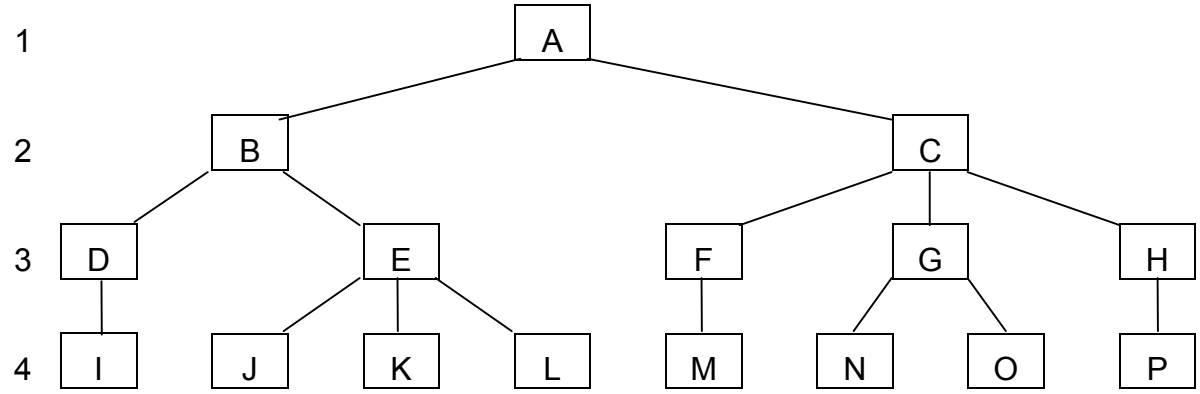
Kapitola 4: Stromy a jejich zpracování

Stromy představují velmi obecné a univerzálně použitelné datové struktury. I posloupnost lze považovat za degenerovaný strom.



Terminologie

- kořen stromu
- (vnitřní) uzly, či vrcholy
- listy
- předchůdci a následníci
- *stupeň (arita) uzlu* = počet přímých následovníků vnitřního uzlu
  stupeň stromu = maximální stupeň mezi všemi uzly stromu
- hloubka stromu
  pro binární strom: $n=2^{h+1}-1$, tedy hloubka $h=floor(log_2 n)$
- cesta ve stromu:
  délka cesty = počet hran od kořene k danému uzlu (=hloubka uzlu)
  délka vnitřní cesty = součet délek cest jednotlivých uzlů
  průměrná délka vnitřní cesty:
  $P_I = \frac{1}{n}\sum_i n_i.i_i$, kde n_i je počet vrcholů na i-té úrovni

Binární stromy

Paměťová reprezentace:

a) pomocí pole

```
type tValue = record
                  key:        tKey;
                  value:  {ostatní položky dat};
              end;
type tNode = record
                  value:  tValue;
                  left,
                  right:  integer;
              end;
type tTree = array [1..N]of tNode;
```

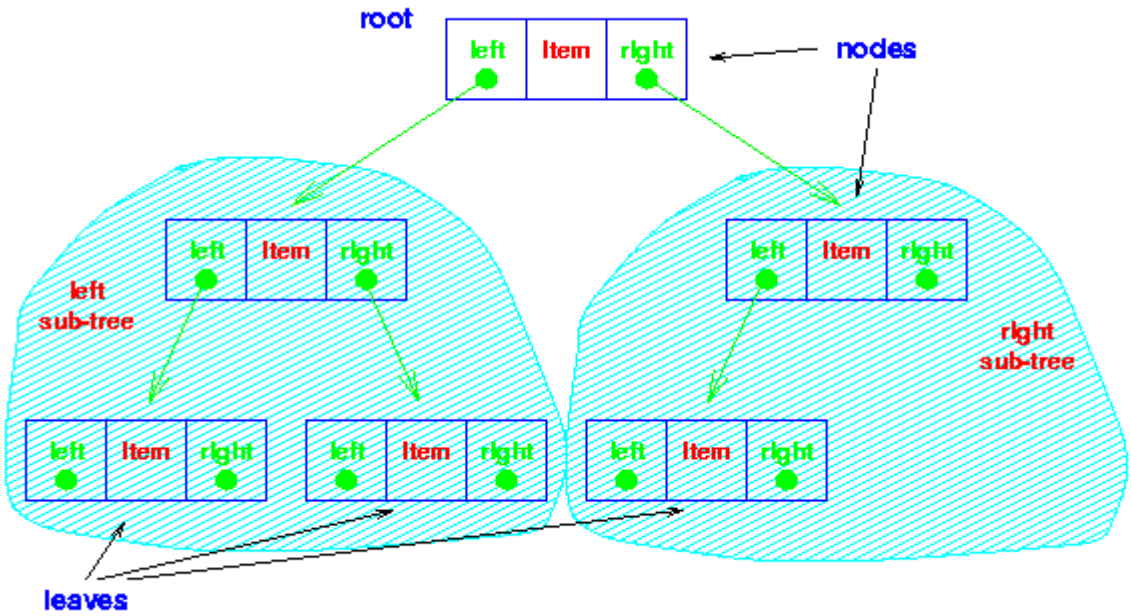
b) pomocí ukazatelů

```
/* Binary tree implementation of a collection */

struct t_node {
  void *item;
  struct t_node *left;
  struct t_node *right;
  };

typedef struct t_node *Node;

struct t_collection {
  int size;   /* Needed by FindInCollection */
  Node root;
  };
```



