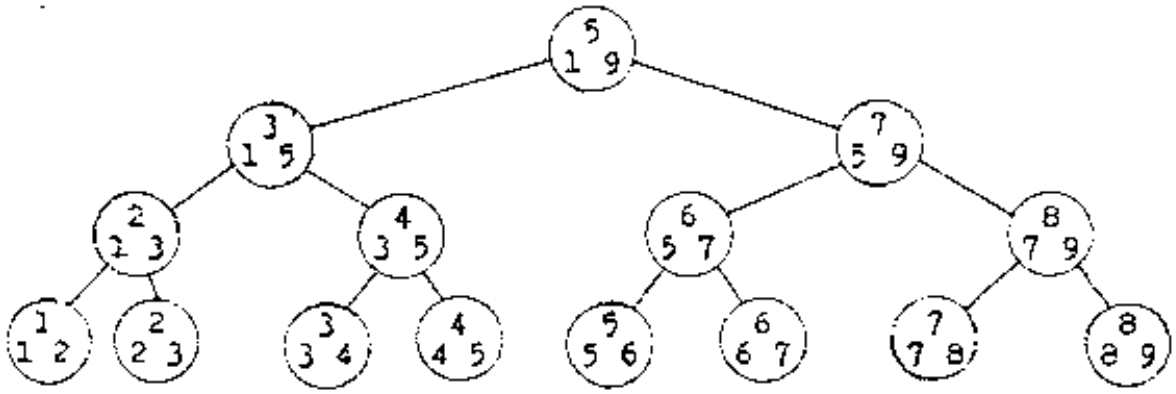
```
        if  POLE[i].KLIC < K
            then  el:=i
            else  er:=i
    end;
    QSEARCH:=K=POLE[i].KLIC;
{if QSEARCH then prvek POLE[i] je nejlevějším prvkem skupiny
     položek se shodným klíčem}
```

Zatímco u prvního algoritmu může trvat úspěšné vyhledávání kratší dobu než neúspěšné, Dijkstrova varianta má úspěšné i neúspěšné vyhledání stejně dlouhé. Podrobnější rozbor binárního vyhledání usnadňuje jeho reprezentace binárním rozhodovacím stromem. Obě varianty jsou pro 9-ti prvkové pole uvedeny na obr. 2 a 3.



Obr.2 Stromová reprezentace binárního vyhledávání

Z uvedené reprezentace vyplývá, že pro binární vyhledávání v seřazeném poli o n aktivních prvcích (kde $n \geq 2^{k-1}$ a $n < 2^k$) je pro úspěšné vyhledání zapotřebí podle prvního algoritmu minimálně jeden a maximálně k porovnání a pro neúspěšné vyhledání minimálně k-1 a maximálně k porovnání. Pro Dijkstrovu variantu je počet porovnání vždy k.



Obr.3 Stromová reprezentace Dijkstrovy varianty

(6) Uniformní binární vyhledávání

Místo tří proměnných (i,el,er) lze použít jen dvou: aktuální index i a odchylka m od aktuálního indexu i. Po každém neúspěšném porovnání ustavíme nové hodnoty

$$i := \pm \lceil m/2 \rceil \quad a \quad m := \lfloor m/2 \rfloor$$

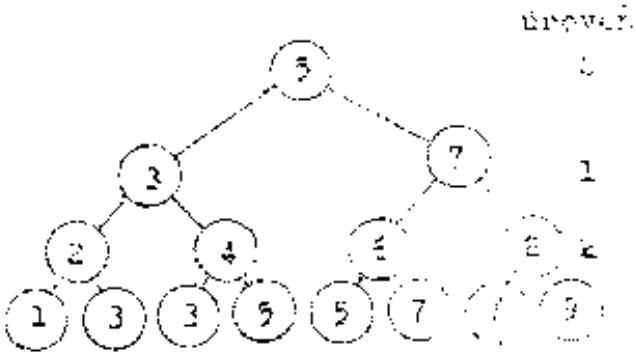
Je-li n sudé, pak algoritmus potřebuje pomocnou položku pole s indexem ∅ a hodnotou klíče menší, než všechny vyhledávané klíče. Algoritmus uniformního binárního vyhledávání má tvar:

```
POLE[∅].KLIC:=MINKLIC;   {Nutné jen je-li n sudé}
i:=(n+1) div 2;          {i := ⌈n/2⌉}
m:=n div 2;              {m := ⌊n/2⌋}
While (m≠∅) and (K ≠ POLE[i].KLIC) do
   Begin
      if K < POLE[i].KLIC
         then i := i - (m+1)div 2
         else i := i + (m+1)div 2;
      m:=m div 2
   end
QSEARCH :=K=POLE[i].KLIC ;
```

Rozhodovací strom pro tuto metodu je uveden na obr. 4 a 5 pro n = 9 a pro n = 10.

