6. VYHLEDÁVÁNÍ

Již v úvodu 2.kapitoly jsme se zmínili o skutečnosti, že "schopnost uchovat velký objem informace a posléze k ní umožnit přístup se považuje za nejzávažnější a primární vlastnost počítače". Touto oblastí problémů se zabývá část disciplíny programování nazvaná tradičně "vyhledávání" (angl. searching). Poněkud výstižnější je úplnější název "ukládání informace a její zpřístupňování" (angl. Storage and Retreival of Information). Základním abstraktním typem dat, který s touto problematikou souvisí je vyhledávací tabulka (angl. Look-up Table), kterou budeme zkráceně nazývat tabulka.

Pojem "vyhledávání" je obecně podstatně širší, než je oblast, kterou postihne předmět Programovací techniky a to především tam, kde se dotýká v současné době vysoce aktuální problematiky datových bází. V rámci této kapitoly se seznámíme se základními principy vyhledávání, jejichž aplikace je významná pro řadu složitějších problémů řešených na počítačích.

Algoritmy vyhledávání a řazení (třídění) mají bohatou tradici již z předpočítačové éry a využívají bohatou škálu principů a postupů. Z toho důvodu jsou zajímavé nejen pro řešení daného problému, ale i pro seznámení se s řadou důležitých programovacích technik a obratů. V řadě případů se programy pro vyhledávání a řazení používaly (a používají) pro praktické ověření vlastností programovacího jazyka či počítače i pro srovnání a hodnocení vlastností různých jazyků či počítačů.

Metody vyhledávání můžeme klasifikovat podle různých hledisek. Metody interního vyhledávání pracují s daty uloženými ve vnitřní paměti počítače, zatím co metody externího vyhledávání pracují s daty na vnějších pamětech počítače. Statické metody vyhledávání pracují nad datovou strukturou, která se v průběhu zpracování nemění, zatím co dynamické vyhledávání předpokládá, že v datové struktuře mohou v průběhu zpracování vznikat nové a zanikat nepotřebné položky. Jiné dělení může přihlížet k tomu, zda se pracuje s původními klíči nebo s transformovanými klíči (které vedou k tabulkám s rozptýlenými položkami) - a jsou známa i jiná hlediska.

Algoritmy pro vyhledávání úzce souvisí s algoritmy pro řazení a jejich volba může záviset na řadě okolností. Tuto skutečnost ilustruje následující příklad [1].

Nechť jsou dány dvě množiny čísel :

$A = \{a_1, a_2, \dots, a_m\}$ a $B = \{b_1, b_2, \dots, b_n\}$

Úkolem je zjistit, zda je množina A podmnožinou množiny B ($A \subseteq B$)

Nabízejí se tři typy řešení :

1. Porovnávat každé a_i sekvenčně se všemi b_j a zjišťovat, zda dojde ke shodě
2. Vložit všechna b_j do tabulky a v této tabulce vyhledávat všechna a_i
3. Seřadit zvlášť všechna a_i a všechna b_j a v sekvenčním průchodu hledat shodu

Každé řešení bude výhodné při splnění jistých podmínek. První řešení bude vyžadovat c_1mn časových jednotek, kde c_1 je jistá konstanta. Třetí řešení bude vyžadovat cca $c_2(m \log_2 m + n \log_2 n)$ časových jednotek, kde c_2 je jistá (poněkud větší) konstanta. Při využití tabulky s rozptýlenými položkami bude vyžadovat asi $c_3m + c_4n$ časových jednotek, kde c_3 a c_4 jsou jisté (ještě větší) konstanty. Z toho vyplývá, že pro velmi malá m a n bude nejvýhodnější první řešení, zatímco pro velká m a n bude účelné řešení třetí. Řešení druhé se může ukázat vhodné, pokud n nepřesáhne použitelnou velikost vnitřní paměti. Tento případ ukazuje, že mohou nastat situace, kdy řazení může nahradit vyhledávací mechanismus, a platí to i naopak. <u>Lze však říci, že prvotním účelem řazení je urychlení vyhledávání</u>.

Existují metody, které vyhledávání realizují na základě částečné shody vyhledávaného klíče s klíčem položky v tabulce. Takový případ je velmi aktuální v systémech, v nichž je klíčem např. jméno osoby, které může být zapsáno na základě nesprávně interpretované výslovnosti chybně. Tyto metody vyhledají položku, jejíž klíč s jistou pravděpodobností vznikl z chybně zapsaného klíče.

Metody vyhledávání, které budou podrobněji rozebrány v této kapitole, rozdělíme do čtyř skupin :

1. Sekvenční vyhledávání
2. Nesekvenční vyhledávání v seřazeném poli
3. Vyhledávání v uspořádanych binárních stromech
4. Vyhledávání v tabulce s rozptýlenými položkami

V popisu principu jednotlivých metod se budeme zabývat především mechanismem operace SEARCH. Jejím vedlejším účinkem může být, v případě úspěšného vyhledání údaj, umožňující zpřístupnění vyhledané položky, který je základem operace READ. V diskusi možnosti dynamického chování dané implementace ATD tabulka bude uveden i princip operací INSERT a DELETE.

6.1. Sekvenční vyhledávání

Principy sekvenčního vyhledávání lze ilustrovat na sekvenčním vyhledávání v tabulce implementované lineárním seznamem s použitím základních operací ATD seznam.

6.1.1. Sekvenční vyhledávání v seznamu

Je-li seznam neprázdný, bude mít algoritmus tuto podobu :

```
FIRST(LIST); SEARCH:=false;
repeat if COPY(LIST)=KEY
       then SEARCH:=true
       else SUCC(LIST)
until  not ACTIVE(LIST) and not SEARCH;
```

Může-li být seznam také prázdný, musíme zabránit chybovému stavu operace COPY nad prázdným seznamem. Algoritmus bude mít tento tvar :

```
FIRST(LIST);
SEARCH :=false;
while ACTIVE (LIST) and not SEARCH  do
    begin SEARCH :=COPY(LIST)=KEY;
          if not SEARCH  then SUCC(LIST)
    end;
 { if SEARCH  then aktivní prvek je hledaným prvkem
   Není-li zapotřebí přístup k nalezenému prvku, nemusí být operace
   SUCC(LIST) podmíněná }
```

Pro stanovení časové náročnosti algoritmu rozlišíme tyto případy :

1. čas při neúspěšném vyhledání (Tf)
2. čas při úspěšném vyhledání i-té položky seznamu (Tt_i)
3. průměrný čas pro úspěšné vyhledání, při stejné pravděpodobnosti vyhledávání všech položek (ø Tt)

Jestliže seznam obsahuje n položek, pak

$Tf \approx cn$ (c je konstanta vyjadřující délku jednoho průchodu cyklu)

$Tt_i \approx ci$

$øTt \approx \frac{n+1}{2} c$

Z těchto vztahů vyplývá, že nejrychleji budou vyhledány ty položky, které jsou zařazeny na začátek seznamu. Je-li známa pravděpodobnost vyhledávání jednotlivých položek, je účelné, aby byly položky seznamu seřazeny sestupně podle pravděpodobnosti svého vyhledávání. Na této myšlence jsou založeny složitější "sebeorganizující" algoritmy, které na základě informace o četnosti vyhledávání jednotlivých položek získaných v průběhu chodu programu, reorganizují uspořádání seznamu tak, že často vyhledávané položky umisťují do čela seznamu v naději, že tyto položky budou i v budoucnu vyhledávány často a že úspora času bude větší než čas potřebný na reorganizaci seznamu.

6.1.2. Sekvenční vyhledávání v poli

Nechť jsou dány datové typy :

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

Kde nad typem TYPKLIC je definována relace ekvivalence,
a dále

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

Nechť POLE:TYPPOLE je použito pro implementaci ATD tabulka a K:TYPKLIC má hodnotu vyhledávaného klíče; pak sekvenční vyhledávání v poli realizuje tento algoritmus :

```
i:=Ø; {var i: Ø..MAX}
repeat
   i:=succ(i)
until (i=n) or (POLE[i].KLIC=K); {POLE[n] je poslední aktivní prvek}
SEARCH:=POLE[i].KLIC=K;
{if SEARCH then prvek POLE[i] je hledaným prvkem}
```

6.1.2.1. Rychlé sekvenční vyhledávání v poli

Jednoduchou úpravou lze tento algoritmus zrychlit. Zrychlení spočívá ve snížení hodnoty konstanty c vyjadřující délku průchodu cyklu tím, že se zjednoduší podmínka na konec cyklu. Předpokládejme, že tabulka obsahuje maximálně MAX-1 položek. Pak lze za poslední aktivní prvek vložit před zahájením cyklu "zarážku", což je položka s hodnotou vyhledávaného klíče. Text na konec cyklu již nemusí "hlídat" konec pole. Cyklus skončí vždy "úspěšným vyhledáním", ke kterému dojde buď až na zarážce (t.zn. ve skutečnosti neúspěšné vyhledání) nebo dříve (což znamená úspěšné vyhledání). Důkaz algoritmu je uveden v odst. 13.7.4. - Teorém lineárního vyhledávání. Algoritmus má tvar :

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

Tomuto algoritmu se také říká "sekvenční vyhledávání se zarážkou" (angl. zarážka je "sentinel" nebo "guard").

6.1.2.2. Dynamické vlastnosti sekvenčního vyhledávání v poli

Jestliže vyhradíme dostatečně velké pole, pak konec cyklu vyhledávání i pozice zarážky je určena posledním aktivním prvkem v poli. Při vložení nové položky do tabulky (operace INSERT) se tato pozice zvýší o jedničku a na novou poslední pozici se vloží nový prvek.

Poněkud složitější je (u všech implementací) rušení položky v tabulce (operace DELETE). V zásadě lze volit mezi dvěma přístupy :

a) Ruší-li se i-tý prvek, posune se část pole od i+1 prvku až do posledního aktivního prvku o jednu pozici doleva
b) Klíč rušeného prvku se nastaví na hodnotu, o které je jisto, že nikdy nebude vyhledávána.

Nevýhodou prvního přístupu je potenciálně významná časová náročnost posuvu. Nevýhodou druhého přístupu je, že každé vyřazení způsobí zmenšení použitelného prostoru tabulky "zaslepením" místa rušených prvků. Tuto nevýhodu lze kompenzovat tím, že při vyčerpání celého pole se nové volné místo vyhledává mezi "zaslepenými" položkami. Nevýhodou však nadále zůstává skutečnost, že se zbytečně prohledávají i "zaslepené" položky, což prodlužuje vyhledávání.

6.1.3. Sekvenční vyhledávání v seřazeném seznamu

Je-li nad typem TYPKLIC definována relace uspořádání, lze seznam seřadit (např. vzestupně) podle velikosti klíče. Sekvenční vyhledávání v seřazeném seznamu se pak zrychlí v případě neúspěšného vyhledávání, protože cyklus prohledávání lze ukončit v okamžiku, kdy klíč testované položky je větší, než vyhledávaný klíč. Princip algoritmu této metody je ilustrován s použitím základních operací nad ATD seznam.

```
FIRST(LIST); SEARCH:=false;
while not SEARCH and ACTIVE(LIST) do
        if COPY(LIST) ≤ KEY
              then SEARCH:= true
              else SUCC(LIST);
if SEARCH then SEARCH:=COPY(LIST)=KEY;

 {if SEARCH then aktivní prvek je hledaným prvkem}
```

6.1.3.1. Sekvenční vyhledávání v seřazeném poli

Z předcházejícího algoritmu lze snadno odvodit jeho variantu pracující se seřazeným polem :

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

6.1.3.2. Sekvenční vyhledávání v seřazeném poli se zarážkou

Sekvenční vyhledávání v seřazeném poli lze urychlit s použitím zarážky, která odstraní nutnost testu na konec pole. Hodnota zarážky musí být větší, než hodnoty všech možných vyhledávaných klíčů K. Algoritmus má tvar :

```
i:=∅;
POLE[n+1].KLIC:=MAXKLIC; { MAXKLIC je větší než hodnoty všech vyhledávaných
                          klíčů }
repeat
    i:=succ(i);
until POLE[i].KLIC ≥ K;
SEARCH:=POLE[i].KLIC=K;
{if SEARCH then prvek POLE[i] je hledaným prvkem}
```

6.1.3.3. Dynamické vlastnosti sekvenčního vyhledávání v seřazeném poli

Na rozdíl od neseřazeného pole, je s použitím seřazeného pole nutné při vkládání nového prvku (operace INSERT) zachovat uspořádanost pole. Prvek je nutno zařadit do seřazeného pole, s čímž je obecně spojen posun části pole od i-té pozice do pozice posledního aktivního prvku o jednu pozici doprava. Jestliže operace INSERT následuje po operaci SEARCH, končí předcházející algoritmy nalezením místa (indexu i), na který se po posunu části pole vloží nový prvek.

Operace DELETE se bude provádět posunem části pole o jednu pozici doleva (viz přístup ad a. v odst. 6.1.2.2.). Metoda "zaslepování" položek je principiálně možná, použije-li se klíč, který je menší než všechny možné vyhledávané klíče. Regenerace zaslepených míst však není jednoduchá.

6.2. N e s e k v e n č n í v y h l e d á v á n í v s e ř a z e n é m p o l i

Implementujeme-li vyhledávací tabulku polem seřazeným podle velikosti klíčů položek, nabízejí se pro vyhledávání účinnější algoritmy, než je sekvenční vyhledávání. Tyto metody se poněkud podobají numerickým metodám pro hledání kořene funkce jedné proměnné, známe-li interval, v němž je právě jeden kořen. Zvlášť nápadná je podoba s metodou "půlení intervalu". Na témže principu jsou založeny algoritmy třídy binárního vyhledávání a tzv. Fibonacciho vyhledávání, které budou blíže popsány v tomto odstavci.

6.2.1. Binární vyhledávání

Nechť pro pole implementující vyhledávací tabulku platí :

POLE[1].KLIC < POLE[2].KLIC < ... < POLE[n].KLIC

a dále nechť pro vyhledávaný klíč K platí :

K ≥ POLE[1].KLIC a K≤POLE[n].KLIC

Pak princip binárního vyhledávání lze slovně popsat takto : Vyhledávaný klíč se porovná s klíčem položky, která je umístěna v polovině prohledávaného pole. Dojde-li ke shodě, končí vyhledávání úspěšně. Je-li vyhledávaný klíč menší, postupuje se porovnáváním prostředního prvku v levé polovině původního pole, je-li větší v pravé polovině původního pole. Vyhledávání končí neúspěchem v případě, že prohledávaná část pole je prázdná (t.zn., že její levý index je větší než pravý).

Zápis algoritmu má tvar :

```
ℓ:=1; { var ℓ :1..MAX; ukazatel levé hranice prohledávaného pole }
r:=n; { var r:1..MAX; ukazatel pravé hranice prohledávaného pole }
repeat
    i:=(ℓ+r) div 2;
    if K< POLE[i].KLIC
         then r:=i-1  { hledaný klíč může být v levé polovině }
         else ℓ:=i+1; { hledaný klíč může být v pravé polovině }
until (K=POLE[i].KLIC) or (r < ℓ );
SEARCH:=K=POLE[i].KLIC;
 { if SEARCH then prvek POLE[i] je hledaným prvkem }
```

Dijkstrova varianta binárního vyhledávání vychází z předpokladu, že pole může obsahovat více položek, jejichž klíče se navzájem rovnají. V případě úspěšného vyhledání se nalezne nejlevější položka ze skupiny položek se shodnými klíči. ATD tabulka sice takový předpoklad vylučuje, algoritmus však může být užitečný pro řadu aplikací. Jeho důkaz je uveden v odst. 13.7.3.

Nechť platí :

POLE[0].KLIC ≤ POLE[1].KLIC ≤ ... ≤ POLE[n-1].KLIC < POLE[n].KLIC

a dále nechť pro vyhledávaný klíč K platí :

POLE[1].KLIC ≤ K

a K < POLE[n].KLIC

Pak algoritmus Dijkstrovy varianty binárního vyhledávání má tvar :

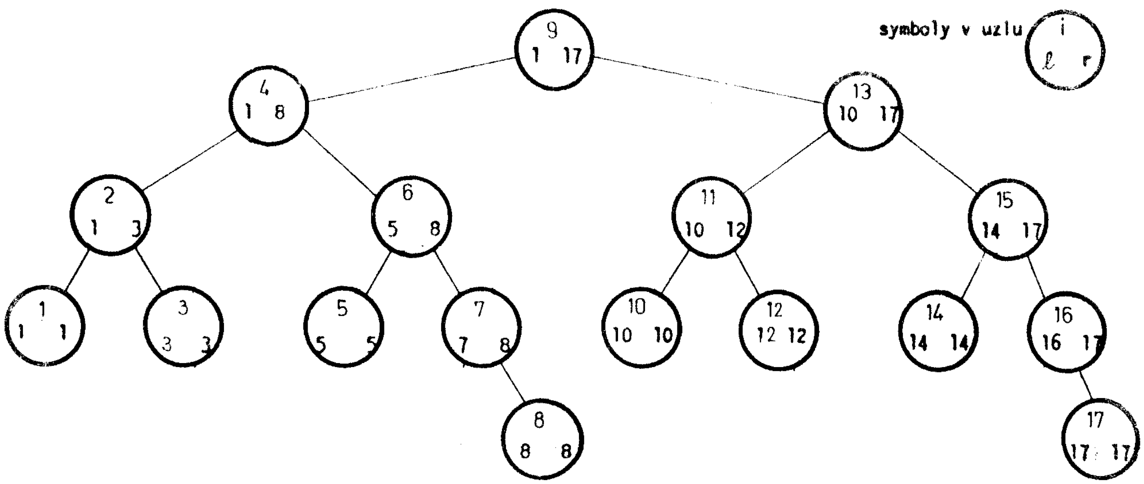
```
ℓ:=0;
r:=n;
while r≠(ℓ+1) do
    begin i:=(ℓ+r) div 2;
         if POLE[i].KLIC ≤ K then ℓ:=i
                              else r:=i
    end;
SEARCH:=K=POLE[i].KLIC;
 { if SEARCH then prvek POLE[i] je nejlevějším prvkem skupiny položek
   s klíčem rovným K }
```

Zatímco u prvního algoritmu může úspěšné vyhledávání trvat kratší dobu než neúspěšné, Dijkstrova varianta má úspěšné i neúspěšné vyhledání stejně dlouhé.