Dokonale vyvážený strom *je strom, pro jehož každý vrchol platí, že* ***počet vrcholů*** *v levém podstromu a pravém podstromu se liší nanejvýše o jedna.*

Mějme vstupní soubor s 21 datovými položkami, jejichž klíče jsou:

8	9	11	15	19	20	21	7	3	2	1
5	6	4	13	14	10	12	17	16	18	

Následující algoritmus vytvoří dokonale vyvážený strom o *n* vrcholech:

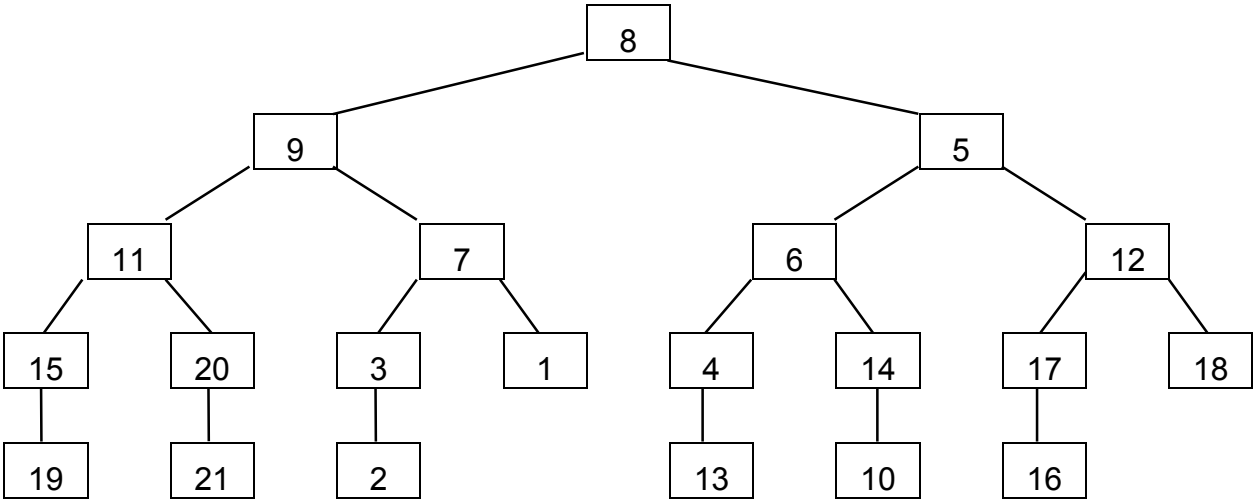
1. zvolíme jeden vrchol za kořen
2. vytvoříme levý podstrom s počtem vrcholů $n_l = n \text{ div } 2$
3. vytvoříme pravý podstrom s počtem vrcholů $n_r = n - n_l - 1$

Pravidla aplikujeme rekurzivně:

```
var root: rNode;
function Tree(n: integer) : rNode;
  var NewNode: rNode;var x: tValue;
  var nl, nr: integer;
begin
  if n = 0 then Tree := nil else begin
    nl:=n div 2; nr:=n - nl -1;
    read(x); new(NewNode);
    NewNode.Value:=x.Value;
    NewNode.left   := Tree(nl);
    NewNode.right  := Tree(nr);
    Tree:=NewNode;
  end
end {Tree};

begin
  root:=Tree(21);
end;
```

Vznikne následující binární strom

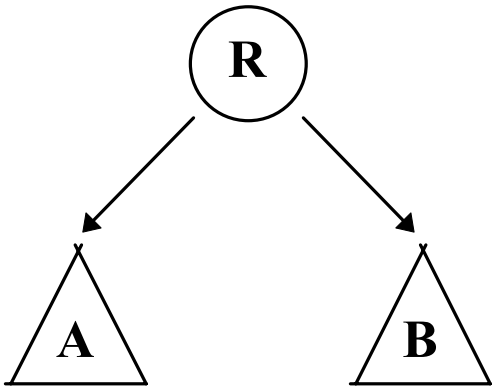


Snadno nahlédneme, že hloubka dokonale vyváženého binárního stromu nepřekročí $\mathrm{Ceil}(\log_2 n)$.