Metoda se jmenuje uniformní proto, že ať se algoritmus přesune ve stromu vlevo či vpravo, je přírůstek m vždy stejný. Výrazná přednost této metody se využije, je-li tabulka statická. V tom případě lze pro danou tabulku předem vytvořit pole odchylek, čímž se odstraní operace dělení a algoritmus se značně zrychlí (podle /1/ až dvojnásobně).
Pro tabulku odchylek DELTA platí :

$$DELTA[j] = round(n/2^j) \quad pro \quad 1 \le j \le (\lfloor \ln_2 n \rfloor + 2)$$



$n=9$, $i_0=5$, $m_0=4$
$m_0=4$, $i_1=i_0\pm2$
$m_1=2$, $i_2=i_1\pm1$
$m_2=3$, $i_3=i_2\pm1$
$m_3=0$

Obr.4 Rozhodovací strom pro n=9

$n=10$, $i_0=5$, $m_0=3$
$m_0=5$, $i_1=i_0\pm3$
$m_1=2$, $i_2=i_1\pm1$
$m_2=1$, $i_3=i_2\pm1$
$m_3=$

Obr. 5 Rozhodovací strom pro n=10

Pak lze algoritmus zapsat takto:

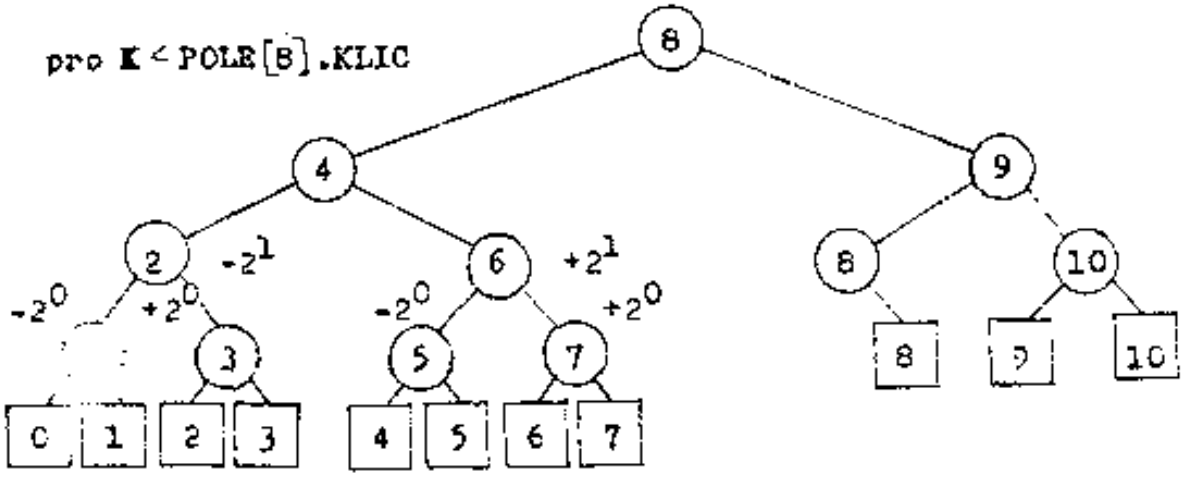
```
MOCNINA  := 2;  {var  MOCNINA:1..MAXINT}
for  j:=1  to  ka+2   {kde pro  ka  platí  2^(ka-1) < n ≤ 2^ka}
begin
   DELTA [j]:=  round(n/MOCNINA);
   MOCNINA  :=  MOCNINA * 2
end
i:=DELTA [1] ;
j:=2;
while (K≠POLE [i] .KLIC)  and  (DELTA [j] ≠0  do
   begin  if  K  POLE  i  .KLIC
                then  i:=i+DELTA [j]
                else  i:=i-DELTA [j]  ;
              j:=succ(j)
   end
   QSEARCH:=K=POLE [i]  .KLIC;
```

Užitečnou variantou uniformního binárního vyhledávání je Sherova metoda, která po prvním rozdělení transformuje pole do tvaru, v němž odchylky mají hodnoty mocninné řady 2 a pro jejich snadný a rychlý výpočet není nutné je ukládat do pomocného pole. Metoda pracuje takto:

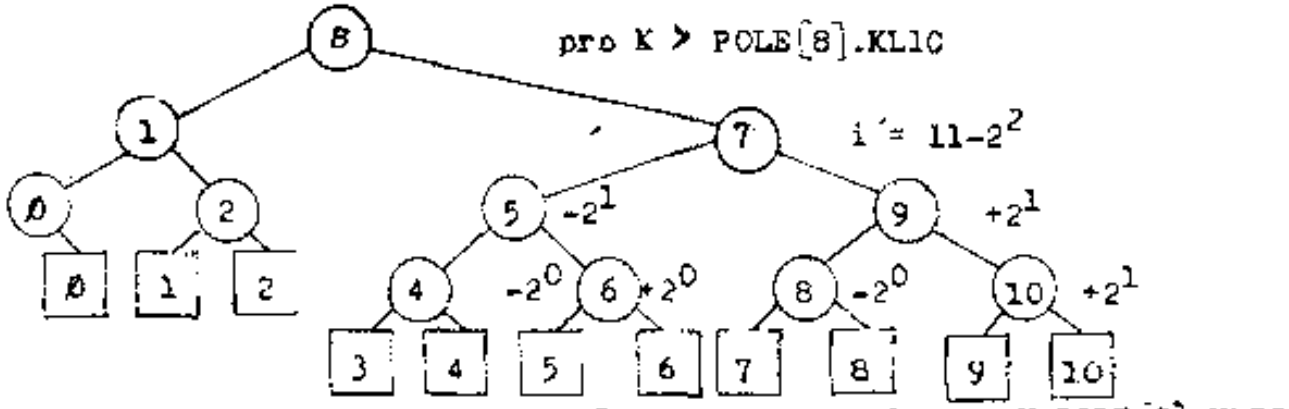
První krok rozdělí pole na indexu $i=2^m$, kde $m=\lfloor lg_2 n \rfloor$.

Je-li klíč $K < POLE[i].KLIC$, budou mít odchylky na sestupujících úrovních hodnoty $2^{m-1}, 2^{m-2}, \ldots, 1, 0$. Je-li naopak $K > POLE[i'].KLIC$ a přitom $n > 2^m$, pak se hodnota indexu i' nastaví na hodnotu $i = n+1-2^{el}$, kde $el = \lfloor lg_2(n-2^m) \rfloor + 1$. V dalších krocích se bude algoritmus chovat tak, jako by výsledkem prvního porovnání bylo, že platí $K > POLE[i'].KLIC$ a uniformní vyhledávání bude pracovat se snižujícími se odchylkami $2^{el-1}, 2^{el-2}, \ldots, 1, 0$.

Na obr. 6 a 7. jsou rozhodovací binární stromy Sharovy metody pro n=10 a pro obě varianty velikosti klíče K vzhledem k hodnotě klíče u položky s indexem $i=2^m$ (m=3).



Obr.6. Sharova metoda pro $K < POLE[8].KLIC$



Obr.7. Sharova metoda pro $K > POLE[8].KLIC$

7. Fibonacciho vyhledávání

Algoritmus Fibonacciho vyhledávání pracuje podobně jako binární vyhledávání. Interval seřazeného pole se však nepůlí, ale dělicí bod

se odvozuje z Fibonacciho posloupnosti a k jeho získání stačí aditivní operace, což zvýší rychlost vyhledávání tam, kde aditivní operace jsou výrazně rychlejší, než multiplikativní operace s celočíselnou hodnotou 2.

Princip metody vychází z Fibonacciho binárního stromu:
Nechť je dána Fibonacciho posloupnost pro jejíž prvky platí:

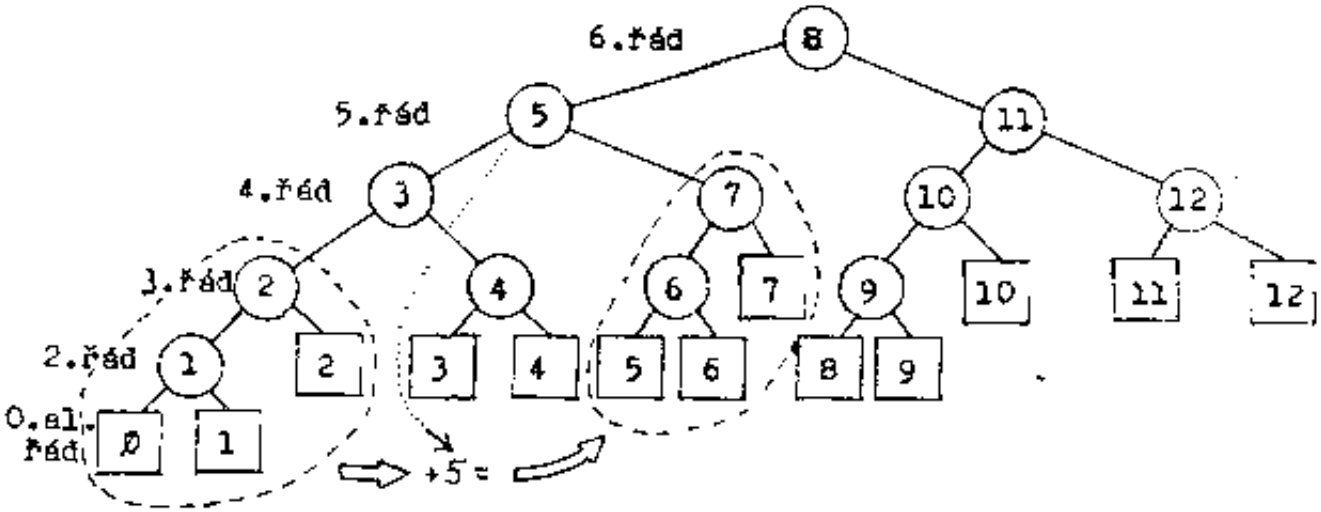
$$F_{\emptyset}=\emptyset\ ,\quad F_1=1,\quad F_{i+1} = F_i + F_{i-1}$$

tzn. $F_{\emptyset}=\emptyset$, $F_1=1$, $F_2=2$, $F_4=3$, $F_5=5$, $F_6=8$, $F_7=13$, $F_8=21$, ...
Fibonacciho strom řádu el má F_{el+1} - 1 uzlů neterminálních (vnitřních) a F_{el+1} uzlů terminálních (listů)

Je-li el=$\emptyset$ nebo el=1, je strom tvořen jediným listem $\boxed{\emptyset}$.
Je-li el=2, pak je kořenem stromu uzel (F_{el}) a přitom:

levý podstrom je Fibonacciho strom řádu el - 1
a pravý podstrom je Fibonacciho strom řádu el - 2, jehož uzly mají hodnotu zvýšenou o F_{el}.

Na obr. 8 je Fibonacciho binární strom 6.řádu.



Obr. 8 Fibonacciho strom 6.řádu

Z obr.78 je vidět, že cestu od nejlevějšího listu ke kořenu stromu tvoří prvky Fibonacciho posloupnosti. Levý podstrom kořene (5) je F-strom 4.řádu a pravý podstrom je F-strom 3.řádu, jehož uzly jsou zvýšeny o hodnotu kořene tj. o 5.

Pro jednoduchost budeme předpokládat, že n+1 je Fibonacciho

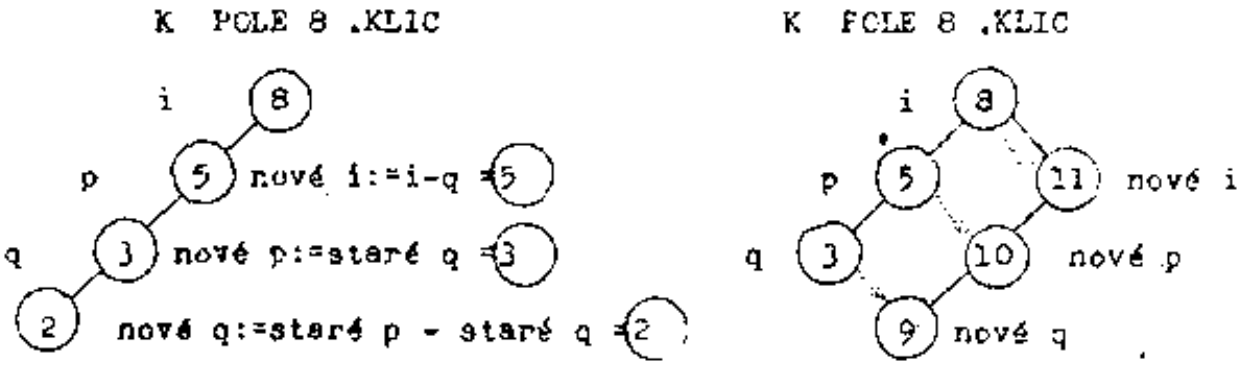
číslo F_{el+1}. Pro jiné n je nutno v prvním kroku provést úpravy podle Sharovy metody. V následujícím algoritmu bude F_{el} funkce pro výpočet prvku Fibonaccího posloupnosti a proměnné p a q budou mít vždy hodnoty dvou po sobě jdoucích čísel. Algoritmus pro vyhledávání v seřazeném poli Fibonaccího metodou pak má tento tvar:

```
          i:= F(el);     {Inicializace pomocných proměnných}
          p:= F(el-1);
          q:= F(el-2);

TERM:=false;  var {TERM:Boolean;  pomocná proměnná k řízení cyklu}

while  (K ≠ POLE [i] .KLIC)  and  (not  TERM)  do
   if  K< POLE [i] .KLIC
       then   {hledá se v levém podstromu}
          if  q=∅
              then TERM:=true   {vyhledání končí v levém podstr.}
              else      {ustaví se nové (i,p,q) pro levý podstrom}
                begin
                   i:=i-q;
                   pl:=q;     {pomocná proměnná  var  pl:integer;}
                   ql:=p-q;   {pomocná proměnná  var  ql:integer;}
                   p:=pl;
                   q:=ql
                end
       else      {hledá se v pravém podstromu}
          if p=1
             then  TERM:=true    {vyhledání končí na pravém listu}
             else        {ustaví se nové  (i,p,q) pro pravý podstrom}
                begin
                   i:=i+q;
                   p:=p-q;
                   q:=q-p
                end;  konec příkazu  if  K< POLE[i] .KLIC a konec
                                cyklu
 QSEARCH := not  TERM;
{if  QSEARCH then  prvek  POLE [i] je hledaným prvkem}
```