Podrobnější rozbor binárního vyhledávání usnadňuje jeho reprezentace stromovou strukturou.

6.2.1.1. Stromová reprezentace binárního vyhledávání

Mechanismus binárního vyhledávání lze znázornit jeho reprezentací binárním rozhodovacím stromem. Binární strom na obr. 6.1. zobrazuje jednotlivé položky pole s rozsahem indexu 1..17, které mohou být v procesu vyhledávání podle prvního algoritmu postupně testovány. Každý uzel je reprezentován trojicí čísel : indexem i a hranicemi části pole ℓ a r. Jednotlivé cesty od kořene k listu představují postup polem při vyhledávání. **Dojde-li při cestě od kořene k listu ke shodě vyhledávaného klíče s klíčem položky, může vyhledání (podle prvního algoritmu) skončit.**



Obr. 6.1. Rozhodovací binární strom pro binární vyhledávání v poli

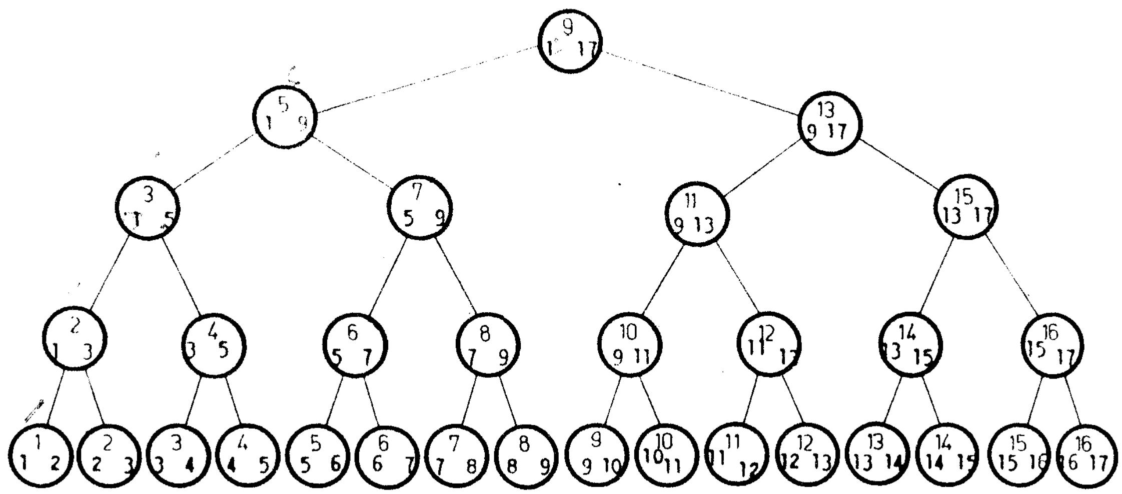
Vyhledává-li se klíč K=POLE[7].KLIC, pak se projde cestou 9,4,6,7 a vyhledání skončí po 4 srovnáních. Vyhledává-li se klíč, pro který platí :
(K < POLE[1Ø].KLIC) and (K > POLE[9].KLIC) projde se cestou 9,13,11,10 a vyhledávání končí, protože po srovnání s POLE[10].KLIC bude r=9 a ℓ =10 a bude splněna podmínka na konec cyklu $r < \ell$.

Rozhodovací binární strom pro Dijkstrovu variantu je na obr. 6.2. Trojice čísel v každém uzlu opět reprezentují index i a hranice intervalu ℓ a r.

Protože cyklus Dijkstrovy varianty končí při splnění podmínky $r=\ell+1$, budou všechna úspěšná i neúspěšná vyhledávání stejně dlouhá (5 porovnání), jak vyplývá z obr. 6.2.

Pozn.: Vzhledem k předběžné podmínce K < POLE[n].KLIC, není pochopitelně součástí stromu uzel 17.

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



Obr. 6.2. Rozhodovací binární strom Dijkstrovy varianty binárního vyhledávání

6.2.1.2. Uniformní binární vyhledávání

Místo tří proměnných i, ℓ a r lze použít pouze dvou : aktuální index i a odchylka m od aktuálního indexu i. Po každém neúspěšném porovnání ustavíme :

i:=i±m   a m:=m div 2

Při návrhu tohoto algoritmu je nutné promyslet všechny detaily, aby nedošlo k nepředvídané chybě.

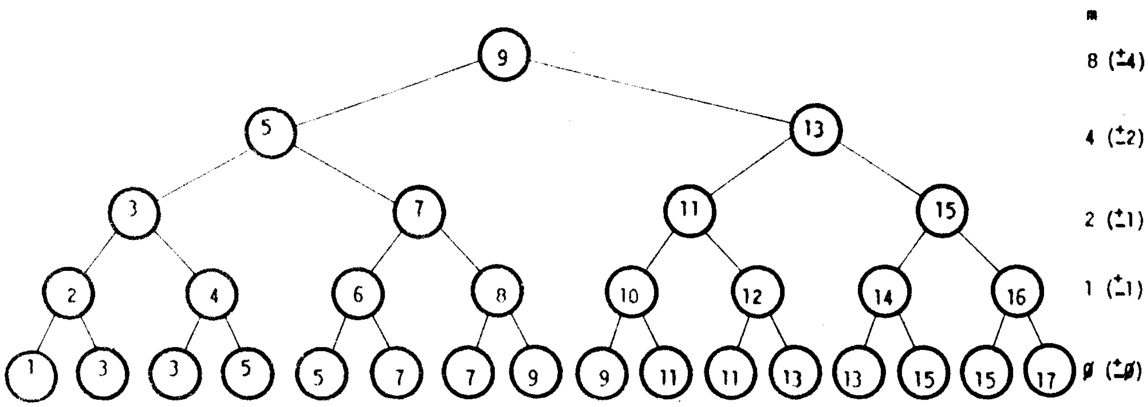
Nechť je dáno pole, pro jehož položky platí :

POLE[1].KLIC < POLE[2].KLIC < ... < POLE[n].KLIC

Je-li MAX sudé, pak algoritmus potřebuje prázdnou (slepou) položku s indexem Ø, jejíž klíč se nastaví na hodnotu menší, než jsou všechny vyhledávané klíče. Algoritmus uniformního binárního vyhledávání má tvar :

```
POLE[Ø].KLIC:=MINKLIC;     { Jen pro sudé MAX }
i:=(n+1) div 2;          { i:=⌈n/2⌉ }
m:= n div 2;             { m:=⌊n/2⌋ }
while (m≠Ø) 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;
SEARCH:=K=POLE[i].KLIC;
 { if SEARCH then prvek POLE[i] je hledaný prvek }
```

Na obr. 6.3. je zobrazen rozhodovací binární strom pro 17 položek pole. Hodnota v uzlu je aktuální index i a vpravo od stromu je vyznačena hodnota m, která je pro všechny uzly téže úrovně stejná. Tato skutečnost je důvodem, proč se vyhledávání jmenuje uniformní. Hlavní přednost této metody se může využít, je-li tabulka statická. V tom případě lze pro daný počet položek vytvořit před



Obr. 6.3. Rozhodovací binární strom pro uniformní binární vyhledávání

vyhledáváním pomocné pole odchylek DELTA a jejím použitím v cyklu vyhledávání odstranit operaci dělení, čímž se algoritmus značně zrychlí (podle [1] i více než 2x). Pro pole DELTA platí :

$$\text{DELTA}\ [j] = \left\lfloor \frac{n+2^{j-1}}{2^j} \right\rfloor = \left\lfloor \frac{n}{2^j} + 0{,}5 \right\rfloor = \text{round}\ (n/2^j)$$

$$\text{pro} \quad 1 \leq j < \lfloor \ln_2(n) + 2 \rfloor$$

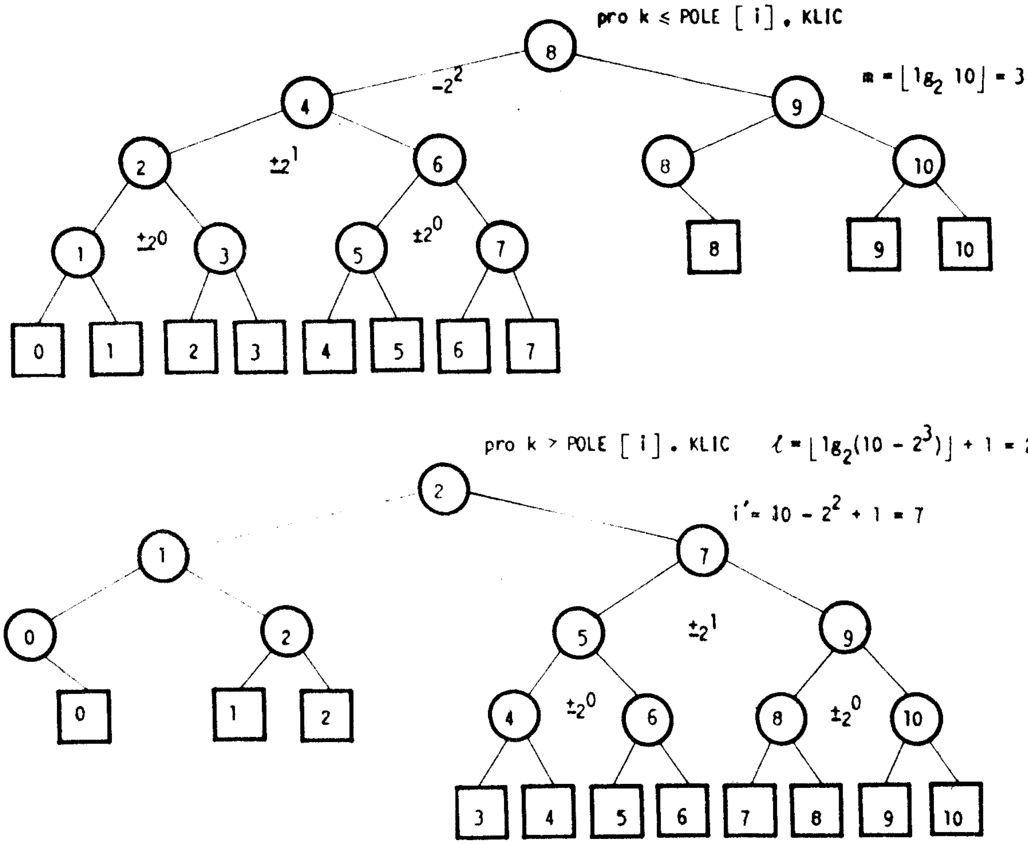
Algoritmus uniformního binárního vyhledávání využívající této výhody má tvar :

```
MOCNINA:=2; { var MOCNINA:POSINT }
for j:=1 to m+2 do      { kde platí 2^(m-1) < n ≤ 2^m }
   begin
        DELTA[j]:=round(n/MOCNINA); {var DELTA:array[1..(m+2)] of POSINT}
        MOCNINA:=MOCNINA*2
   end; { konec vytvoření tabulky DELTA }
i:=DELTA[1];
j:=2;
while (K≠POLE[i].KLIC) and (DELTA[j]≠Ø) do
   begin
        if K < POLE[i].KLIC then i:=i-DELTA[j]
                            else i:=i+DELTA[j];
        j:=succ(j)
   end;
SEARCH:=K=POLE[i].KLIC;
{ if SEARCH then prvek POLE[i] je hledaným prvkem }
```

Variantou uniformního binárního vyhledávání je Sharova metoda [1], která je na některých počítačích rychlejší než uniformní vyhledávání, protože po prvním rozdělení pole je již uniformní podle mocninné řady 2 a nevyžaduje žádnou tabulku odchylek. Metoda pracuje takto :

První krok rozdělí pole na indexu $i=2^m$, kde $m = \lfloor \ln_2 n \rfloor$. Je-li klíč K < POLE[i].KLIC, pokračuje vyhledávání tak, že odchylky na jednotlivých sestupujících úrovních mají hodnoty $2^{m-1}, 2^{m-2}, \ldots, 1, \emptyset$. Je-li naopak klíč K > POLE[i].KLIC a přitom $n > 2^m$, pak se hodnota indexu i nastaví na hodnotu $i' = n+1-2^{\ell}$, kde $\ell = \lfloor \ln_2(n-2^m) \rfloor + 1$. 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 hodnotami odchylky $2^{\ell-1}$, $2^{\ell-2}$, ..., 1, ∅.

Na obr. 6.4.a je znázorněn rozhodovací binární strom pro Sharovu metodu pro pole o 10 prvcích.



Obr. 6.4. Rozhodovací binární strom pro Sharovu metodu

6.2.2. Fibonacciho vyhledávání

Z předcházejících odstavců vyplývá, že existuje vzájemný vztah mezi binárním vyhledáváním v seřazeném poli a binárním stromem. Algoritmus Fibonacciho vyhledávání pracuje podobně jako binární vyhledávání. Daný interval v poli však nedělí na dvě poloviny, ale dělicí bod odvozuje z Fibonacciho posloupnosti a k jeho získání stačí aditivní operace, což zvýší rychlost vyhledávání tam, kde aditivní operace jsou výrazně rychlejší než celočíselné dělení číslem 2.

V řadě metod hraje Fibonacciho posloupnost podobnou roli, jako mocninná řada dvou v obdobných metodách. K pochopení Fibonacciho vyhledávání se jen ztěží obejdeme bez Fibonacciho rozhodovacího stromu.

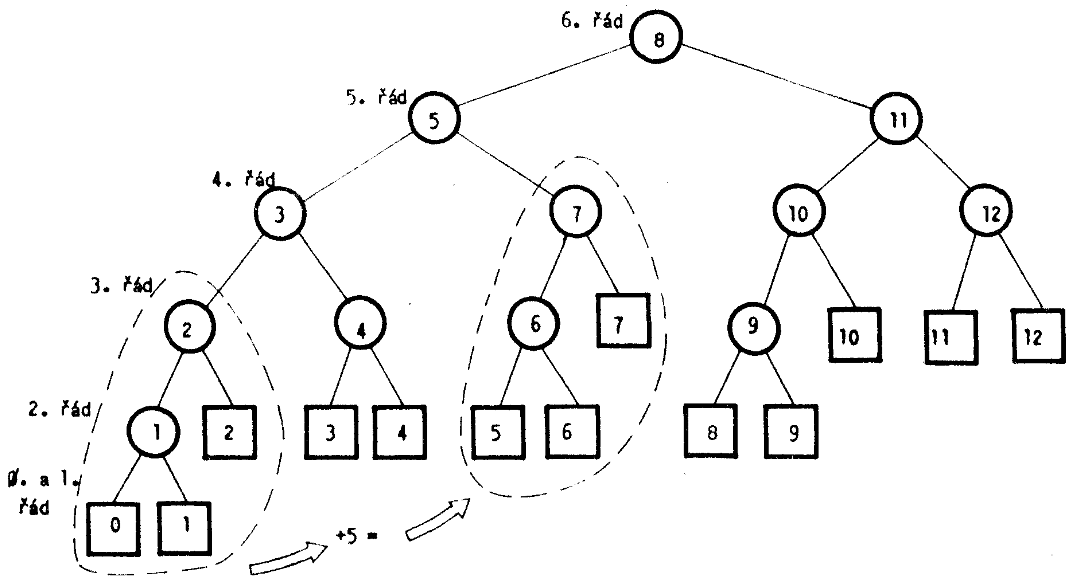
Fibonacciho strom řádu ℓ má $F_{\ell+1}-1$ uzlů neterminálních a $F_{\ell+1}$ listů.
Je-li $\ell = ∅$ nebo $\ell = 1$ je strom tvořen jen listem $\boxed{∅}$.
Je-li $\ell \geq 2$, pak kořenem stromu je uzel F_ℓ
přitom levý podstrom je Fibonacciho strom řádu $\ell - 1$
a pravý podstrom je Fibonacciho strom řádu $\ell - 2$
jehož uzly mají hodnotu zvýšenou o F_ℓ

V tab. 6.1. jsou hodnoty F_i Fibonacciho posloupnosti

i	0	1	2	3	4	5	6	7	8	...
F_i	0	1	1	2	3	5	8	13	21	...

Tab. 6.1. Hodnoty Fibonacciho posloupnosti $F_{i+1}=F_i+F_{i-1}$ pro $F_o=\emptyset$ a $F_1=1$

Na obr. 6.4. je zobrazen Fibonacciho strom 6.řádu. Všimněme si, že oba "synovské" uzly každého vnitřního (neterminálního) uzlu se liší od svého "otcovského" uzlu o stejnou hodnotu a touto hodnotou je opět Fibonacciho číslo.



Obr. 6.5. Fibonacciho strom 6.řádu

Na obr. 6.5. je vidět, že cestu od nejlevějšího listu ke kořeni tvoří čísla Fibonacciho posloupnosti. Levý podstrom kořene (5) je Fibonacciho strom 4.řádu a pravý podstrom je strom 3.řádu, jehož všechny uzly jsou zvýšeny o hodnotu kořene "otcovského" stromu, t.j. o hodnotu 5. Situaci znázorňuje na obr. 6.4. čárkované ohraničení podobných podstromů 3.řádu. Prověřte, že totéž platí pro pravý podstrom kořene (8).