Základní operace na binárních stromech

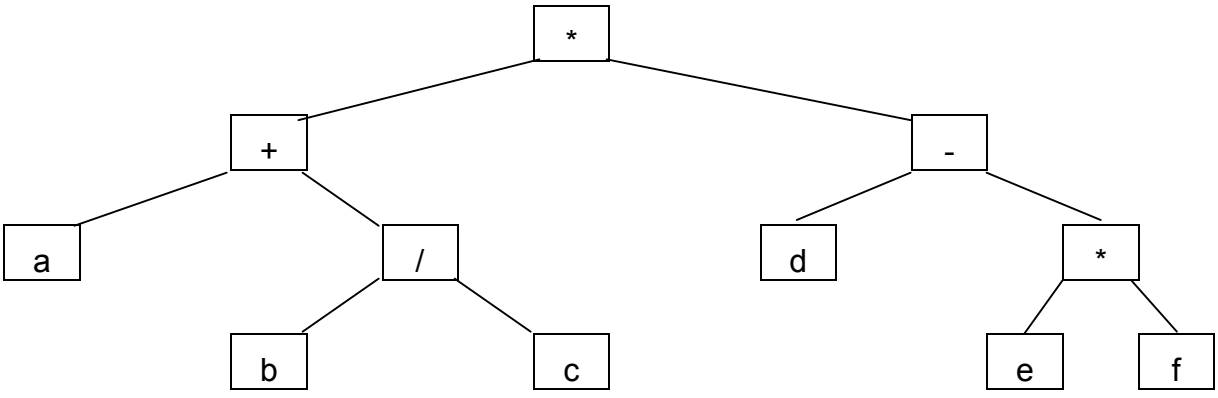
Nejběžnějším zpracováním je uskutečnění operace *P* na každém uzlu stromu, tedy je třeba navštívit všechny uzly. Avšak v jakém pořadí?

Rozlišujeme tři uspořádání, která jsou přirozeným důsledkem struktury stromů.



R - kořen, A - levý a B - pravý podstrom

1. Přímé (pre-order) uspořádání - R, A, B = napřed kořen, pak levý a nakonec pravý podstrom
2. Vnitřní (in-order) uspořádání - A, R, B
3. Inverzní (post-order) uspořádání - A, B, R



Strom reprezentuje výraz (*a* + *b*/*c*) * (*d* - *e***f*), vnitřní uspořádání

Příklad s algebraickým výrazem (*a* + *b*/*c*) * (*d* - *e***f*):

1. Přímé:	* + *a* / *b c* - *d* * *e f*	prefixová notace
2. Vnitřní:	*a* + *b* / *c* * *d* - *e* * *f*	infixová notace
3. Inverzní:	*a b c* / + *d e f* * - *	postfixová notace

Od uspořádání lze odvodit tři metody průchodu stromem (operaci na daném uzlu t^ značíme P(t) a vyjádřit je algoritmicky:

```
procedure PreOrder(t: rNode);
begin if t <> nil then
  begin
    P(t);
    PreOrder(t^.left);
    PreOrder(t^.right);
  end
end;


procedure InOrder(t: rNode);
begin if t <> nil then
  begin
    InOrder(t^.left);
    P(t);
    InOrder(t^.right);
  end
end;


procedure PostOrder(t: rNode);
begin if t <> nil then
  begin
    PostOrder(t^.left);
    PostOrder(t^.right);
    P(t);
  end
end;
```

Binární stromy (ne nutně vyvážené) často slouží k reprezentaci množiny dat s klíči. Pokud strom uspořádáme tak, že všechny klíče v levém podstromu jsou menší než klíč v daném vrcholu a všechny klíče v pravém podstromu jsou větší, pak strom označujeme jako **vyhledávací**.