Na obr.9 je vysvětleno ustavení (i,p,q) pro levý a pravý podstrom.



Obr.9 Ustavení (i,p,q) pro levý a pravý podstrom

Pro nové hodnoty (i,p,q) v pravém podstromu platí:

nové i:= staré i + q (v obr.9.=11)
nové p:= staré p - staré q (=2 ale hodnota uzlu je 2+8=10)
nové q:= staré q - nové p (=1 , ale hodnota uzlu je 1+8=9)

Hodnoty p a q v pravém podstromu mají hodnoty odpovídajícího levého podstromu (nezvýšené o hodnotu kořene), protože nulová hodnota q a jedničková hodnota p signalizuje ukončení vyhledávání vzhledem k dosažení levého či pravého listu.

(8.) Jiné metody vyhledávání v seřazeném seznamu

Existují ještě jiné principy použitelné pro seřazený seznam. Jejich podstata je důvěrně známá z praktického života.

a) Některé slovníky mají na straně opačné ke hřbetu výřezy, označené jednotlivými písmeny abecedy (tzv. prstové indexy), které umožňují jedním hmatem otevřít slovník na stránce, kde začínají hesla s daným písmenem. Tento princip je podobný indexsekvenčnímu přístupu a proto se mu říká indexsekvenční vyhledávání.

b) hledáme-li ve slovníku bez prstových indexů, "nepůlíme" obvykle plný rozsah slovníku, ale "dělicí" bod získáváme intuitivní interpolecí. Víme-li, že hledaný klíč K leží mezi klíči K_1 a K_2 (kde $K_1<K_2$), pak dělicí bod hledáme na místě blízkém hodnotě $(K-K_1)/(K_2-K)$ a mlčky předpokládáme, že rozdělení klíčů v intervalu

od K_1 do K_2 je rovnoměrné. Tato úvaha je základem interpolačního vyhledávání.

Řada experimentů i praktické zkušenosti (viz /1/) však ukazují, že interpolační metoda aplikovaná na seřazené pole nesníží počet porovnání tak dostatečně, aby se kompenzoval čas, potřebný navíc pro složitější určení dělicího bodu. Metoda může být poněkud úspěšnější při aplikaci na vnějších paměťových médiích. Tento závěr platí také pro indexsekvenční vyhledávání.

9. Z á v ě r

S ohledem na omezený rozsah příspěvků v tomto sborníku, nelze problematiku vyhledávání zdaleka vyčerpat. Z hlediska praktického použití jsou velmi zajímavé dynamicky se chovající implementace pomocí binárních vyhledávacích stromů, a to zejména jejich výškově vyvážené verze (tzv. AVL stromy) a pomocí tabulek s rozptýlenými hesly (tzv. Hashtable). K této problematice bude možné vrátit se v dalších letech.

Literatura

/1/ Knuth,D.: The Art of Computer programming, VOL 3, 1975
/2/ Honzík,J: Dokazování programu, sborník PROGRAMOVÁNÍ '83