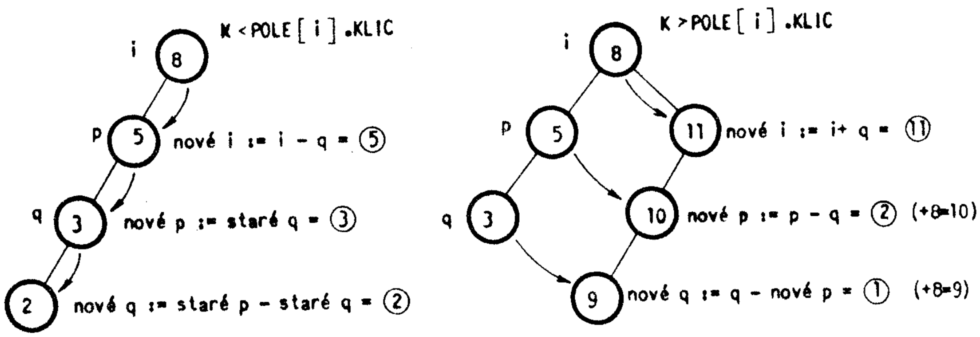
Pro jednoduchost budeme předpokládat, že n+1 je Fibonacciho číslo $F_{\ell+1}$. Pro jiné n je nutno provést potřebné počáteční úpravy podobné Sharově metodě. V následujícím algoritmu budeme používat funkce F(ℓ) pro určení Fibonacciho čísla řádu

Proměnné p a q budou mít vždy hodnoty dvou po sobě jdoucích Fibonacciho čísel. Algoritmus Fibonacciho vyhledávání má tvar :

```
i:=F(ℓ);      {Inicializace pomocných proměnných}
p:=F(ℓ-1);
q:=F(ℓ-2);
TERM:=false;
while (K≠POLE[i].KLIC) and (not TERM) do
   if K < POLE[i].KLIC
   then {hledá se v levém podstromu}
      if q = Ø
```

```
            then TERM:=true    {vyhledávání končí na levém listu}
            else     {ustaví se nové hodnoty i,p a q pro levý podstrom}
            begin i:=i-q;
                  pl:=q;          {pomocná proměnná pl}
                  ql:=p-q;        {pomocná proměnná ql}
                  p:=pl;
                  q:=ql
            end
      else {hledá se v pravém podstromu}
        if p=l then TERM:=true    {vyhledávání končí na pravém listu}
               else  {ustaví se nové hodnoty i,p a q pro pravý podstrom}
                 begin  i:=i+q;
                        p:=p-q;
                        q:=q-p
                 end; {konec if K < POLE[i].KLIC a konec cyklu while}
      SEARCH:=not TERM;
      {if SEARCH then prvek POLE[i] je hledaným prvkem}
```

Na obr. 6.6. je zobrazeno ustavení nových hodnot i,p a q pro levý a pravý podstrom



Obr. 6.6. Ustavení nových hodnot i,p a q

Hodnoty p a q v pravém podstromu mají hodnoty původního levého odpovídajího podstromu (nezvýšené o hodnotu kořene), protože nulová hodnota q a jedničková hodnota p znamená ukončení vyhledávání v důsledku toho, že došlo k listům (viz listy ∅ a 1 v obr. 6.4.).

6.2.3. Jiné metody vyhledávání v seřazeném seznamu

Má-li se vyhledávat prvek v seznamu seřazeném podle velikosti klíčů jednotlivých položek, nabízejí se principiálně i jiné metody, které známe dobře z praxe při vyhledávání žádaného hesla ve slovníku, či osoby v telefonním seznamu. V tom případě lze vyhledávání usnadnit s využitím těchto principů :

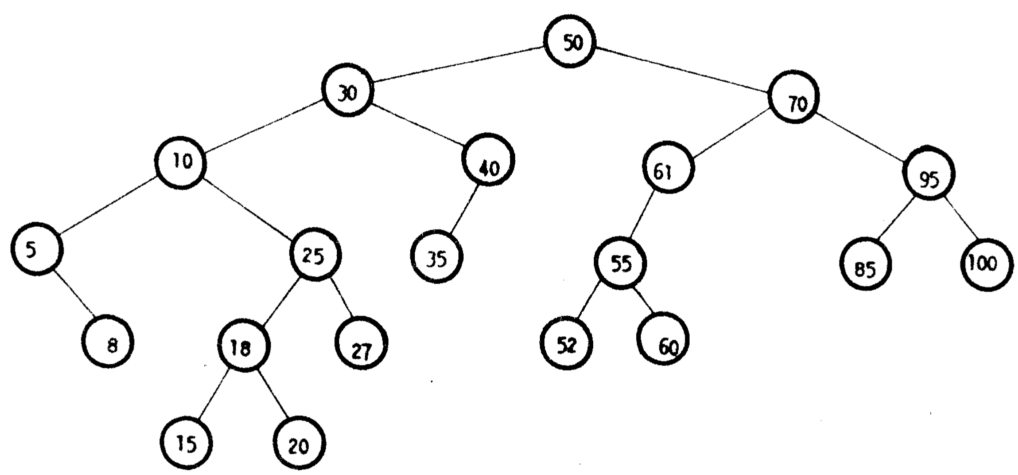
a) Některé slovníky mají na straně opačné, než je hřbet výřezy, označené jednotlivými písmeny abecedy (tzv. "prstové indexy"), které umožňují jedním hmatem otevřít slovník na stránce, od které jsou uvedena hesla začínající na dané písmeno. Tento princip je podobný indexsekvenčnímu přístupu k seznamu (k vhodn místu seznamu se dostaneme pomocí "indexu" a dále postupujeme sekvenčně) a proto mu říkáme indexsekvenční vyhledávání. Tento přístup je vhodný především pro externí vyhledávání.

b) Hledáme-li ve slovníku, který není vybaven prstovými indexy, jistě nevyhledáváme tak, že bychom "půlili" plný rozsah slovníku abychom zjistili, zda je hledané heslo v levé nebo v pravé polovině, ale dělicí bod získáváme intuitivní interpolací. Víme-li, že hledaný klíč **K** leží mezi klíči K_1 a K_2 ($K_1 < K_2$), pak dělicí bod hledáme v 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 vyhledávání na vnějších paměťových zařízeních.

6.3. Binární vyhledávací stromy

Binární vyhledávací strom (dále zkráceně BVS, anglicky - binary search tree) je takový binární uspořádaný strom, pro jehož každý uzel platí, že jeho levý podstrom je buď prázdný, nebo sestává z uzlů, hodnoty jejichž klíčů jsou menší, než hodnota klíče daného uzlu a podobně jeho pravý podstrom je buď prázdný, nebo sestává z uzlů, hodnoty jejichž klíčů jsou větší, než hodnota klíče daného uzlu. V příkladu BVS na obr. 6.7. jsou do uzlů vepsány hodnoty jejich klíčů (typu integer).



Obr. 6.7. Příklad uspořádání BVS

V předcházející části kapitoly byla uvedena souvislost mezi nesekvenčním vyhledáváním v seřazeném poli a uspořádaným binárním stromem, která vyplývá ze stromové interpretace binárního a Fibonacciho vyhledávání. Vzájemnost tohoto vztahu doplňuje vlastnost průchodu typu INORDER binárním vyhledávacím stromem, kterým získáme seřazený lineární seznam. Průchodem INORDER stromem na obr. 6.7. získáme sekvenci : 5,8,10,15,18,20,25,27,30,35,40,50,52,55,60,61,70,85,95,100.

Protože základní operací nad ATD tabulka je operace SEARCH a na ní závislá operace READ, budeme se nejdříve zabývat implementací těchto operací v BVS. Strom je typická dynamická datová struktura, a proto v dalších odstavcích rozebereme také operace INSERT a DELETE.

6.3.1. Vyhledávání v BVS - operace SEARCH

Nechť jsou dány typy :

```
TYPUKUZEL= ↑TYPUZEL;
TYPUZEL = record
            DATA:TYPDATA;
            KLIC:TYPKLIC;
            LEVY,PRAVY:TYPUKUZEL
          end
```

Nechť je dále dán binární vyhledávací strom, jehož uzly jsou typu TYPUZEL a nechť je dána proměnná KOREN typu TYPUKUZEL, která ukazuje na kořen tohoto stromu. Algoritmus vyhledání uzlu stromu, jehož složka KLIC je shodná s vyhledávaným klíčem K typu TYPKLIC lze zapsat s pomocí rekurze nebo nerekurzívně.

6.3.1.1. Rekurzívní zápis vyhledávání

Princip rekurzívního zápisu algoritmu vyhledávání ve stromu je založen na rekurzívnosti datové struktury strom. Algoritmus má tvar (upraveno podle [2]):

```
function SEARCH (KOREN:TYPUKUZEL; K:TYPKLIC):Boolean;
  {Funkce má hodnotu true, jestliže v BVS daném ukazatelem KOREN
   existuje uzel, jehož složka KLIC se rovná vyhledávanému klíči K}
begin
  if KOREN ≠ nil
    then if KOREN↑.KLIC = K
           then SEARCH:=true      {Našel se uzel s klíčem K}
           else if KOREN↑.KLIC > K
                  then SEARCH:=SEARCH (KOREN↑.LEVY,K)  {Hledej v levém
                                                        podstromu}
                  else SEARCH:=SEARCH (KOREN↑.PRAVY,K) {Hledej v pravém
                                                        podstromu}
    else SEARCH:=false  {Cesta končí v listu - neúspěšné vyhledání}
end  {funkce SEARCH}
```

Tato funkce odpovídá specifikaci operace SEARCH tak, jak byla uvedena v před cházející kapitole. Při praktické práci s ATD tabulka je však často vhodný takový druh operace, jehož vedlejším účinkem je v případě úspěšného vyhledávání lokalizace a zpřístupnění nalezeného prvku. Tohoto účinku se může využít k implementaci operace READ. Nechceme-li použít další parametr (např. parametr var KDE:TYPUKUZEL pro zpřístupnění nalezeného prvku, můžeme pro tento účel použít parametr KOREN s vědomím, že procedura změní jeho původní hodnotu. Protože funkce s výstupními parametry odporuje zásadám správného programování, bude mít operace tvar procedury :

```
procedure SEARCHTREE (var KOREN:TYPUKUZEL; K:TYPKLIC);
 {Procedura hledá uzel s klíčem K ve stromu zadaném ukazatelem KOREN;
  najde-li uzel, zpřístupní ho ukazatelem KOREN, nenajde-li uzel, bude
  mít  ukazatel KOREN hodnotu nil }
begin
   if KOREN ≠ nil
        then if KOREN↑.KLIC ≠ K
                then begin
                     if KOREN↑.KLIC > K
                           then KOREN:=KOREN↑.LEVY
                           else KOREN:=KOREN↑.PRAVY;
                      SEARCHTREE (KOREN,K)
                     end
end  {Procedury SEARCHTREE}
```

Po vyvolání procedury SEARCHTREE lze ustavit hodnotu Booleovské proměnné SEARCH na základě hodnoty parametru KOREN takto :

```
SEARCH:=KOREN ≠ nil;
 {if SEARCH then ukazatel KOREN zpřístupňuje nalezený prvek tabulky }
```

Korektnější podoba této procedury bude mít čtyři parametry a tedy hlavičku :

```
procedure SEARCHTREE (KOREN:TYPUKUZEL; K:TYPKLIC; var SEARCH:Boolean;
                      var KDE:TYPUKUZEL);
```

6.3.1.2. Nerekurzívní zápis vyhledávání

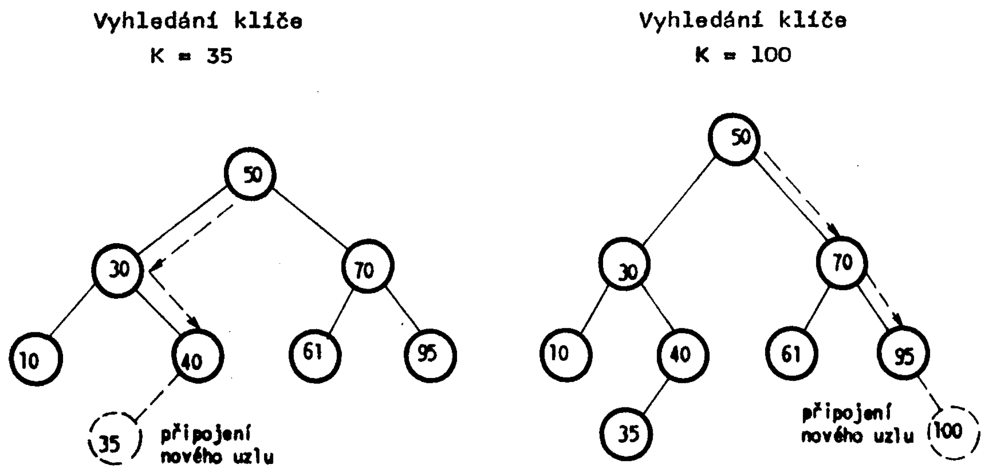
Nerekurzívní zápis algoritmu operace SEARCH má tvar :

```
function SEARCH (KOREN:TYPUKUZEL; K:TYPKLIC):Boolean;
var KONEC:Boolean;    {pomocná proměnná pro řízení cyklu}
begin
   SEARCH:=false;
   KONEC:=KOREN=nil;
   while not KONEC do
      begin
         if KOREN↑.KLIC=K
            then begin KONEC:=true; {Úspěšný konec vyhledávání}
                       SEARCH:=true
                 end
            else if KOREN↑.KLIC > K
                    then KOREN:=KOREN↑.LEVY   {Pokračuj v levém podstromu}
                    else KOREN:=KOREN↑.PRAVY; {Pokračuj v pravém podstromu}
         if KOREN=nil then KONEC:=true
      end  {cyklu while}
end  {funkce SEARCH}
```

Odvození nerekurzívního zápisu algoritmu operace SEARCHTREE, která v případě úspěšného vyhledání zpřístupní nalezený uzel stromu ponecháme čtenářům.

6.3.2. Vkládání prvku do BVS - operace INSERT

Podle sémantické definice přepíše operace INSERT hodnotu prvku, který byl v tabulce nalezen. Nebyl-li v tabulce nalezen prvek s daným klíčem, operace INSERT vloží do tabulky nový prvek s tímto klíčem. Je-li tabulka implementována binárním vyhledávacím stromem, končí neúspěšné vyhledávání právě na tom listu stromu, na který se má připojit nový prvek. Je-li neúspěšně vyhledaný klíč menší než klíč listu, připojí se nový uzel vlevo k danému listu, je-li větší, připojí se vpravo k danému listu. Situaci neúspěšného vyhledání a vložení znázorňuje pro hodnoty K=35 a K=100 obr. 6.8.



Obr. 6.8. Vkládání prvku do BVS

Algoritmus operace INSERT může mít rekurzívní i nerekurzívní zápis.

6.3.2.1. Rekurzívní zápis operace INSERT

Rekurzívní zápis procedury INSERT je velice krátký a přehledný (upraveno podle [2]) :

```
procedure INSERT (var KOREN:TYPUKUZEL; K:TYPKLIC; DATAUZLU: TYPDATA);
 { procedura vloží do stromu prvek s klíčem K a se složkou DATAUZLU }
begin
   if KOREN = nil
        then { Prvek s klíčem K není prvkem stromu; vloží se nový prvek }
            begin
               new (KOREN);
               with KOREN↑do  {Ustavení hodnoty, klíče a ukazatelů nového uzlu
                 begin KLIC:=K;
                        LEVY:=nil;
                        PRAVY:=nil;
                        DATA:=DATAUZLU
                 end
            end
```

```
else  {Zatím se hledaný uzel nenašel a ještě se nedošlo k listu }
   if K < KOREN↑.KLIC
       then {Pokračuj v levém podstromu}
             INSERT (KOREN↑.LEVY,K,DATAUZLU)
       else
             if K > KOREN↑.KLIC
                   then  {Pokračuj v pravém podstromu}
                         INSERT (KOREN↑.PRAVY,K,DATAUZLU)
                   else  {Uzel s klíčem K je prvkem stromu,
                           jeho datová složka se přepíše}
                         KOREN↑.DATA:=DATAUZLU
end {procedury INSERT}
```

6.3.2.2. Nerekurzivní zápis operace INSERT

Nerekurzivní zápis procedury INSERT je podstatně delší a vyžaduje samostatný mechanismus vyhledání.

```
procedure INSERT (var KOREN:TYPUKUZEL; K:TYPKLIC; DATAUZLU:TYPDATA);
     var POMUK, KDE:TYPUKUZEL;  {Pomocné proměnné pro příkaz new a pro
                                  lokalizaci místa vkládání }
        NASEL:Boolean;   {Řídicí proměnná cyklu }
        procedure INSERTSEARCH (KOREN:TYPUKUZEL; K:TYPKLIC; var NASEL:
                                Boolean; var KDE:TYPUKUZEL);
 {Tělo procedury bude rozvedeno později. Procedura vyhledává v BVS prvek
  s klíčem K. Nalezne-li jej, pak NASEL=true a KDE zpřístupňuje nalezený
  prvek. Jinak je NASEL=false a KDE zpřístupňuje list, na který se připojí
  nový uzel }
begin  {Začátek těla procedury INSERT }
   INSERTSEARCH (KOREN,K,NASEL,KDE);  {Vyhledání }
   if NASEL
       then KDE↑.DATA:=DATAUZLU {Přepis datové složky nalezeného uzlu }
       else  {prvek nebyl nalezen, bude se vkládat nový uzel }
         begin new (POMUK);
               with POMUK↑do  {Ustavení hodnot složek nového uzlu }
                 begin DATA:=DATAUZLU; KLIC:=K;
                       LEVY:=nil; PRAVY:=nil
                 end;
               if KDE=nil
                 then KORENː=POMUK  {Strom byl prázdný; nový uzel se stane
                                     kořenem }
                 else  {Strom byl neprázdný, uzel se připojí k listu }
                    if KDE↑.KLIC > K
                       then KDE↑.LEVY:=POMUK  {Připojení uzlu k listu vlevo
                       else KDE↑.PRAVY:=POMUK {Připojení uzlu k listu vprav
         end  {příkazu if NASEL }
end  {procedury INSERT}
```