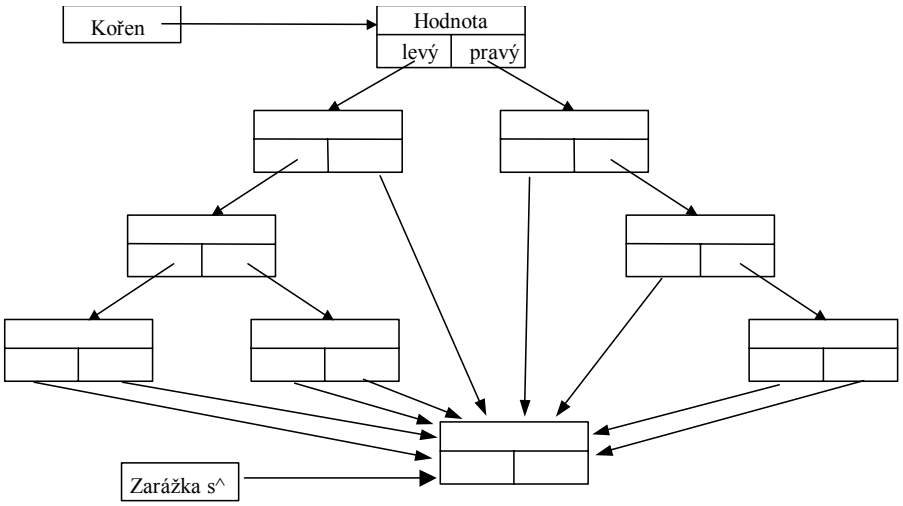
Vyhledávací stromy

Nalezení uzlu s daným klíčem ve vyhledávacím stromu je velmi rychlé:

```
function Locate(sKey: tKey; t: rNode): rNode;
  var found: boolean;
begin
  found := false;
  while not found and (t <> nil) do begin
    found := (t^.Value.Key = sKey);
    if t^.Value.Key>sKey then t:= t^.left
                         else t:= t^.right;
  end;
  Locate := t
end
```

Algoritmus lze dále zjednodušit využitím *zarážky:*

```
function ZLocate(sKey: tKey; t: rNode): rNode;
  var found: boolean;
begin
  s^.Value.Key := sKey; {do zarazky x}
  while t^.Value.Key <> sKey do begin
    if t^.Value.Key>sKey then t:= t^.left
                         else t:= t^.right;
  end;
  ZLocate := t
end
```



Přetvoření existujícího dokonale vyváženého stromu na vyhledávací, je však velmi nákladné. Vyžaduje prakticky úplné přeuspořádání stromu. Velmi často však můžeme podmínku na dokonalé vyvážení stromu oslabit a pak můžeme stromy budovat již přímo jako vyhledávací. To vede na úlohu vyhledávání spojeného s přidáváním, kdy nově příchozí záznam do stromu vhodně přidáme, pokud tam již není. Tento postup ukážeme jako rekurzivní.

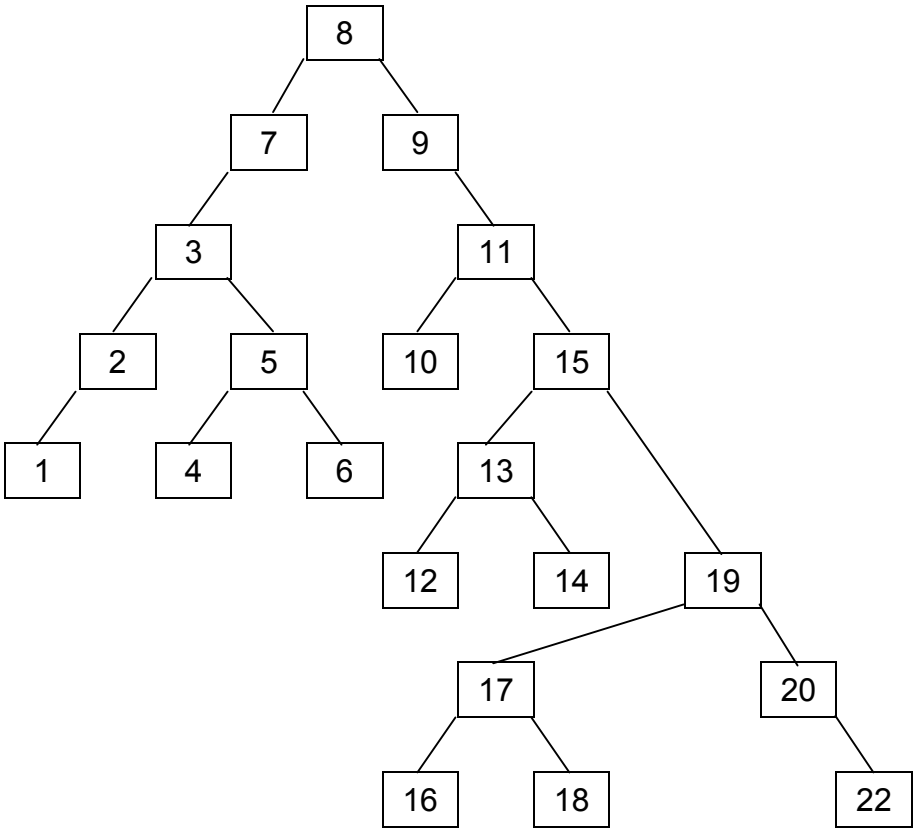
```
procedure Loc_Add(x: tValue; var p: rNode);
begin
  if p = nil then begin
    {záznam nenalezen - přidat}
    new(p);   p^.Value:= x.Value;
    p^.left:= nil;   p^.right:= nil;
  end  else
  if x.Key<t^.Value.Key then Loc_Add(x,t^.left)
  else
  if x.Key>t^.Value.Key then Loc_Add(x,t^.right)
end;
```

Napišme nyní hlavní program

```
var root: rNode; y: tValue;
begin
  root := nil;
  while not eof(input) do begin
    read(y);
    Loc_Add(y, root)
  end
end.
```

Aplikujme opět na soubor 21 datových položek s klíči:

8 9 11 15 19 20 22 7 3 2 1
5 6 4 13 14 10 12 17 16 18



Výsledný strom je však značně nevyvážený. I přesto má své opodstatnění pro relativní jednoduchost a rychlost v aplikacích, které vyžadují jak vyhledávání, tak i třídění.