Procedura vyhledávání za účelem vkládání uzlu má tvar :

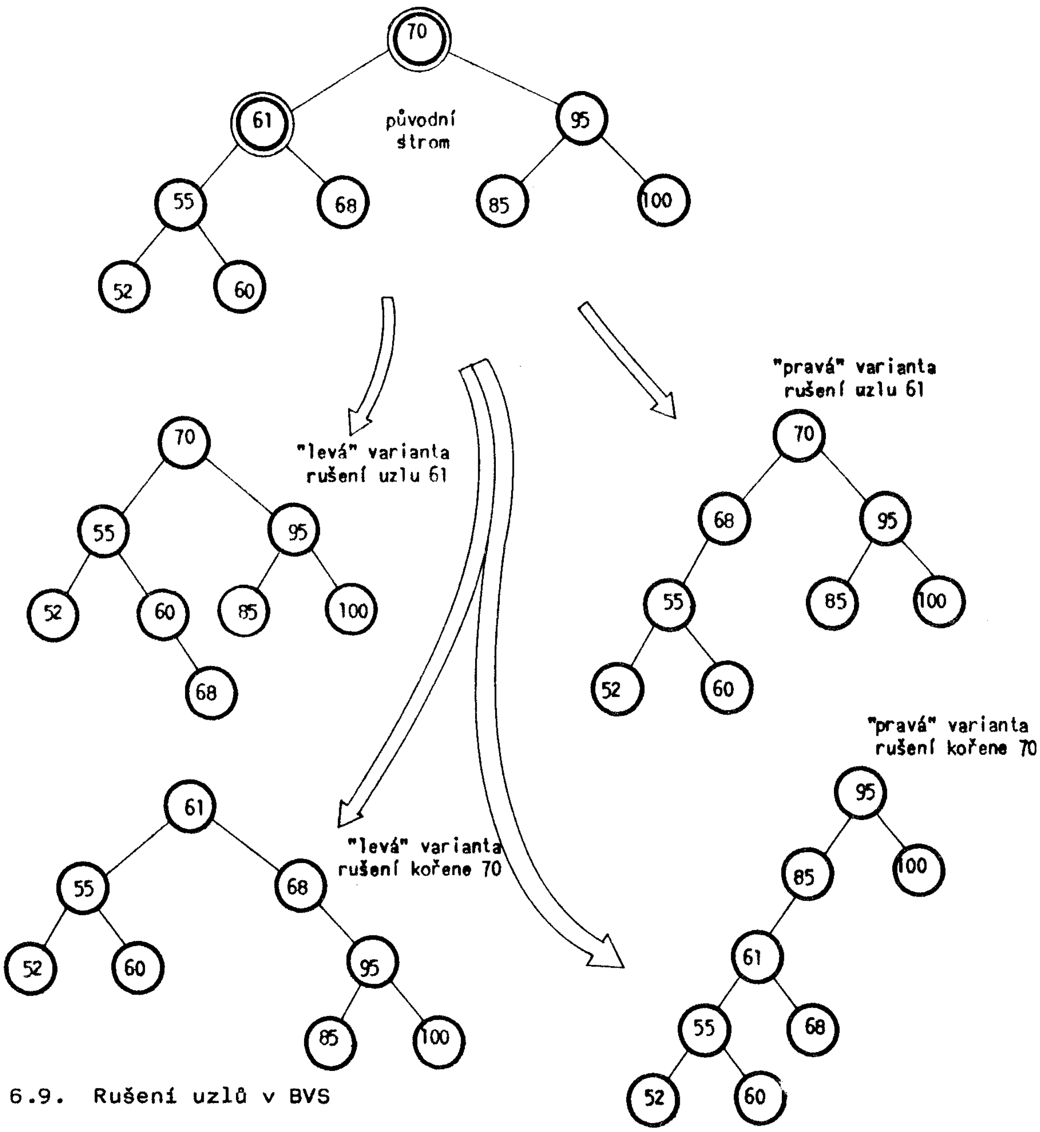
```
procedure INSERTSEARCH (KOREN:TYPUKUZEL; TYPKLIC; var NASEL:Boolean;
                        var KDE:TYPUKUZEL);
var KONEC:Boolean; {Pomocná proměnná pro řízení cyklu}
begin
  NASEL:=false;
  if KOREN=nil
     then {Strom je prázdný}
          KDE:=nil
     else {Strom je neprázdný}
       begin
          KONEC:=false;
          while not KONEC do
             begin KDE:=KOREN
                   if KOREN↑.KLIC=K
                      then {Úspěšné vyhledání}
                         begin NASEL:=true
                               KONEC:=true
                         end
                   else begin if KOREN↑.KLIC > K
                                 then KOREN:=KOREN↑.LEVY
                                 else KOREN:=KOREN↑.PRAVY;
                                 KONEC:=KOREN=nil
                        end
              end {cyklu while}
         end
end {procedury INSERTSEARCH}
```

6.3.3. Rušení prvku v BVS - operace DELETE

U všech předcházejících implementačních metod pro ATD tabulka bylo vždy rušení prvku v tabulce obtížnější než vkládání. Binární vyhledávací strom je typický svou vhodností pro implementaci dynamické tabulky. Přesto je vyřazování uzlu ze stromu (operace DELETE) složitější, než vkládání nového uzlu (operace INSERT). Jeden z možných postupů při vyřazování uzlu znázorňuje obr. 6.9., na němž je "levá" a "pravá" varianta rušení uzlů. Jako první se v původním stromu ruší vnitřní uzel 61, v druhé ukázce se ruší kořen-uzel 70.

Mechanismus tohoto způsobu rušení lze popsat slovně takto :

Rušíme-li neterminální uzel, který má "synovské" uzly, pak levý synovský uzel připojíme k nadřazenému (praotcovskému) uzlu místo rušeného (otcovského) uzlu a pravý synovský uzel připojíme k nejpravějšímu listu podstromu levého synovského uzlu (tzv. "levá" varianta), nebo provedeme stranově sdruženou variantu ("pravá" varianta). Je-li levý nebo pravý synovský podstrom prázdný, pak se situace zjednoduší: na místo rušeného otcovského uzlu se připojí neprázdný synovský podstrom. Ruší-li se kořen, stane se kořenem levý (resp. pravý) synovský uzel a pravý (resp. levý) podstrom se připojí k nejpravějšímu (resp. nejlevějšímu) list levého (resp. pravého) podstromu. Vyřazovaný prvek se zruší operací typu DISPOSE



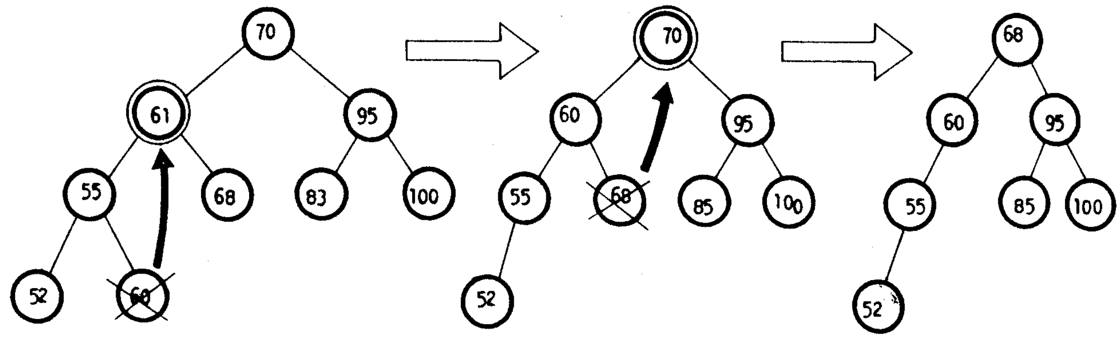
Obr. 6.9. Rušení uzlů v BVS

Vážnou nevýhodou tohoto řešení je skutečnost, že rušením uzlů (zvláště těch, které jsou blízké kořenu) se zvyšuje počet úrovní (výška) stromu. Tím se prodlužuje cesta k listům stromu a tudíž i proces vyhledávání. Nebudeme proto tuto metodu dále rozvíjet. Z cvičných důvodů lze doporučit vytvoření procedury DELETE pracující podle této metody.

Výhodnější řešení nabízí mechanismus, vycházející z této úvahy: rušení listu a uzlu, jehož jeden z podstromů je prázdný, je velmi jednoduché. Položme si otázku, zda existuje list, nebo uzel jen s jedním synem, jehož hodnotu lze vložit do rušeného uzlu (přepsat hodnotu rušeného uzlu), takový, že strom, který vznikne tímto přepisem a zrušením takového listu (nebo uzlu jen s jedním synem), splňuje podmínky BVS? Takovým uzlem je nejpravější uzel levého podstromu "rušeného" uzlu.[x] Mechanismus postupného rušení uzlů 61 a 70 touto metodou znázorňuje obr. 6.10.

---

[x] I zde lze vytvořit stranově souměrnou variantu "nejlevějšího uzlu pravého podstromu".



Obr. 6.10. Vyřazování uzlů v BVS

6.3.3.1. Rekurzívní zápis operace DELETE

Rekurzívní zápis operace DELETE je podobně jako v předcházejících případech kratší a elegantnější než jeho nerekurzívní podoba. Rekurzívní zápis je však obtížnější a nelze očekávat, že by takový algoritmus samostatně vytvořil nezkušený programátor. Procedura DELETE je po úpravách převzata z [2].

```
procedure DELETE (var KOREN:TYPUKUZEL; K:TYPKLIC);
 {Procedura DELETE vyhledá ve stromu zadaném ukazatelem na kořen (KOREN)
  uzel s klíčem K a zruší tento uzel druhou z metod uvedených v předcháze-
  jícím odstavci. Nenajde-li se uzel s klíčem K, neprovede procedura žádné
  změny nad stromem a na tuto situaci dále nereaguje. Případnou reakci lze
  vložit na komentářem vyznačené místo}
  var POMUK:TYPUKUZEL;  {Pomocný ukazatel na rušený uzel}
  procedure DEL (var UK:TYPUKUZEL);
   {Pomocná procedura DEL prochází po nejpravější větvi levého podstromu
    vyřazovaného uzlu (POMUK) a hledá nejpravější uzel (UK). Po jeho nale-
    zení přepíše jeho hodnotou datovou složku a klíč uzlu POMUK, a uvolní
    uzel UK tak, aby po skončení procedury mohl být zrušen příkazem dispose}
    begin if UK↑.PRAVY ≠ nil
            then DEL(UK↑.PRAVY)  {hledá dále v pravém podstromu}
            else {Nalezl nejpravější, provede přepis a uvolnění uzlu UK}
              begin
                 POMUK↑.KLIC:=UK↑.KLIC;   {Přepis složky KLIC}
                 POMUK↑.DATA:=UK↑.DATA;   {Přepis složky DATA}
                 POMUK:=UK;
                 UK:=UK↑.LEVY  {Uvolnění uzlu UK↑.Pozor! UK je
                                 v proceduře DEL ukazatelová složka
                                nadřazeného uzlu k uzlu UK↑!}
              end
    end; {Pomocné procedury DEL}
    begin { Začátek těla procedury DELETE }
       if KOREN ≠ nil
          then {Hledání není u konce; hledaný prvek ještě může
                být ve stromu}
            if K < KOREN↑.KLIC
```

```
                  then DELETE (KOREN↑.LEVY,K) {Hledej v levém podstromu}
                  else if K > KOREN↑.KLIC
                    then DELETE (KOREN↑.PRAVY,K) {Hledej v pravém podstromu}
                    else {hodnota uzlu KOREN↑ se ruší}
                      begin
                          POMUK:=KOREN;
                          if POMUK↑.PRAVY=nil
                            then  {Nemá pravý podstrom; uvolní se tím,
                                   že se levý podstrom připojí na nad-
                                   řazený uzel}
                                   KOREN:=POMUK↑.LEVY
                            else  {Má pravý podstrom; uzel se bude pře-
                                   pisovat nejpravějším v levém podstro-
                                   mu procedurou DEL,
                                   je-li levý strom neprázdný; je-li
                                   levý strom prázdný, připojí se pravý
                                   podstrom na nadřazený uzel}
                              if POMUK↑.LEVY=nil
                                then KOREN:=POMUK↑.PRAVY {připojení
                                                 pravého podstromu}
                                else DEL (POMUK↑.LEVY);
                        dispose (POMUK)  {zrušení uvolněného uzlu}
                    end {za else na tomto místě lze reagovat na nenalezení
                         prvku ve stromu}
      end  {Procedury DELETE}
```

6.3.3.2. Nerekurzivní zápis operace DELETE

Nerekurzivní zápis operace DELETE je sice průhlednější, ale delší. Uveďme proto napřed její návrh.

```
     procedure DELETE (var KOREN:TYPUKUZEL; K:TYPKLIC);
     begin
        DELETESEARCH (KOREN,K,NASEL,OTECLEVY,PRAOTEC,OTEC);
        {Procedura hledá ve stromu KOREN uzel s klíčem K. Je-li strom
         prázdný pak OTEC=nil, NASEL=false,OTECLEVY a PRAOTEC nedefinované.
         Je-li nalezený uzel kořen, pak PRAOTEC=nil, OTEC=KOREN, NASEL=true,
         OTECLEVY nedefinovaný. Je-li nalezený uzel uvnitř stromu, pak NASEL=
         =true, PRAOTEC je uzel nadřazený OTCI, OTEC=nalezený uzel, OTECLEVY=
         =true, je-li OTEC levým synem PRAOTCE a OTECLEVY=false, je-li pra-
         vým synem. Nenašel-li se hledaný uzel, pak NASEL=false a ostatní
         výstupní parametry nejsou definované.}
      if NASEL then
            begin if OTEC↑.PRAVY=nil
                    then {rušený nemá pravý podstrom}
                      begin if PRAOTEC=nil
                              then KOREN:=OTEC↑.LEVY {Rušený je kořen}
                              else [PŘIPOJ LEVÉHO SYNA NA PRAOTCE]
                      end
                    else if OTEC↑.LEVY=nil
```

```
                                         then  {rušený nemá levý podstrom}
                                            begin if PRAOTEC=nil
                                                     then KOREN:=OTEC↑.PRAVY
                                                          {Rušený je kořen}
                                                     else [PŘIPOJ PRAVÉHO SYNA NA PRAOTCE]
                                            end
                                         else RIGHTMOST(OTEC);
                                              {Procedura najde nejpravější uzel levého
                                               podstromu OTCE. Přepíše hodnoty OTCE,
                                               uvolní nejpravější uzel a předá ho v pa-
                                               rametru OTEC ke zrušení}
                               dispose (OTEC)
                    end
          end
```

Z návrhu vyplývá potřeba dvou pomocných procedur - DELETESEARCH a RIGHTMOST a také realizace bloků připojení SYNA NA PRAOTCE. Procedura DELETE pak bude mít tento tvar :

```
procedure DELETE(var KOREN:TYPUKUZEL;K:TYPKLIC);
    var OTEC,PRAOTEC:TYPUKUZEL;
        NASEL,OTECLEVY:Boolean;
    procedure DELETESEARCH (KOREN:TYPUKUZEL;K:TYPKLIC;var NASEL,OTECLEVY:Boolean;
                            var OTEC,PRAOTEC:TYPUKUZEL);
     {Tělo procedury bude uvedeno později}
    procedure RIGHTMOST(var OTEC:TYPUKUZEL);
     {Tělo procedury bude uvedeno později}
    begin {začátek těla procedury DELETE}
      DELETESEARCH(KOREN,K,NASEL,OTECLEVY,OTEC,PRAOTEC)
      if NASEL then
              begin if OTEC↑.PRAVY=nil
                       then if PRAOTEC=nil
                               then KOREN:=OTEC↑.LEVY
                               else if OTECLEVY  {Připoj levého syna na praotce}
                                       then PRAOTEC↑.LEVY:=OTEC↑.LEVY
                                       else PRAOTEC↑.PRAVY:=OTEC↑.LEVY
                               else if OTEC↑.LEVY=nil
                                       then if PRAOTEC=nil
                                               then KOREN:=OTEC↑.PRAVY
                                               else if OTECLEVY {Připoj pravého
                                                         syna na praotce}
                                                       then PRAOTEC↑.LEVY:=
                                                            OTEC↑.PRAVY
                                                       else PRAOTEC↑ .PRAVY:=
                                                            OTEC↑.PRAVY
                                       else RIGHTMOST(OTEC);
                 dispose (OTEC)
              end
end {procedury DELETE}
```

Nyní uvedeme rozvedení procedury DELETESEARCH, která provede vyhledávání prvku ve stromu za účelem jeho zrušení.

```
procedure DELETESEARCH(KOREN:TYPUKUZEL;K:TYPKLIC;var NASEL,OTECLEVY:Boolean;
                        var OTEC,PRAOTEC:TYPUKUZEL);
   var KONEC:Boolean; {Pomocná proměnná pro řízení cyklu}
   begin
         NASEL:=false;
         OTEC:=KOREN;
         if KOREN≠nil
         then if KOREN↑.KLIC=K
                 then begin NASEL:=true; {Nalezený uzel je kořen stromu}
                            PRAOTEC:=nil
                      end
                 else begin KONEC:=false;
                            while not KONEC do
                              begin if OTEC↑.KLIC=K
                                       then begin NASEL:=true;
                                                  KONEC:=true
                                            end
                                       else begin PRAOTEC:=OTEC;
                                                  if OTEC↑.KLIC > K
                                                     then begin OTECLEVY:=true;
                                                                OTEC:=OTEC↑.LEVY
                                                          end
                                                     else begin OTECLEVY:=false:
                                                                OTEC:=OTEC↑.PRAVY
                                                          end;
                                            end;
                                    if OTEC=nil then KONEC:=true
                              end {cyklu while}
                      end
end  {procedury DELETESEARCH}
```