Rušení vrcholů ve stromu

Inverzním problémem k přidávání je rušení vrcholů. Odstraňování prvku ze stromu je mnohem složitější než jeho přidání. V binárním stromu jsou možné čtyři případy:

(a) prvek s daným klíčem se ve stromu nenachází - není co rušit

(b) prvek je listem - zrušení je triviální

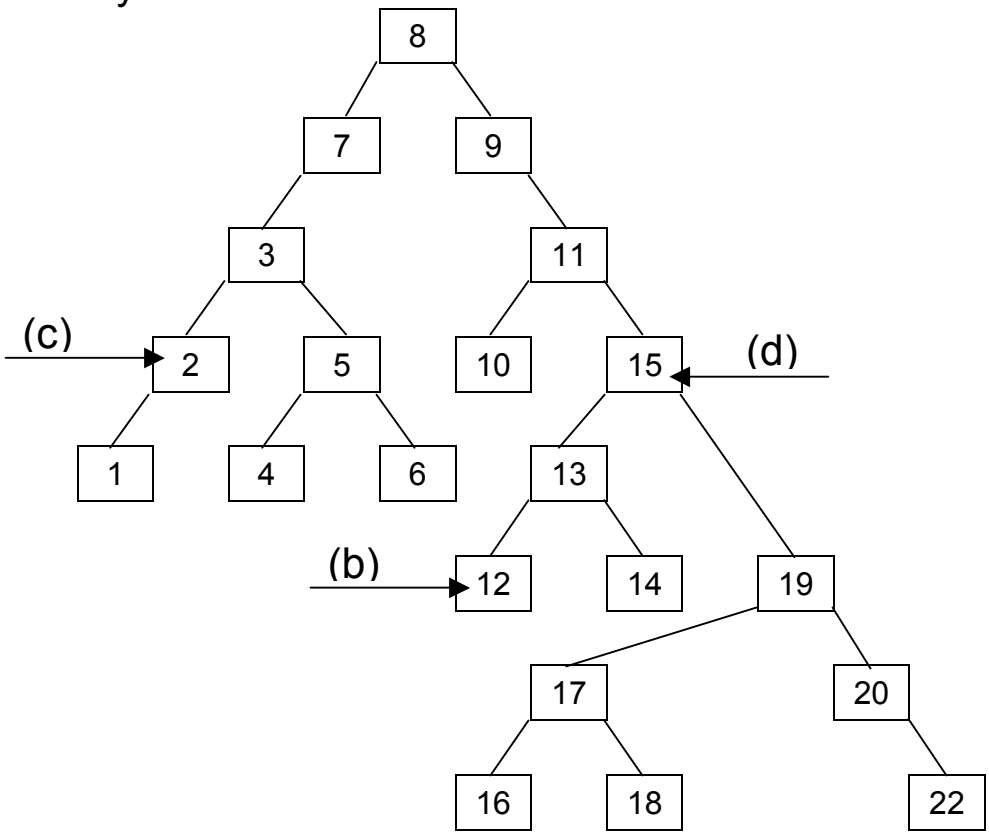
(c) prvek má jednoho následníka - prvek se nahradí svým následníkem

(d) prvek má dva následníky - prvek se nahradí buď nejpravějším prvkem levého podstromu nebo nejlevějším prvkem pravého podstromu

```
procedure Extract(K: tKey; var p: rNode);
  var q: rNode;
  procedure Del(var r: rNode);
  begin   if r^.right<>nil then Del(r^.right)
          else begin
             q^.Value:= r^.Value;
             q:= r; r:= r^.left;
          end
  end;
begin {Extract}
  if p=nil then {není ve stromu} else
  if K<p^.Value.Key then Extract(K,p^.left) else
  if K>p^.Value.Key then Extract(K,p^.right)
  else begin {vlastní vyjmutí prvku}
    q:=p;
    if q^.right=nil   then p:=q^.left   else
    if q^.left=nil    then p:=q^.right  else
    Del(q^.left);
    dispose(q);
  end
end; {Extract}
```

Pomocná procedura `Del` se volá jen v případě 4, kdy sestupuje po nejpravější větvi levého podstromu prvku `q^`, který chceme odebrat. Při návratu zpět vzhůru nahradí podstatné informace v prvku `q^` hodnotami nejpravějšího prvku `r^` v levém podstromu.