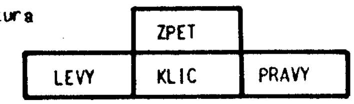
Procedura RIGHTMOST bude mít tvar :

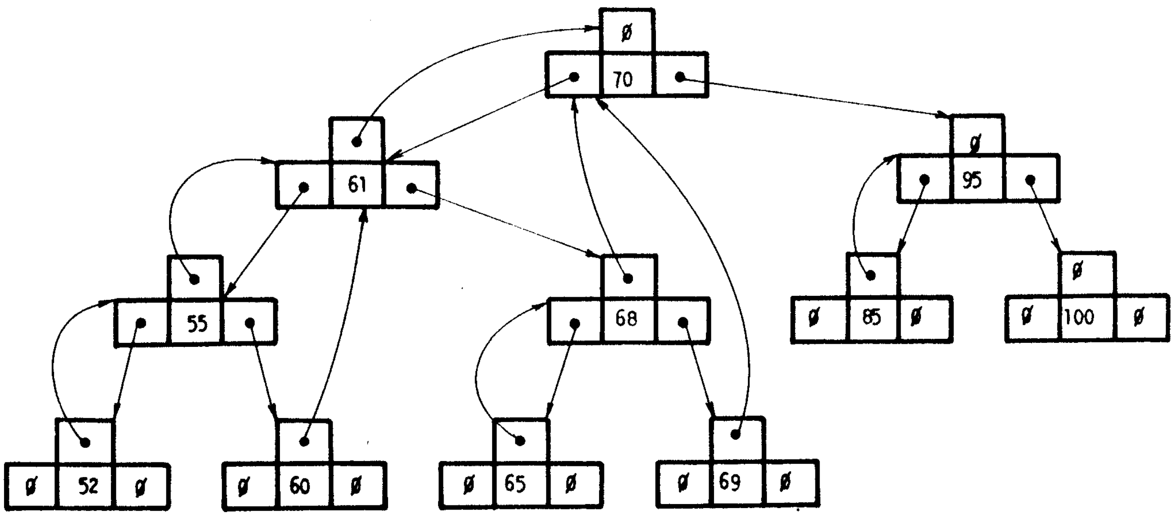
```
procedure RIGHTMOST (var OTEC:TYPUKUZ);
var NEJPRAV, OTECNEJPRAV:TYPUKUZ;
  begin
      NEJPRAV:=OTEC↑.LEVY;
      if NEJPRAV↑.PRAVY≠nil
         then {hledej nejpravějšího}
             begin repeat OTECNEJPRAV:=NEJPRAV;
                          NEJPRAV:=NEJPRAV↑.PRAVY
                   until NEJPRAV↑.PRAVY=nil;
                   OTECNEJPRAV↑.PRAVY:=NEJPRAV↑.LEVY
             end
         else {NEJPRAV je sám nejpravější}
              OTEC↑.LEVY:=NEJPRAV↑.LEVY;
      OTEC↑.KLIC:=NEJPRAV↑.KLIC;     {Přepis klíče}
      OTEC↑.DATA:=NEJPRAV↑.DATA;     {Přepis údajů}
      OTEC:=NEJPRAV  {OTEC se bude rušit pomocí dispose}
  end
```

6.3.4. BVS se zpětnými ukazateli uzlů

BVS se zpětnými ukazateli uzlů je strom, jehož uzly jsou rozšířeny o jeden ukazatel, který umožňuje nerekurzivní zápis průchodu typu INORDER bez použití zásobníku. Na obr. 6.11. je takový strom, kterému se také anglicky říká "Monkey-puzzle tree" [6], zobrazen.

Struktura uzlu





Obr. 6.11. BVS se zpětnými ukazateli

Předpokládejme, že definován typ TYPUKUZEL=↑TYPUZEL a

```
TYPUZEL=record KLIC:TYPKLIC;
               DATA:TYPDATA;
               LEVY,PRAVY,ZPET:TYPUKUZEL;
        end
```

Pak průchod typu INORDER s vkládáním navštívených uzlů do lineárního seznamu abstraktní operací OUT lze zapsat následujícím úsekem programu.

```
LEFTMOST (KOR,UK);
{Procedura LEFTMOST vyhledá nejlevější uzel stromu KOR a ukazatel
 na něj odkazující vloží do UK}
 KONEC:=UK≠nil;  {var KONEC:Boolean}
 while not KONEC do
   begin OUT (UK↑.DATA);
         if UK↑.PRAVY≠nil
            then LEFTMOST (UK↑.PRAVY,UK)
            else if UK↑.ZPET=nil
                     then KONEC:=true
                     else UK:=UK↑.ZPET
  end {cyklu while}
  OUT(UK↑.DATA);
```

Pro vkládání nových uzlů do BVS se zpětnými ukazateli uzlů platí tato pravidla :

a) Je-li vkládán uzel do prázdného stromu, nastaví se všechny ukazatelé na hodnotu nil

b) Je-li vkládán uzel do neprázdného stromu pak :

   A) Je-li vkládaný uzel levým uzlem, bude jeho zpětný ukazatel ukazovat na nejblíže nadřazený (otcovský uzel)

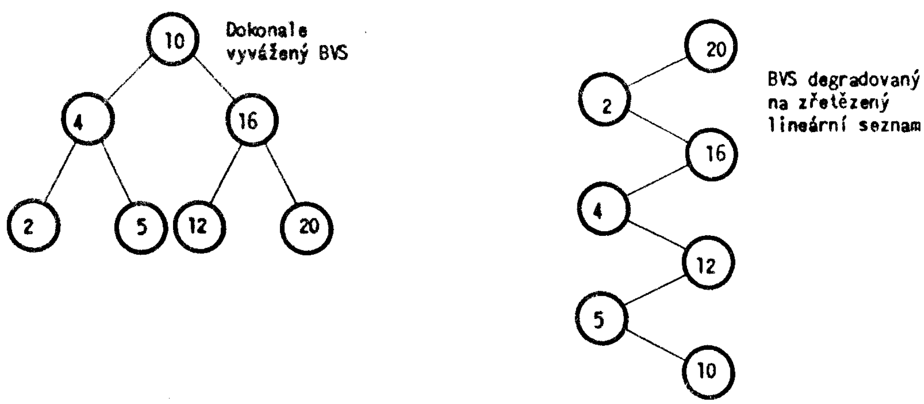
   B) Je-li vkládaný uzel pravým uzlem, "zdědí" vkládaný uzel hodnotu zpětného ukazatele po svém "otci" (t.zn. že bude ukazovat tam, kam ukazoval jeho otec)

Pro rušení uzlu v BVS se zpětnými ukazateli platí stejná pravidla, jaká byla uvedena v předcházejících odstavcích.

6.3.5. Vyvážené binární stromy

Délka vyhledávání ve stromové struktuře záleží na uspořádání stromu. Nejhorší případ neúspěšného vyhledávání je dán nejdelší cestou od kořene k listu stromu. Z toho vyplývá, že ideálně uspořádaný je strom, u něhož jsou délky všech cest od kořene k listům stejně dlouhé. V této souvislosti se hovoří o vyvážených binárních stromech.

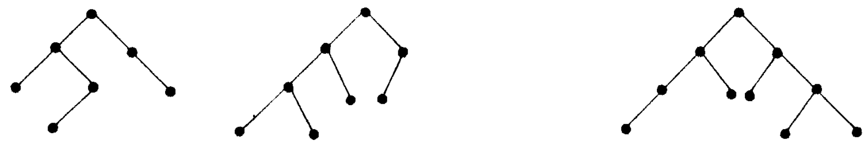
Dokonale vyvážený binární strom je strom, pro jehož každý uzel platí, že počet uzlů v jeho levém a pravém podstromu se liší maximálně o 1. Není obtížné vytvořit dokonale vyvážený BVS. Obtížné je však udržet stav vyváženosti po každé operaci INSERT a DELETE. Skutečnost, že vyvažování stromu je aktuální, ukazuje obr. 6.12, na němž je dokonale vyvážený BVS o 7 uzlech a "dokonale nevyvážený" strom (degradovaný na zřetězený seznam), který vznikl opakovanou operací INSERT pro postupně vkládané uzly: 20,2,16,4,12,5,10.



Obr. 6.12. Příklady vyvážených a nevyvážených BVS

Adělson-Velskij a Landis [7] definovali méně přísné požadavky na vyvážení stromu, které mají velký praktický význam právě pro dynamické BVS. Stromům podle jejich definice se říká AVL-stromy (podle jmen autorů), nebo také výškově vyvážené stromy. Výškově vyvážený strom je strom, pro jehož každý uzel platí, že výška obou jeho podstromů se liší maximálně o 1.

Adělson-Velskij a Landis dokázali, že AVL-strom není vyšší o více než o 45% než odpovídající dokonale vyvážený strom. Na obr. 6.13. jsou příklady AVL-stromů.



Obr. 6.13. Příklady AVL-stromů

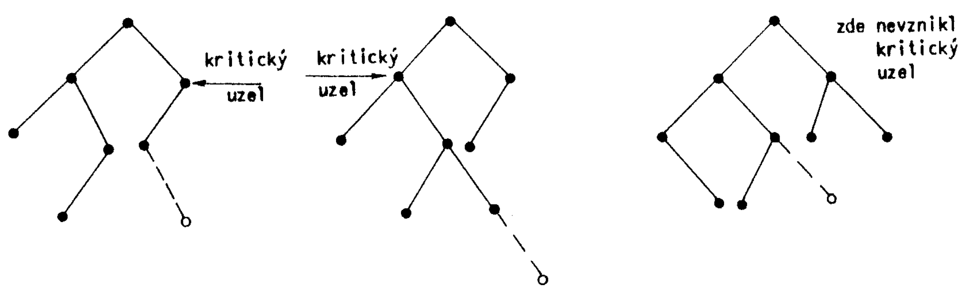
Pro AVL-stromy zavedeme některé nové pojmy :

a) vyvážený uzel AVL-stromu je uzel, jehož oba podstromy jsou stejně vysoké
b) vlevo těžší uzel AVL-stromu je uzel, jehož levý podstrom je o 1 vyšší než pravý podstrom
c) vpravo těžší uzel AVL-stromu je uzel, jehož pravý podstrom je o 1 vyšší než levý podstrom

Připojíme-li operací INSERT nový uzel, může dojít k těmto stavům :

a) uzel těžší na jednu stranu se stane vyváženým uzlem
b) vyvážený uzel se stane uzlem těžším na jednu stranu
c) uzel těžší na jednu stranu se stane nevyváženým

Případ ad c. poruší pravidlo AVL-stromu a proto bude třeba znovu ustavit výškovou vyváženost stromu. Poruší-li se rovnováha ve stromu, nesplňuje řada uzlů pravidlo vyváženosti. Ten z nich, který je nejdál od kořene , se nazývá kritický uzel. Kritický uzel je vždy na cestě od vloženého listu (který porušil rovnováhu) ke kořeni. Na obr. 6.14. jsou uvedeny příklady vkládání nových uzlů do AVL-stromu.

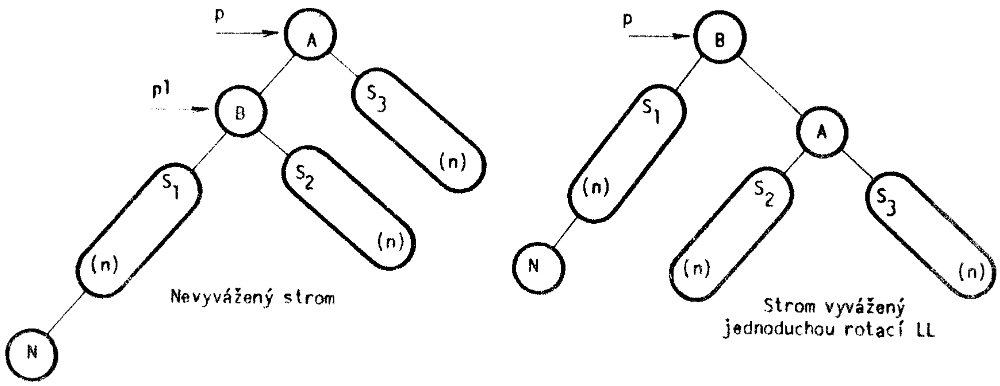


Obr. 6.14. Přidávání uzlu do AVL-stromů

Na obr. 6.14. a 6.15. jsou znázorněny základní situace při porušení rovnováhy a mechanismy jejího znovuustavení. Na těchto obrázcích je N v kroužku nový vkládaný uzel, jehož vložením se porušila rovnováha AVL-stromu. Oválem je označen podstrom S_1; jeho výška je uvedena v závorce. Předpokládá se, že vkládaný uzel byl připojen k nejnižšímu listu podstromu. Na obr. 6.15. je znázorněn výsledek operace "jednoduchá rotace LL", která transformuje nevyvážený strom na vyvážený při zachování pravidel BVS.

Algoritmus jednoduché rotace LL má tento tvar :

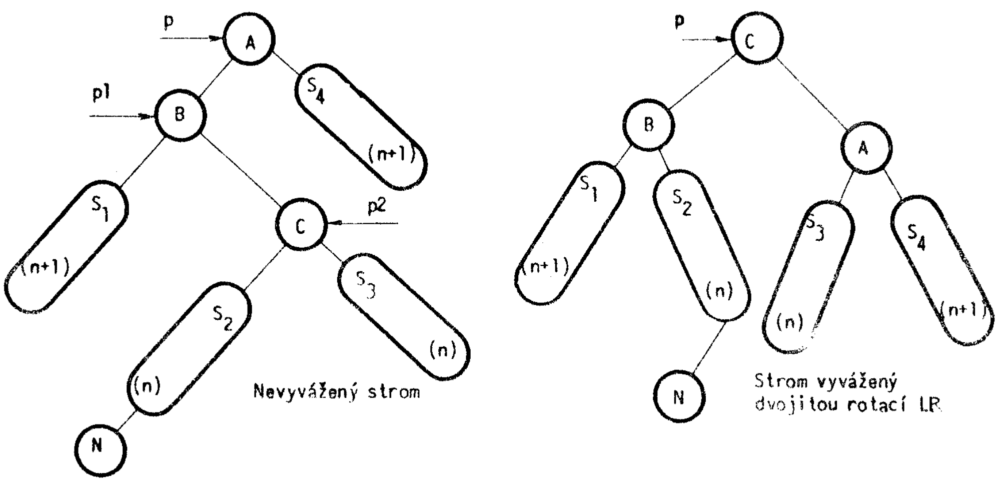
```
pl:=p↑.LEVY; {ustavení pl}
p↑.LEVY:=pl↑.PRAVY;
pl↑.PRAVY:=p;
p:=pl;
```



Obr. 6.15. Mechanismus jednoduché rotace LL

Stranově symetrickou situaci řešíme pomocí jednoduché rotace RR, jejíž odvození ponechejme čtenářům.

Na obr. 6.16. je znázorněn mechanismus operace "dvojité rotace LR".



Obr. 6.16. Vyvážení stromu dvojitou rotací LR

Algoritmus dvojité rotace LR má tento tvar :

```
pl:=p↑.LEVY;    {ustavení pl}
p2:=pl↑.PRAVY; {ustavení p2}
pl↑.PRAVY:=p2↑.LEVY;
p2↑.LEVY:=pl;
p↑.LEVY:=p2↑.PRAVY;
p2↑.PRAVY:=p;
p:=p2;
```

Stranově symetrická situace se řeší dvojitou rotací RL, jejíž odvození opět ponechejme čtenářům.

6.3.5.1. Rekurzivní zápis operace INSERT v AVL-stromu

Nechť jsou dány tyto typy pro práci s AVL stromem :

```
TYPUKUZEL=↑TYPUZEL;
TYPUZEL=record KLIC:TYPKLIC;
               DATA:TYPDATA;
               BALANC:-1..1;  {vyváženost uzlu}
               LEVY,PRAVY:TYPUKUZEL
        end
```

Složka BALANC vyjadřuje, zda je uzel těžký vlevo (-1), vpravo (1), nebo zda je vyvážený (Ø).