Příklady variant rušení vrcholů:



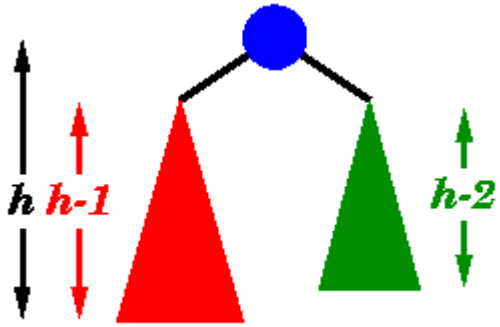
Pozn. Pro vyvážené stromy lze dokázat, že za předpokladu stejné pravděpodobnosti výskytu požadavku na vyhledání pro všechny klíče lze očekávat průměrné zlepšení délky vyhledávané cesty max. o cca 40% (rozhodnutí o nutnosti vyvažování závisí ještě na poměru *r* mezi frekvencemi přístupu do vrcholů (vyhledávání) a přidávání - čím je *r* větší, tím užitečnější reorganizace).

AVL-vyvážené stromy

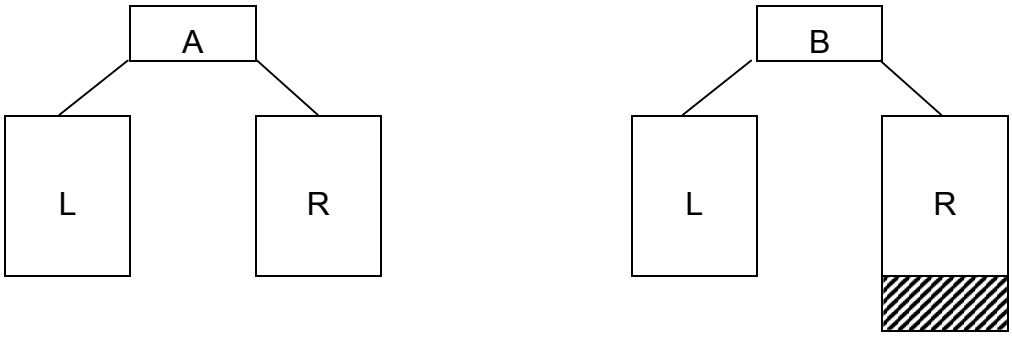
Přidávání a rušení prvků stromu způsobuje nevyvážení. Dokonalé vyvážení stromu je velmi obtížné. Proto se hledaly alternativní definice vyváženosti, které by poskytovaly dostatečně efektivní vyhedávací možnosti, a byly přitom zvládnutelné. ADELSON, VELSKII a LANDIS [1962] publikovali následující kriterium vyváženosti:

Binární strom je (AVL-)vyvážený právě tehdy, když

- ***výšky obou podstromů*** *každého uzlu se liší nejvýše o jedna,*
- ***každý podstrom je AVL*** *vyvážený.*

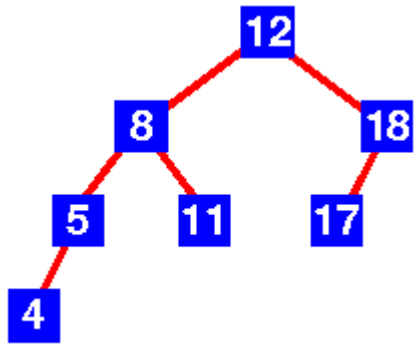


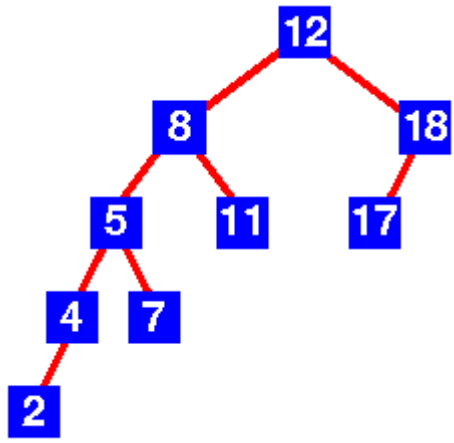
Autoři dokázali, že na vyvážených stromech mají operace *vyhledání prvku podle klíče* a *přidání* či *vyjmutí prvku s daným klíčem* asymptotickou složitost $O(\log n)$, kde n je počet prvků ve stromu.