Algoritmus operace má tři po sobě jdoucí části :

a) Průchod stromem až do zjištění, že hledaný uzel není prvkem stromu
b) Připojení nového uzlu a určení výsledného faktoru vyvážení (BALANC)
c) Chod stromem zpátky a kontrola faktoru vyvážení v každém uzlu

Při chodu zpátky stromem je nutno znát, zda se výška podstromu zvýšila, či ne. Tuto informaci zajišťuje parametr h (který je vstupní i výstupní). Jeho počáteční hodnota při vyvolání procedury INSERT je h=false.

```
procedure INSERT (var KOR:TYPUKUZEL;K:TYPKLIC;var h:Boolean;DATAUZLU:TYPDATA);
var p1, p2:TYPUKUZEL;
begin
   if KOR=nil
     then {Uzel není ve stromu}
       begin new(KOR); h:=true;
             KOR↑.LEVY:=nil; KOR↑.PRAVY:=nil;
             KOR↑.KLIC:=K;KOR↑.DATA:=DATAUZLU;
             KOR↑.BALANC:=Ø;
       end
     else {Hledej dále}
       if K < KOR↑.KLIC
          then {hledej v levém podstromu}
               begin HLEDEJ(KOR↑.LEVY,K,h,DATAUZLU);
                     if h then  {výška levého stromu se zvýšila}
                        case KOR↑.BALANC of
                           1: {Uzel těžší vpravo se vyvážil}
                                begin KOR↑.BALANC:=Ø; h:=false
                                end;
                           Ø: {vyvážený uzel se stal těžší vlevo}
                              KOR↑.BALANC:=-1;
                          -1: {Uzel těžší vlevo porušil rovnováhu, musí
                               se vyvážit }
                              [USTAV VYVÁŽENÍ ROTACÍ LL nebo LR]
                        end  {příkazu case}
               end {hledání v levém podstromu}
          else {v levém podstromu není}
               if K > KOR↑.KLIC
                  then {hledej v pravém podstromu}
                       begin HLEDEJ (KOR↑.PRAVY,K,h,DATAUZLU);
                             if h then {výška pravého stromu se zvýšila}
                                case KOR↑.BALANC of
```

```
                                        -1: {uzel těžší vlevo se vyvážil}
                                             begin KOR↑.BALANC:=Ø;
                                                       h:=false
                                             end;
                                        Ø: {vyvážený uzel se stal těžší
                                             vpravo}
                                             KOR↑.BALANC:=1;
                                        1: {uzel těžší vpravo porušil
                                             rovnováhu, musí se vyvážit}
                                             USTAV VYVÁŽENÍ ROTACÍ RR NEBO RL
                                   end {příkazu case}
                              end {hledání v pravém podstromu}
                         else {Uzel s klíčem K se našel}
                              begin KOR↑.DATA:=DATAUZLU;
                                    h:=false
                              end
end  {procedury INSERT}
```

Blok označený "USTAV VYVÁŽENÍ ROTACÍ LL NEBO LR" má tvar :

```
begin p1:=KOR↑.LEVY; {Uzel těžší vlevo porušil rovnováhu, musí se vyvážit}
      if p1↑.BALANC=-1
           then {jednoduchá rotace LL}
                begin KOR↑.LEVY:=p1↑.PRAVY; p1↑.PRAVY:=KOR;
                      KOR↑.BALANC:=Ø; KOR:=p1
                end {jednoduché rotace LL}
           else  {dvojitá rotace LR}
                begin p2:=p1↑.PRAVY; p1↑.PRAVY:=p2↑.LEVY;p2↑.LEVY:=p1;
                      KOR↑.LEVY:=p2↑.PRAVY;p2↑.PRAVY:=KOR;
                      if p2↑.BALANC=-1 then KOR↑.BALANC:=1
                                       else KOR↑.BALANC:=Ø;
                      if p2↑.BALANC=1 then p1↑.BALANC.=-1
                                      else p1↑.BALANC:=Ø;
                      KOR:=p2
                end;{dvojité rotace LR}
      KOR↑.BALANC:=Ø;
      h:=false
end  {ustavení vyvážení rotací LL nebo LR}
```

Blok označený "USTAV VYVÁŽENÍ ROTACÍ RR NEBO RL" má tvar :

```
begin  {uzel těžší vpravo porušil rovnováhu, musí se vyvážit}
      p1:=KOR↑.PRAVY;
      if p1↑.BALANC=1
          then {jednoduchá rotace RR}
                begin KOR↑.PRAVY:=p1↑.LEVY;p1↑.LEVY:=KOR;
                      KOR↑.BALANC:=Ø;KOR:=p1;
                end {jednoduché rotace RR}
          else {dvojitá rotace RL}
                begin p2:=p1↑.LEVY; p1↑.LEVY:=p2↑.PRAVY;p2↑.PRAVY:=p1;
                      KOR↑.PRAVY:=p2↑.LEVY;p2↑.LEVY:=KOR;
                      if p2↑.BALANC=1 then KOR↑.BALANC:=-1
```

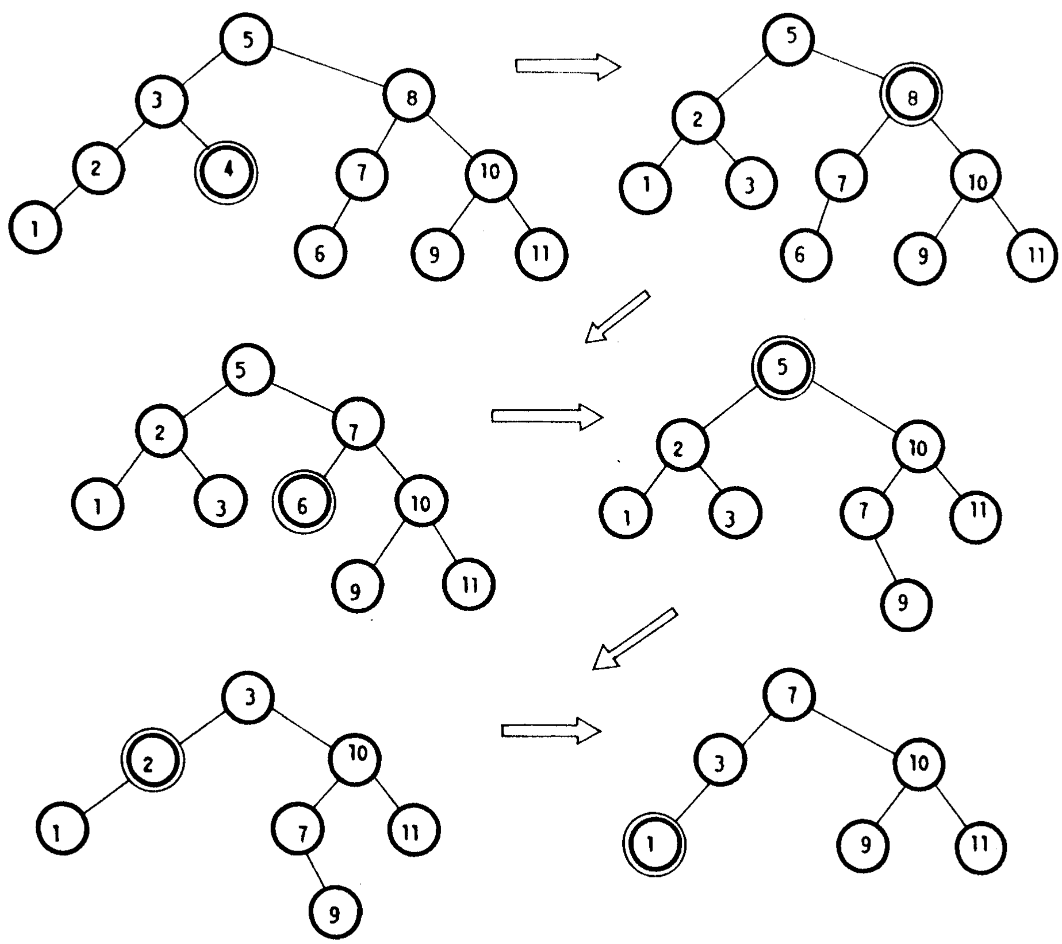
```
                                        else KOR↑.BALANC:=Ø;
                        if p2↑.BALANC=-1 then p1↑.BALANC:= 1
                                        else p1↑.BALANC:=Ø;
                        KOR:=p2
                    end; {dvojité rotace RL}
   KOR .BALANC:=Ø;
   h:=false
end   {ustavení vyvážení rotací RR nebo RL}
```

6.3.5.2. Nerekurzívní zápis operace INSERT v AVL-stromu

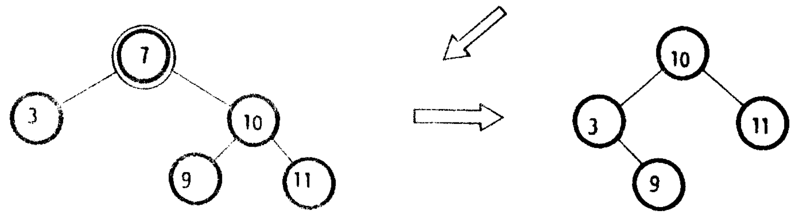
Nerekurzívní zápis operace INSERT v AVL-stromu je uveden v ukázce programu AVLSTROM v příloze A. Program byl vytvořen a odladěn v jazyce FEL-PASCAL na počítači ADT 4500.

6.3.5.3. Rekurzívní zápis operace DELETE v AVL-stromu

Zkušenost již ukázala, že operace DELETE je vždy složitější než operace INSERT, a platí to i pro AVL-strom. Algoritmy pro znovuustavení vyvážení - jednoduché a dvojité rotace - zůstávají stejné jako u operace INSERT. Na obr. 6.17. je ukázáno jak bude vypadat strom po postupném vyřazování uzlů 4,8,6,5,2,1 a 7 při použití operace DELETE, která znovuustavuje vyváženost stromu.



Obr. 6.17. Operace DELETE v AVL-stromu



Obr. 6.17. Operace DELETE v AVL-stromu
(pokračování)

Situace, kdy se vyřazuje list nebo uzel, který má jen jeden podstrom, je poměrně jednoduchá. Má-li vyřazovaný uzel oba podstromy, pak se nahradí nejpravějším uzlem levého podstromu. Podobně jako v rekurzívní operaci INSERT i zde bude parametr h nést informaci o tom, že výška stromu se snížila (h=true). Znovuustavení vyvážení se provede pouze tehdy, je-li h=true. Na hodnotu true se h nastavuje po vyřazení uzlu, nebo jestliže proces vyvažování sám redukuje výšku podstromu. V programu, který bude uveden (podle [2]) jsou zavedeny dvě symetrické vyvažovací procedury. Procedura BALANC1 se použije redukuje-li se levý a procedura BALANC2 redukuje-li se pravý podstrom.

```
procedure DELETE (var KOR:TYPUKUZEL; K:TYPKLIC; var h:Boolean); { při vyvolání
var POMUK:TYPUKUZEL;                                               h=false }
procedure BALANC1 (var KOR:TYPUKUZEL; var h:Boolean);
 {Tělo procedury bude uvedeno později. Procedura ustavuje vyvážení stromu
  redukuje-li se levý podstrom }
procedure BALANC2 (var KOR:TYPUKUZEL; var h:Boolean);
 {Tělo procedury bude uvedeno později. Procedura ustavuje vyvážení stromu redu-
  kuje-li se pravý podstrom}
procedure DEL (var UKUZ:TYPUKUZEL; var h:Boolean);
   begin  { h=false }
      if UKUZ↑.PRAVY≠nil
         then begin DEL(UKUZ↑.PRAVY,h);
                    if h then BALANC2 (UKUZ,h)
              end
         else
              begin POMUK↑.KLIC:=UKUZ↑.KLIC;
                    POMUK↑.DATA:=UKUZ↑.DATA;     POMUK:=UKUZ;
                    UKUZ:=UKUZ↑.LEVY;
                    h:=true
              end
   end;
   {Konec deklarační části procedury DELETE }
   begin {Začátek těla procedury DELETE }
      if KOR=nil
         then {Kořen není prvkem stromu; operace INSERT nezmění strom }
              h:=false
         else
           if K < KOR↑.KLIC
              then begin DELETE (KOR↑.LEVY,K,h);
                         if h then BALANC1 (KOR,h)
                   end
```

```
                    else if K > KOR↑.KLIC
                            then begin DELETE (KOR↑.PRAVY,K,h);
                                       if h then BALANC2 (KOR,h)
                                 end
                            else begin {zruší se KOR↑}
                                       POMUK:=KOR;
                                       if POMUK↑.PRAVY=nil
                                          then begin KOR:=POMUK↑.LEVY;
                                                     h:=true
                                               end
                                          else if POMUK↑.LEVY=nil
                                                  then begin KOR:=POMUK↑.PRAVY;
                                                             h:=true
                                                       end
                                                  else begin DEL(POMUK↑.LEVY,h);
                                                             if h then BALANC1
                                                                       (KOR,h)
                                                       end;
                                       dispose (POMUK)
                                 end {rušení KOR}
   end  {procedury DELETE}
```

Procedura BALANC1 má následující tvar :

```
procedure BALANC1 (var KOR:TYPUKUZEL; var h:Boolean);
   var p1, p2:TYPUKUZEL; b1,b2:-1..1;
   begin {h=true, levý podstrom se stal těžší}
      case KOR↑.BALANC of
        -1: KOR↑.BALANC:=Ø;
         Ø: begin
                 KOR↑.BALANC:=1; h:=false
            end;
         1: begin {vyvažování uzlu}
                 p1:=KOR↑.PRAVY;b1:=p1↑.BALANC;
                 if b1≥Ø then begin {jednoduchá rotace RR}
                                 KOR↑.PRAVY:=p1↑.LEVY;p1↑.LEVY:=KOR;
                                 if b1=Ø
                                    then begin KOR↑.BALANC:=1;
                                               p1↑.BALANC:=-1;h:=false
                                         end
                                    else begin KOR↑.BALANC:=Ø;p1↑.BALANC:=Ø
                                         end;
                                 KOR:=p1
                              end
                         else {dvojitá rotace RL}
                            begin p2:=p1↑.LEVY;b2:=p2↑.BALANC;
                                  p1↑.LEVY:=p2↑.PRAVY;p2↑.PRAVY:=p1;
                                  KOR↑.PRAVY:=p2↑.LEVY;p2↑.LEVY:=KOR;
                                  if b2=1 then KOR↑.BALANC:=-1
                                          else KOR↑.BALANC:=Ø;
                                  if b2=-1 then p1↑.BALANC:=1
                                           else p1↑.BALANC:=Ø;
```

```
                    KOR:=p2;p2↑.BALANC:=Ø
              end
         end {alternativy case 1:}
     end {příkazu case}
  end {procedury BALANC1}
```

Procedura BALANC2 je naprosto stranově symetrická a získáme ji jednoduchou záměnou těchto prvků programu :

místo	PRAVY	uvedeme	LEVY
	LEVY		PRAVY
	RR		LL
	RL		LR
	-1		1
	1		-1
	$\geqslant$		$\leqslant$

6.4. Tabulky s rozptýlenými položkami

6.4.1. Tabulky s přímým přístupem

Je-li známa množina všech klíčů $K=\{K_1, \ldots, K_n\}$, které se budou vkládat do vyhledávací tabulky, a je-li možné nalézt jedno-jednoznačnou mapovací funkci $f(K_i)=i$ $(i=1,2,\ldots,n)$ pro všechny prvky množiny K, je možné vytvořit tabulku s přímým přístupem. Tuto tabulku tvoří pole, v němž položka s klíčem K_i bude uložena na indexu i daného pole.

Nechť jsou dány typy :

```
TYPPOLOZKY=record
              DATA:TYPDATA;
              KLIC:TYPKLIC;
              OBSAZEN:Boolean;
           end;
TYPTABULKY=array[1..n] of TYPPOLOZKY
```

a proměnné

```
TAB: TYPTABULKY; K:TYPKLIC;DATAPOLOZKY:TYPDATA;
```

Pak operace INITTAB ustaví složku OBSAZEN u všech položek pole na hodnotu false. Algoritmus operace INSERT bude mít jednoduchý tvar :

```
procedure INSERTTAB(var TAB:TYPTAB;K:TYPKLIC;DATAPOLOZKY:TYPDATA);
begin    TAB[f(k)].DATA:=DATAPOLOZKY;
         TAB[f(k)].KLIC:=K;  {Tato operace není při jedno-jednoznačnosti
                              funkce f nutná!}
         TAB[f(k)].OBSAZEN:=true
end
```

Operace SEARCHTAB ustaví svou hodnotu na základě složky OBSAZEN :

```
function SEARCHTAB(TAB:TYPTAB;K:TYPKLIC):Boolean
begin SEARCHTAB:=TAB[f(K)].OBSAZEN end
```

a operace DELETETAB vyřadí prvek s klíčem K příkazem :

```
procedure DELETETAB (var TAB:TYPTAB;K:TYPKLIC);
   begin TAB[f(K)].OBSAZEN:=false end.
```

Obtíž s využitím jinak vysoce účinné tabulky s přímým přístupem spočívá v obtížném nalezení vhodné mapovací funkce f. V praxi se tato potíž někdy obchází používáním numerických klíčů pro identifikaci položek. Takové klíče dobře známe pod názvy "pořadové" nebo "evidenční číslo". V řadě případů je však tento manévr neúčinný, protože je nutné pracovat s textovým (nejčastěji sugestivním) tvarem klíče. Typickým příkladem takového případu je manipulace překladače s identifikátory, které tvoří uživatel programovacího jazyka při tvorbě svých programů.

6.4.2. Mapovací funkce

Jak nalézt vhodnou mapovací funkci pro danou množinu klíčů? Pro 31 různých prvků, které se mají zobrazit do 41 prvkové množiny existuje 41^{31} ($\approx 10^{50}$) možných mapovacích funkcí. Přitom pouze (41!/31!) z nich dává odlišné hodnoty pro různé klíče (t.zn. že funkce je jedno-jednoznačná). T.zn., že poměr vhodných funkcí ku všem možným funkcím je asi $1:10^7$. Jedno-jednoznačné funkce jsou tedy velmi řídkým jevem. Dokládá to i paradox "společných narozenin", který říká, že je dobrá naděje, že mezi 23 osobami, které se sejdou ve společnosti, se najdou dvě osoby, které mají narozeniny ve stejný den. Jinými slovy: najdeme-li náhodně funkci, která mapuje 23 klíčů do tabulky o 365 prvcích, je pravděpodobnost, že se žádné dva klíče nemapují do jednoho místa rovna pouze 0.4927. Je zřejmé, že tato pravděpodobnost se bude zvyšovat se zvětšováním počtu prvků množiny, do které se mapuje (t.zn. počtu prvků pole tabulky).

V dalších úvahách budeme hledat takovou mapovací funkci, která klíče z dané předpokládané množiny klíčů "rozptýlí" v dané tabulce, aniž budeme trvat na jedno-jednoznačnosti funkce. Klíče, které mají stejnou hodnotu mapovací funkce budeme nazývat <u>synonyma</u>. Dojde-li k pokusu umístit novou položku na místo, které je již obsazeno, budeme situaci nazývat <u>kolize</u>. Mapovací funkci používané k rozptýlení položek v tabulce budeme říkat <u>rozptylovací funkce</u> (angl. hash-function, říká se jí také hashovací či hašovací funkce).

Předpokládejme, že máme k dispozici celočíselnou funkci Num (K), která z libovolného typu klíče získá celé <u>kladné číslo</u>. Je-li klíč textovým řetězcem, může se využít jeho binárního obrazu (v Pascalu budeme pracovat např. s funkcí ord). Právě volba funkce Num nejvíce ovlivní počet synonym a množství kolizí v tabulce. Pro volbu této funkce však nelze stanovit obecně platná pravidla. Volba musí vycházet z konkrétních vlastností množiny klíčů. Bude-li funkce Num odvozovat svou hodnotu např. z prvního znaku textového klíče, pak všechny identifikátory (použité jako klíče) se stejným písmenem na začátku, budou synonyma. Protože všechna písmena nemají rovnoměrný výskyt jako první znaky identifikátorů, bude u některých znaků docházet k častějším kolizím.

Na dobrou rozptylovací funkci se kladou dva základní požadavky :

a) výpočet rozptylovací funkce musí být dostatečně rychlý

b) rozptylovací funkce má vytvářet co nejméně kolizí

Je zřejmé, že tyto požadavky jsou většinou protichůdné.

Nechť rozptylovací funkce R(K) = h(Num(K)), kde funkce h zajistí transformaci (mapování) libovolného celého kladného čísla (získaného funkcí Num) do intervalu daného rozsahem indexu pole, kterým implementujeme tabulku. Tímto intervalem bude nejčastěji Ø..MAX nebo 1..MAX.

Poměrně úspěšné rozptylovací funkce h jsou založeny na vlastnostech celočíselného dělení a to především operace modulo. Např.

$$h(K) = K \text{ mod } (MAX+1)$$

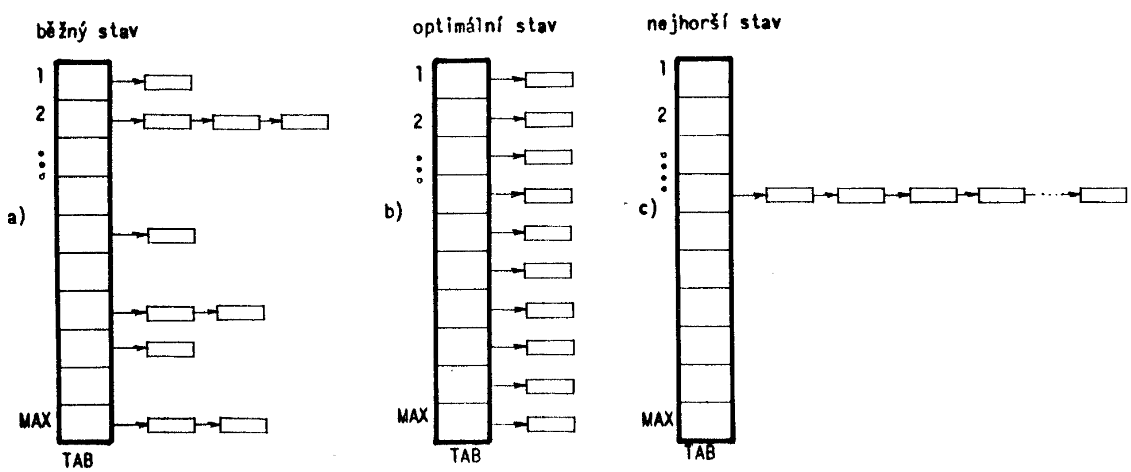
získá hodnoty z intervalu 0..MAX, a funkce

$$h(K)=K \text{ mod } MAX+1 \text{ z intervalu } 1..MAX$$

Úspěšnost zvolené funkce však může záviset také na volbě velikosti tabulky (MAX). Pro funkci h(K)=K mod MAX jsou některé hodnoty MAX nevhodné. Jsou to např. hodnoty MAX = $r^{l} \pm c$, kde l a c jsou malá celá čísla a r je řád použitého alfanumerického kódu (64, 128, 256 ap.).

6.4.3. Princip tabulek s rozptýlenými položkami

Princip vyhledávání v tabulkách s rozptýlenými položkami (dále jen TRP) je velmi podobný principu vyhledávání v indexsekvenčním souboru. Pomocí rozptylovací funkce se získá index pole, na nějž se uloží (od něj se vyhledává) položka s daným klíčem. Obsahuje-li tabulka synonyma vzhledem k danému indexu (více různých klíčů mělo shodnou hodnotu rozptylovací), pak na daném indexu začíná lineární seznam synonym, v němž se vyhledává položka s daným klíčem. Vyhledávání v TRP bude účinné tehdy, jestliže počet seznamů synonym bude co největší a jejich délka bude co nejkratší. Situaci znázorňuje obr. 6.18.



Obr. 6.18. Princip tabulek s rozptýlenými hesly

Na obr. 6.18. je znázorněn "běžný stav" (ad a.), v němž nejhorší případ vyhledávání je ekvivalentní průchodu lineárním seznamem synonym o třech prvcích. Nejhorší případ TRP, způsobený nevhodnou rozptylovací funkcí, je uveden jako případ c. V něm je TRP degradováno na lineární seznam, obsahující všechny prvky tabulky jako synonyma.

Podle způsobu, jakým se realizuje lineární seznam synonym, rozdělují se nejznámější metody implementace TRP do dvou skupin :

a) Tabulky s explicitně zřetězenými synonymy
b) Tabulky s implicitně zřetězenými synonymy (této skupině se také říká "tabulky s otevřeným adresováním položek").

Explicitním zřetězením se rozumí, že každý prvek seznamu obsahuje ukazatel na následníka v seznamu. Implicitním zřetězením se rozumí, že z adresy (indexu) každého prvku seznamu se může určit adresa (index) následujícího prvku.

6.4.4. TRP s explicitně zřetězenými synonymy

Základem TRP je pole. Na i-tém prvku pole je začátek lineárního seznamu všech položek se synonymními klíči, pro které platí R(K)=i. TRP s explicitně zřetězenými seznamy můžeme principiálně rozdělit podle způsobu, kterým se přiděluje paměťový prostor položkám seznamu.

a) Pole obsahuje pouze ukazatele na jednotlivé seznamy synonym (viz obr. 6.18.).
b) Pole obsahuje položky seznamu. Synonyma se ukládají do "oblasti přeplnění". Kolidující položka v tabulce je zřetězena se seznamem synonym umístěným v oblasti přeplnění.

Oblast přeplnění může být organizována různými způsoby :

A) Oblast přeplnění budou vytvářet paměťová místa získaná mechanismem dynamického přidělování paměti (new).
B) Oblast přeplnění tvoří zvláštní pole (v některých případech to může být "horní" část pole, jehož "dolní" část je použita jako základní pole TRP).
C) Oblast přeplnění se překrývá se základním polem TRP.

Jako první si ukažme implementaci typu a), v níž všechny prvky tabulky jsou obsaženy v "oblasti přeplnění" realizované podle způsobu A). (V tom případě nemá vlastně smysl hovořit o oblasti přeplnění, protože jsou v ní uloženy všechny prvky).

```
Nechť jsou dány typy :
TYPUK=↑TYPPOLOZKY
TYPPOLOZKY=record
              DATA:TYPDATA;
              KLIC:TYPKLIC;
              DALSI:TYPUK
           end;
TYPTAB=array [1..MAX] of TYPUK;
```

Pak operace INIT nastaví všechny ukazatele tabulky TAB na hodnotu nil.

```
procedure INIT (var TAB:TYPTAB);
   var I:POSINT;
   begin for I:=1 to MAX do TAB[I]:=nil end;
```

Operace SEARCH bude mít tvar Booleovské funkce :

```
function SEARCH (TAB:TYPTAB;K:TYPKLIC):Boolean;
var I:POSINT;
   POMUK:TYPUK;
   begin I:=R(K);
         POMUK:=TAB[I];
         SEARCH:=false;
         if POMUK≠nil
            then begin while (POMUK↑.DALSI≠nil)and(POMUK↑.KLIC≠K) do
                                POMUK:=POMUK↑.DALSI;
                       SEARCH:=POMUK↑.KLIC=K
                 end
   end {funkce SEARCH}
```

Operace INSERT buď vloží do tabulky novou položku (bylo-li vyhledání daného klíče neúspěšné) nebo přepíše (aktualizuje) datovou složku položky s vyhledaným klíčem. Uvedený algoritmus vkládá nový prvek na začátek seznamu synonym.

```
procedure INSERT (var TAB:TYPTAB;K:TYPKLIC;DATAPOLOZKY:TYPDATA);
   var I:POSINT;
      POMUK:TYPUK;
      NASEL:Boolean;
      begin I:=R(K);
            POMUK:=TAB[I];
            NASEL:=false;
            if POMUK≠nil
               then { seznam synonym je neprázdný; prohledá se.}
                  begin while (POMUK↑.DALSI≠nil) and (POMUK↑.KLIC≠K) do
                              POMUK:=POMUK↑.DALSI;
                        NASEL:=POMUK↑.KLIC=K
                  end;
            if NASEL then POMUK↑.DATA:=DATAPOLOZKY  { Přepis datové složky }
                  else begin { Nová položka se zařadí na začátek seznamu }
                              new (POMUK);
                              POMUK↑.DALSI:=TAB[I];
                              TAB[I]:=POMUK;
                              POMUK↑.KLIC:=K;
                              POMUK↑.DATA:=DATAPOLOZKY
                       end
      end { procedury INSERT }
```

Operace DELETE je podobná operaci DELETE nad lineárním zřetězeným seznamem. Její implementace je ponechána čtenáři.

V [1] je uveden algoritmus, v němž se oblast přeplnění překrývá se základním polem. Není-li při vkládání nového prvku volné místo na indexu získaném rozptylovací funkcí R, vyhledá se v poli tabulky první volný prvek (pro usnadnění vyhledávání se používá pomocný ukazatel ukazující na poslední volný prvek), naplní se daty a připojí se k seznamu synonym. Parametr ERROR oznamuje, že novou položku nelze vložit, protože tabulka je plná.

Nechť jsou dány typy :

```
TYPPOLOŽKY=record
              KLIC:TYPKLIC;
              DATA:TYPDATA;
              VOLNY:Boolean;
              DALSI:Ø..MAX
           end;
TYPTAB=array[1..MAX] of TYPPOLOZKY;
```

Operace inicializace bude ustavovat pole volných prvků a nastavovat pomocný ukazatel RR

```
procedure INIT (var TAB:TYPTAB; var RR:POSINT);
var I:POSINT;
begin for I:=1 to MAX do TAB[I].VOLNY:=true;
   RR:=MAX
end {Procedury INIT}
```

Protože mechanismus vyhledávání je obsažen v operaci INSERT, nebudeme uvádět samostatnou operaci SEARCH. Operace INSERT podle uvedeného Knuthova algoritmu bude mít tvar :

```
procedure INSERT (var TAB:TYPTAB;K:TYPKLIC;DATAPOLOZKY:TYPDATA;var RR:POSINT;
                  var ERROR:Boolean);
var I ,J:integer;
    JESTE,NASEL:Boolean;
    begin I:=R(K); {Rozptylovací funkce poskytuje hodnoty z intervalu 1..MAX}
          ERROR:=false;NASEL:=false; {Inicializace Booleovských proměnných}
          if not TAB[I].VOLNY
             then {prvek není volný, hledej v lineárním seznamu}
               begin JESTE:=TAB[I].KLIC≠K; {Inicializace řídicí proměnné cyklu}
                     while JESTE do
                       begin J:=TAB[I].DALSI;    {Index následníka}
                             if J=∅ then JESTE:= false   {konec seznamu}
                                    else begin JESTE:=TAB[I].KLIC≠K;I:=J end;
                       end;
                     if I≠∅ then begin NASEL:=true;
                                       TAB[I].DATA:=DATAPOLOZKY {Přepis při
                                                                  nalezení}
                                 end;
                     if not NASEL
                        then {Vyhledej místo pro vkládanou položku}
                           begin
                             while(RR≠∅) and (not TAB[RR].VOLNY) do RR:=RR-1;
                             ERROR:=RR=∅;
                             if not ERROR then begin TAB[I].DALSI:=RR;
                                                {připojení k seznamu}
                                                     I:=RR{příprava pro naplnění}
                                               end
                           end
               end;
          if (not NASEL) and (not ERROR) then {naplnění složek vkládané položky}
             begin TAB[I].KLIC:=K;
                   TAB[I].DALSI:=∅;
                   TAB[I].DATA:=DATAPOLOZKY;
                   TAB[I].VOLNY:=false
             end
    end {procedury INSERT}
```

6.4.5. TRP s implicitně zřetězenými synonymy

TRP s implicitně zřetězenými synonymy jsou implementovány jedním polem, které plní jak funkci základního pole (do něhož míří index získaný rozptylovací funkcí), tak oblasti přeplnění. Index následníka v seznamu synonym je dán součtem indexu předchůdce a přírůstku INC. Podle vlastnosti přírůstku dělíme metody s implicitně zřetězenými synonymy na :

a) lineární vyhledávání (INC je konstantní; nejčastěji INC:=1)
b) kvadratické vyhledávání (INC lineárně vzrůstá; nejčastěji INC:=INC+1)
c) metoda dvou rozptylovacích funkcí (INC je konstantní, ale získá se druhou rozptylovací funkcí INC:=R_2(K)).

6.4.5.1. TRP s lineárním vyhledáváním

Vyhledávání v této tabulce postupuje podle pravidel :

a) Rozptylovací funkce určí index prvního přístupu do pole tabulky

b) Od této pozice se zahájí vyhledávání v sekvenčně umístěném seznamu v poli. Vyhledávání končí úspěšně při nalezení položky s vyhledávaným klíčem nebo neúspěšně, dojde-li se k prvnímu neobsazenému prvku pole. S polem se pracuje jako s kruhovým seznamem.

Pro zajištění konečnosti neúspěšného vyhledávání se musí zachovat alespoň jedna zarážka ve formě neobsazeného prvku pole. Nejčastěji se to dělá pomocnou proměnnou POCET, která udává počet prvků v tabulce. Je-li POCET právě o 1 menší než je maximální kapacita, je tabulka zaplněna. (Jinou možností konečnosti vyhledávání je test na shodu s výchozím indexem). Na obr. 6.19. je uveden příklad, který ukazuje, že prohledávaný seznam nemusí být složen vždy jen ze synonym. Nechť klíče A1, A2, A3 a A4 jsou synonyma (t.zn. že R(A1) = R(A2) = R(A3) = R(A4)= = I_1) a klíče B1, B2 a B3 jsou také synonyma (t.zn. R(B1) = R(B2) = R(B3) = I_2). Pak budou-li se do TRP vkládat klíče v pořadí A1, B1, A2, A3, B2, A4, B3, bude mít TRP tvar podle obr. 6.19.

	I_1		I_2							
volný	A1	A2	B1	A3	B2	A4	B3	volný	volný	volný

Obr. 6.19. TRP s lineárním vyhledáváním

Nechť jsou dány typy :

```
TYPPOLOZKY=record
                KLIC:TYPKLIC;
                DATA:TYPDATA;
                OBSAZENY:Boolean
           end;
TYPTAB=array[Ø..MAX] of TYPPOLOZKY;
```

Pak operace INIT a INSERT nad TRP s lineárním vyhledáváním budou mít tvar :

```
procedure INIT (var TAB:TYPTAB; var POCET:POSINT);
var I:POSINT;
  begin for I:=Ø to MAX do TAB[I].OBSAZENY:=false;
        POCET:=Ø
  end

procedure INSERT (var TAB:TYPTAB;K:TYPKLIC;DATAPOLOZKY:TYPDATA;var POCET:
var INC,I:POSINT; B:Boolean;                                       POSINT);
 begin
      I:=R(K); {R(K) je v intervalu Ø..MAX}
      INC:=1; B:= true;
      while TAB[I].OBSAZENY and    B do begin
       B:=TAB[I].KLIC≠K; if not B then I:=(I+INC)mod(MAX+1) end;
      if TAB[I].KLIC=K
         then TAB[I].DATA:=DATAPOLOZKY {Našel,přepisuje}
         else {vkládá}
           begin
```

```
                  TAB[I].KLIC:=K;
                  TAB[I].DATA:=DATAPOLOZKY;
                  TAB[I].OBSAZENY:=true;
                  POCET:=POCET+1
                end
       end {procedury INSERT}
```

6.4.5.2. TRP s kvadratickým vyhledáváním

Obr. 6.18. ukazuje případ, který je častý u TRP s lineárním vyhledáváním. Projevuje se vytvářením shluků obsazených prvků pole, vznikajících nejčastěji tehdy, je-li přírůstek INC=1 a jsou-li indexy několika skupin synonym dosti blízké. Situaci lze zlepšit změnou hodnoty INC na hodnotu větší než 1. Hodnota INC však nesmí být dělitelem délky základního pole (aby se zajistil průchod všemi prvk pole).

Jiný významný způsob zabránění vzniku shluků spočívá v kvadratickém vyhledávání, to znamená, že hodnoty indexu po sobě jdoucích prvků vytvářejí kvadratickou funkci. Je-li I_o=R(K) pak $I_\ell=(I_o+\ell^2)$ mod MAX. Přitom platí $I_{\ell+1} = I_\ell + INC_\ell$ a $INC_{\ell+1} = INC_\ell+2$. Je-li I_o=1, pak se při průchodu postupně projde indexy 1,2,5, 10,17,26,37,... Je-li MAX prvočíslo, pak při neúspěšném vyhledávání se v nejhorším případě projde polovinou prvků pole a průchod končí při INC=MAX.

V [6] je popsána a odvozena metoda, která postupně prochází indexy

I_o=R(K)

$I=(I_o+0.5\ell+0.5\ell^2)$ mod MAX

přičemž platí $I_{\ell+1} = I_\ell + INC_\ell$

$INC_{\ell+1} = INC_\ell + 1$

Je-li I_o = 1 pak se při průchodu postupně projde indexy 1,2,4,7,11,16,22,... Pro tuto metodu musí mít MAX hodnotu prvočísla ve tvaru 4n+3 (např. 991).