$h_L = 3$ $h_R = 3$ $h_L = 3$ $h_R = 4$

Zjednodušená lokální podmínka vyváženosti:
Levý a pravý podstrom se ve výšce liší nejvýše o jedna.
Tuto definici je nutné využívat uváženě, nebrání vzniku evidentně nevyvážených stromů:

<table>
<tr>
<td> 
</td>
<td>AVL vyvážený

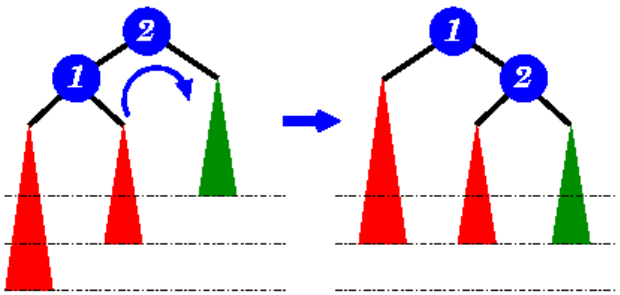
Je zřejmé, že každý levý podstrom má výšku o 1 větší než každý pravý podstrom.</td>
</tr>
<tr>
<td> 
</td>
<td>Nevyvážený

Podstrom s kořenem 8 má výšku 4 a podstrom s kořenem 18 má výšku 2</td>
</tr>
</table>

Přidávání do vyvážených stromů

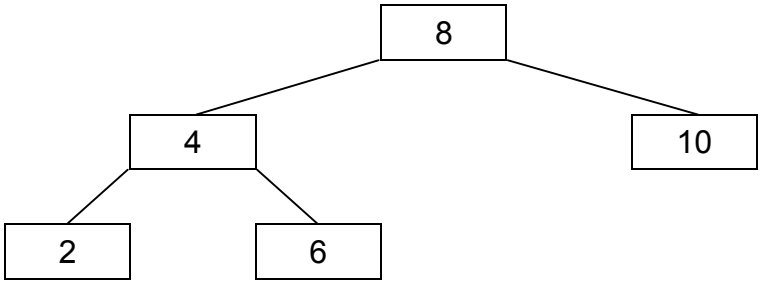
Při přidávání nového vrcholu do stromu mohou nastat 3 případy. Předpokládejme, že přidáváme do levého podstromu, čímž způsobíme nárůst jeho výšky.

1. $h_L = h_R$: L a R budou mít po přidání rozdílné výšky, avšak podmínka vyváženosti se neporuší.
2. $h_L < h_R$: L a R budou mít výšku shodnou, vyvážení se dokonce vylepší.
3. $h_L > h_R$: Kriterium vyváženosti se poruší a strom je nutno rekonstruovat.

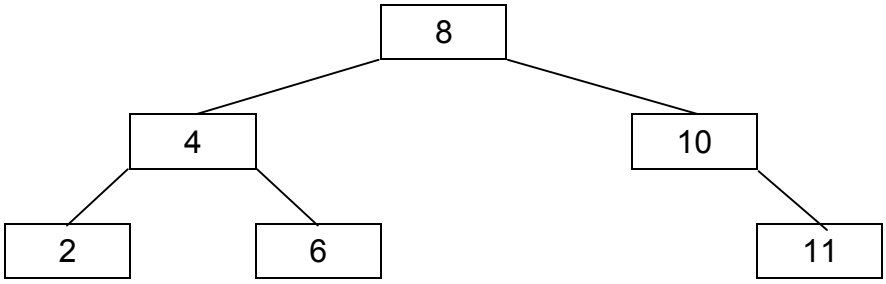


Příklad: Nový prvek byl přidán do levého podstromu vrcholu 1, způsobilo nárůst výšky na hodnotu o 2 vyšší než pravý podstrom vrcholu 2. Vyváženost se obnoví rotací.

Uvažme strom:



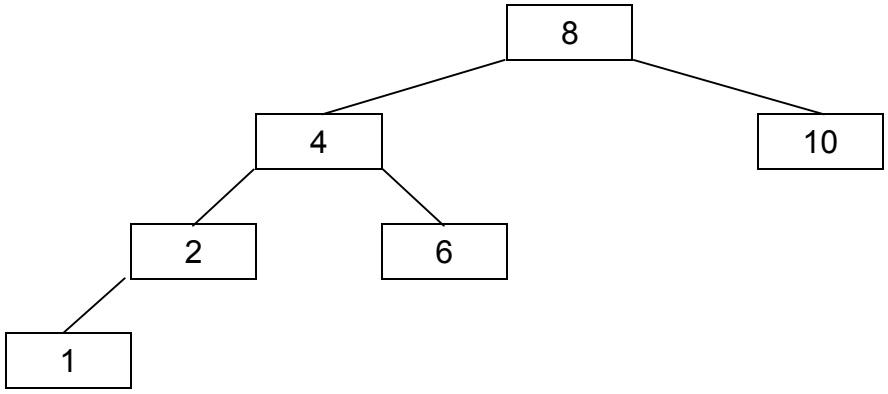
Přidáme-li klíč 11 není nutno vyvažovat (vyvážení se zlepší):