Nechť je dán typ

```
TYPTAB=array [1..PRVOCISLO] of TYPPOLOZKY
```

kde PRVOCISLO=4K+3 pro celé kladné K (např. 991) a TYPPOLOZKY je shodný s typem z předchozího odstavce. Pak operace INIT a INSERT budou mít tvar :

```
procedure INIT (var TAB:TYPTAB);
var I: integer;
begin for I:=1 to PRVOCISLO do TAB[I].OBSAZENY:=false
end;
procedure INSERT (var TAB:TYPTAB;K:TYPKLIC;DATAPOLOZKY:TYPDATA;var ERROR:
var I,INC:POSINT;                                                  Boolean);
    KONEC,VLOZIL:Boolean;
begin ERROR:=false;
      INC:=1; {*}
      I:=R(K); {R(K) je z intervalu 1..PRVOCISLO}
      VLOZIL:=false;
      repeat KONEC:=false;
             if not TAB[I].OBSAZENY
                then begin {Našel volný prvek, vkládá a končí}
```

```
                    TAB[I].KLIC:=K;
                    TAB[I].DATA:=DATAPOLOZKY;
                    TAB[I].OBSAZENY:=true;
                    KONEC:=true;
                    VLOZIL:=true
                 end
            else {Prvek neni volný}
            begin if TAB[I].KLIC=K
                  then {Našel shodu, přepisuje}
                     begin TAB[I].DATA:=DATAPOLOZKY;
                           KONEC:=true
                     end
                  else {Připrav index dalšího prvku}
                     begin
                         I:=I+INC; {*}
                         if I > PRVOCISLO then I:=I-PRVOCISLO;
                         INC:=INC+1; {*}
                         if INC >= (PRVOCISLO div 2) {*}
                           then begin KONEC:=true
                                      ERROR:=true
                                end
                     end
            end
       until KONEC
   end {procedury INSERT}
```

Skutečnost, že algoritmus prochází v nejhorším případě jen polovinu prvků tabulky neubírá metodě na účinnosti. Přesto lze jednoduchou úpravou algoritmus zlepšit tak, aby byla pro každý klíč dostupná celá tabulka. V [6] je tato metoda nazvána "Full Table Quadratic Searching" - tedy kvadratické vyhledávání v plné TRP. Úprava spočívá ve třech změnách na místech v programu označených {*}

```
Místo INC:=1        se uvede  INC:=-PRVOCISLO
 -"-  I:=I+INC       -"-      I:=I+abs(INC)
 -"-  if INC >=(PRVOCISLO div 2)
                     se uvede if INC >= PRVOCISLO
 -"- INC:=INC+1    se uvede   INC:=INC+2
```

6.4.5.3. TRP s dvojí rozptylovací funkcí

Odstranění shluků v TRP se může dosáhnout také s přírůstkem INC, který je sice konstantní, ale jeho hodnota se získá druhou rozptylovací funkcí z klíče K. Zatímco první rozptylovací funkce R(K) dává hodnotu z intervalu Ø..MAX, druhá rozptylovací funkce Q(K) musí mít hodnotu z intervalu 1..MAX takovou, která není dělitelem čísla MAX+1. (Bude-li MAX+1 prvočíslo, pak to může být každé číslo z daného intervalu; bude-li $(MAX+1)=2^m$, pak to může být každé liché číslo z intervalu $1..(2^m-1)$).

Nechť je dán typ

```
TYPTAB=array [Ø..MAX] of TYPPOLOZKY; {MAX+1 je prvočíslo}
```

Pak operace INIT a INSERT budou mít následující tvar :

```
procedure INIT (var TAB:TYPTAB; var POCET:POSINT);
    var I:POSINT;
    begin for I:=Ø to MAX do TAB[I].OBSAZENY:=false;
          POCET:=Ø
    end;

    procedure INSERT (var TAB:TYPTAB;K:TYPKLIC;DATAPOLOZKY:TYPDATA;
                      var POCET:POSINT; var ERROR:Boolean);
    var I, INC:POSINT;
        NASEL,KONEC:Boolean;
    begin
        ERROR:=false;
        NASEL:=false;
        I:=R(K); {Funkce dává hodnotu z intervalu Ø..MAX}
        if TAB[I].OBSAZENY
           then {prvek není volný}
                if TAB[I].KLIC=K
                   then NASEL:=true {Našel shodu}
                   else begin {hledej v seznamu}
                          INC:=Q(K); {funkce dává hodnotu z intervalu
                                      1..MAX}
                          repeat KONEC:=false;
                                 I:=I+INC;
                                 if I > MAX then I:=I-MAX-1;
                                 if not TAB[I].OBSAZEN
                                    then KONEC:=true {Našel volný,končí}
                                    else if TAB[I].KLIC=K
                                            then {Našel shodu, končí}
                                              begin KONEC:=true;
                                                    NASEL:=true
                                              end
                         until KONEC;
                        end;
        if NASEL then TAB[I].DATA:=DATAPOLOZKY {Přepisuje nalezenou položku}
                 else if POCET = MAX then ERROR:=true
                                     else {vkládá novou položku}
                                       begin POCET:=POCET+1
                                             TAB[I].DATA:=DATAPOLOZKY;
                                             TAB[I].KLIC:=K;
                                             TAB[I].OBSAZENY:=true
                                       end
end {procedury INSERT}
```

6.4.5.4. Brentova varianta

Brentova varianta (viz [1]) vychází z metody dvou rozptylovacích funkcí a předpokládá, že úspěšné vyhledání je častější než neúspěšné vyhledání s následným vkládáním nového prvku. Algoritmus vkládání je složitější, ale má za důsledek zkrácení úspěšného vyhledávání. Metoda používá dvou rozptylovacích funkcí R a Q - stejných jako v předcházející metodě. Stejně jako v předcházející metodě se při vyhledávání prochází indexy $I_s=(r+s*q)$ mod (MAX+1) kde $r=R(K)$ a $q=Q(K)$. Pro $S \geq \emptyset$ se postupně prohlíží TAB$[I_0]$, TAB$[I_1]$, ..., TAB$[I_s]$. Platí-li TAB$[I_s]$.KLIC=K, pak došlo k úspěšnému nalezení po $\underline{p(K)=S+1}$ porovnáních. Platí-li TAB$[I_s]$.OBSAZENY = false, kde $S > \emptyset$, pak to znamená, že položka s klíčem K není v tabulce obsažena. Než vložíme do tabulky novou položku, zjistíme, zda by se nová položka mohla vložit jinam, než na první volné místo. Úvaha vede k mechanismu, v němž se na první volné místo může vložit položka, která již byla dříve do tabulky vložena, zatímco nová položka se vloží na její původní místo. Jak se hledá taková vhodná položka ?

Definujme

$$q_m = Q(\text{TAB}[I_m].\text{KLIC}) \qquad \text{pro } m \geq \emptyset$$

$$\text{a} \quad I_{m,n} = (I_m + nq_m) \ \underline{\text{mod}} \ (\text{MAX}+1) \ \text{pro } n \geq 1$$

Mezi všemi m a n pro která platí, že TAB$[I_{m,n}]$.OBSAZENY=false hledáme takové m a n, pro které platí, že součet m+n je minimální. Je-li takových dvojic více, volíme tu, u níž je nejmenší index $\underline{m}$.

V případě neúspěšného hledání jsme skončili nalezením takového S, pro které platilo TAB$[I_S]$.OBSAZENY=false. Pro vložení nové položky jsou dvě možnosti :

a) Nenalezli jsme žádnou položku TAB$[I_{m,n}]$.OBSAZENY=false, pro níž by platilo $m+n < s$. Pak vložíme novou položku na index I_s (stejně jako to dělá metoda dvou rozptylovacích funkcí).

b) Nalezli jsme položku TAB$[I_{m,n}]$.OBSAZENY=false, pro $m+n < S$.
Pak provedeme následující přesun :

TAB$[I_{m,n}]$:=TAB$[I_m]$
TAB$[I_m]$:= nová položka

Pozn.: lze si všimnout, že pro známé S se pro zjištění vhodného volného místa nebude dělat více než $\frac{S}{2}(S-1)$ pokusů. Jsou to indexy :

$$I_{\emptyset,1}, I_{\emptyset,2}, \ldots, I_{\emptyset,S-1}, I_{1,1}, I_{1,2}, I_{1,S-2}, \ldots, I_{S-2,1}$$

Příklad :

a) Neúspěšné vyhledávání skončilo nalezením volného místa při S=3
(Vertikálně - neúspěšné vyhledávání. Horizontálně - hledání vhodného $I_{m,n}$).

TAB$[I_\emptyset]$ =obsazený → TAB$[I_{\emptyset,1}]$= obsaz. → TAB$[I_{\emptyset,2}]$ = obsaz.
TAB$[I_1]$ =obsaz. → TAB$[I_{1,1}]$= obsaz.
TAB$[I_2]$ =obsaz.
TAB$[I_3]$ =volný

Protože se nenašel volný prvek TAB$[I_{m,n}]$, pro který platí $m+n < 3$ <u>vložíme novou položku na index I_3</u>, tedy TAB$[I_3]$:=nová položka

b) Neúspěšné vyhledávání skončilo nalezením volného místa při S=3
TAB$[I_\emptyset]$ =obsaz. → TAB$[I_{\emptyset,1}]$=obsaz. → <u>TAB$[I_{\emptyset,2}]$ =volný</u>
TAB$[I_1]$ =obsaz.
TAB$[I_2]$ =obsaz.
TAB$[I_3]$ =volný

Protože TAB$[I_{\emptyset,2}]$ = volný a $m=\emptyset$, $n=2$, $m+n=2$. Protože $m+n < S$ provedeme přesun

$TAB[I_{\emptyset,2}] := TAB[I_{\emptyset}]$

$TAB[I_{\emptyset}]$:= nová položka

Důkaz Brentovy varianty :

Nechť $c = \sum_{i=1}^{w} p(K_i)$ je celkový počet porovnání pro vyhledání všech klíčů $K_1, \ldots, K_w$ v poli TAB. Jestliže má každá položka stejnou pravděpodobnost, že bude vyhledávána, pak c/w je průměrný počet porovnání pro vyhledání jednoho klíče. Z toho vyplývá, že cílem je co nejmenší c. Proto musíme novou položku vkládat do tabulky tak, aby výsledný přírůstek D pro určení nového c byl co nejmenší.

V případě příkladu a) bude p(K)=S+1 a tudíž D=S+1.
V případě b) bude p(K)=m+1 a $p(TAB[I_m])$ vzroste o n a tudíž D = m+n+1.

Z toho vyplývá, že druhý přírůstek bude menší, je-li m+n < S.

6.4.6. Operace DELETE v TRP

U jednotlivých metod TRP byly uvedeny většinou pouze operace INIT a INSERT. Z operace INSERT lze snadno odvodit algoritmus operací SEARCH a READ. Poněkud složitější je to s operací DELETE. V metodách s explicitním zřetězením lze vyřazovaný prvek vyloučit na principu operace DELETE v jednosměrném zřetězeném seznamu. (U Knuthovy varianty by se měl aktualizovat pomocný ukazatel R.)

U TRP s implicitním zřetězením je jedinou praktickou možností "zaslepení" vylučované položky. Nastavením složky OBSAZENY na hodnotu false by se totiž porušilo implicitní zřetězení mezi předchůdcem a následníkem vyřazovaného prvku. Zaslepení však vyžaduje indikaci zaslepeného prvku. Místo složky OBSAZENY lze definovat např. složku STAV:(VOLNY,OBSAZENY,SLEPY). Operaci INSERT lze pak upravit tak, že si při vyhledávání pamatuje index první "slepé" položky v procházeném seznamu a na tento index pak vloží novou položku v případě neúspěšného vyhledání.

6.4.7. Hodnocení vyhledávání v tabulkách s rozptýlenými položkami

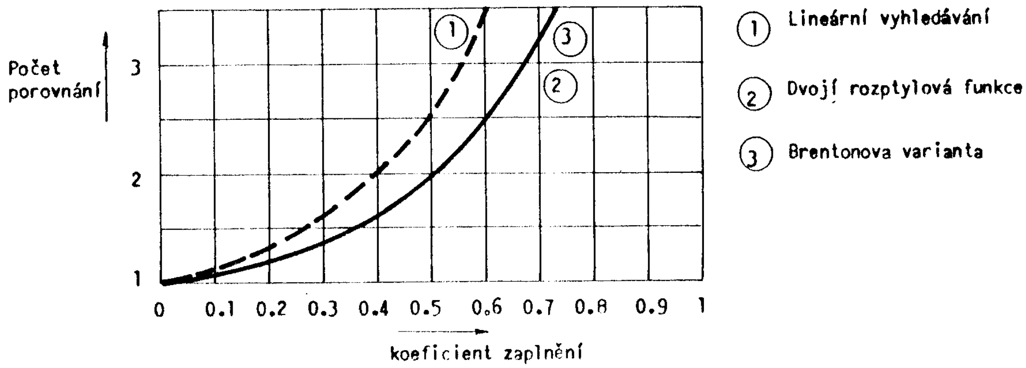
U tabulek, které jsou celé implementované polem hraje nejdůležitější roli pro hodnocení délky vyhledávání koeficient zaplnění tabulky α =POCET/MAX (poměr počtu prvků v tabulce k maximálnímu možnému počtu).

Analýza jednotlivých metod není jednoduchá a analytické vyjádření průměrné doby úspěšného a neúspěšného vyhledávání mají podobu vztahů jen zřídka použitelnýchvpraxi. Pro podrobnější informace odkazujeme čtenáře na [1] a [2]. Pro srovnání jednotlivých metod je na obr. 6.20. a 6.21. grafické znázornění závislosti počtu porovnání úspěšného a neúspěšného vyhledávání na koeficientu zaplnění.

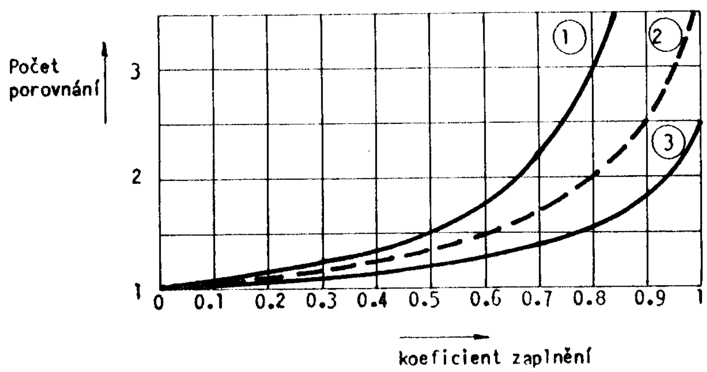
Ze srovnání některých metod lze podle [1] vyvodit tyto závěry :

Metoda lineárního vyhledávání potřebuje větší počet porovnání než ostatní metody, ale její výhodou je jednoduchost. Uvážíme-li, že při 90% zaplnění potřebuje pro úspěšné vyhledání v průměru méně než 5.5 porovnání, lze říci, že metoda je užitečná. Pro vložení nové položky po neúspěšném vyhledání však potřebuje při tomto zaplnění v průměru 50.5 porovnání.

Metody s explicitním zřetězením jsou ekonomické s ohledem na počet porovnání, ale jejich nevýhodou je to, že pro řetězící ukazatele potřebuje více paměti.



Obr. 6.20. Počet provnání při neúspěšném vyhledávání



Obr. 6.21. Počet porovnání při úspěšném vyhledávání

Jsou-li položky tabulky krátké může se ukázat, že je výhodnější větší tabulka s implicitním zřetězením než menší tabulka s explicitním zřetězením.

Z uvedených metod se jeví nejúčinnější Brentova varianta, která umožňuje úspěšné vyhledání v zaplněné tabulce s průměrem 2.5 porovnání. Neúspěšné vyhledávání s vložením nové položky je však pomalé a vyžaduje cca 0.5 N porovnání.

Ze srovnání s jinými vyhledávacími metodami vyplývá, že tabulky s rozptýlenými hesly jsou rychlejší než binární vyhledávání, zejména pro větší N. Binární vyhledávání je přitom vhodné jen pro statické tabulky. Tabulky s rozptýlenými položkami jsou rychlejší než vyhledávání ve stromech, je-li počet prvků řádově větší než 100.

Tabulky s rozptýlenými hesly však mají také své nevýhody :

a) Z neúspěšného vyhledání nelze získat žádnou dodatečnou informaci. Některé jiné metody snadno poskytnou nejbližší nižší klíč a nejbližší vyšší klíč, což je často výhodné např. pro potřeby interpolace.

b) Dimenzování polí pro tabulky s rozptýlenými hesly není vždy snadné.

c) Všechny uvedené údaje o tabulkách s rozptýlenými hesly mají statistický charakter. Nejhorší případy se však od průměrných hodnot liší významně. Proto jsou TRP málo vhodné pro některé aplikace v reálném čase, kde jsou výhodnější např. vyvážené vyhledávací stromy, které zaručují horní hranici vyhledávací doby.

6.5. L i t e r a t u r a

[1] Knuth,D. : The art of Computer programming.
Vol.3. Sorting and Searching
Addison-Wesley, 1975

[2] Wirth,N. : Algorithmus + Data Structures = Programs
Prentice - Hall INC. 1976

[3] Wiedermann,J. : Vyhladávanie
Sborník semináře SOFSEM'81, VUT UJEP Brno, VVS Bratislava

[4] Rábová,Z., Češka,M., Honzík,J., Hruška,T. : Počítače a programování
skriptum FE VUT v Brně, SNTL 1982

[5] Wiedermann,J. : Algoritmy vyhladávania
Informačné systémy 1-83, 2-83 a 3-83 VVS Bratislava

[6] Colin Day, A. : Fortran techniques with special reference to
non-numerical application. Cambridge University Press 1972

[7] Adělson-Velskij, G.M., Landis,E.M. : Doklady Akademii Nauk SSSR
No. 146, 1962