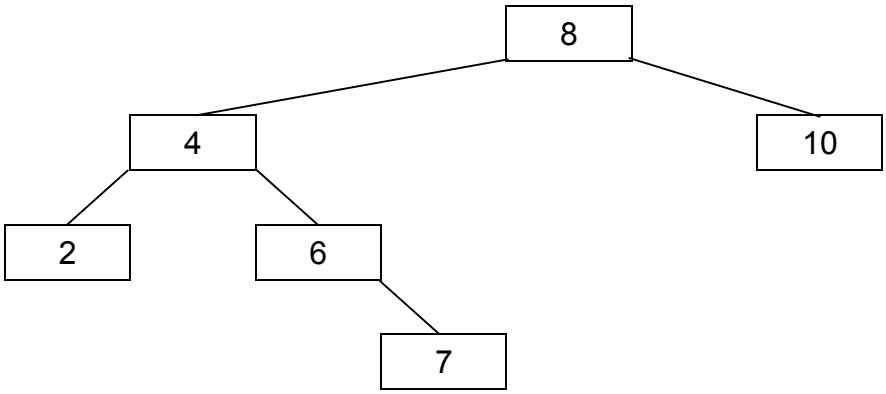
Budeme-li však chtít přidat klíč 1, 3, 5 nebo 7 bude nutno vyvažovat.

Důkladnou analýzou najdeme dva možné případy.

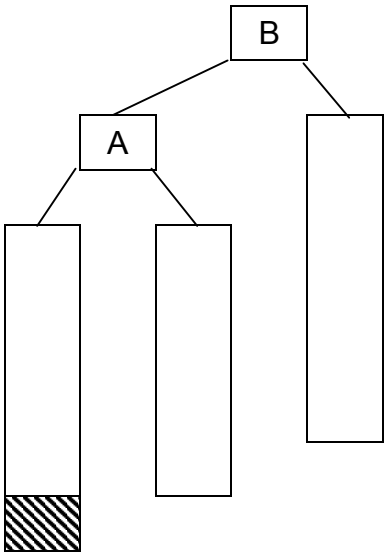
Případ (a) je situace, kdy budeme přidávat klíče 1 nebo 3:



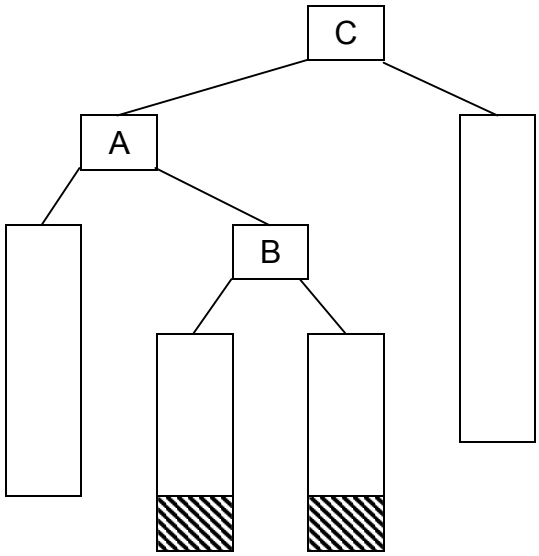
Případ (b) je situace, kdy budeme přidávat klíče 5 nebo 7:



Případ (a)



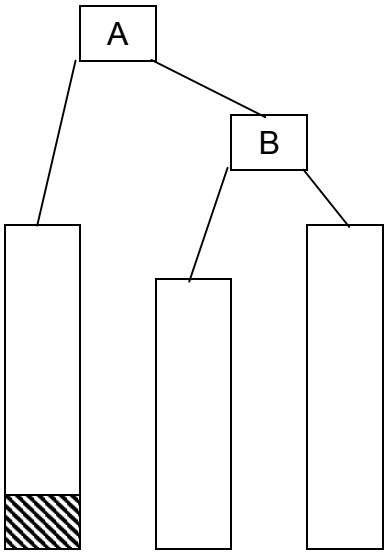
Případ (b)



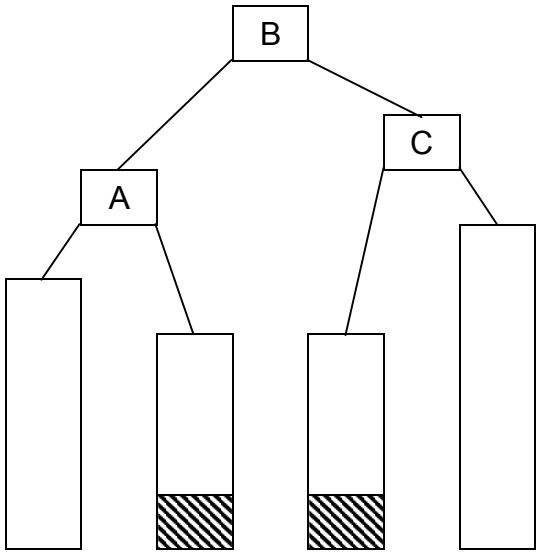
K obnovení vyváženosti si musíme uvědomit, že použité transformace musí zachovat vnitřní (in-order) uspořádání

.

Případ (a)

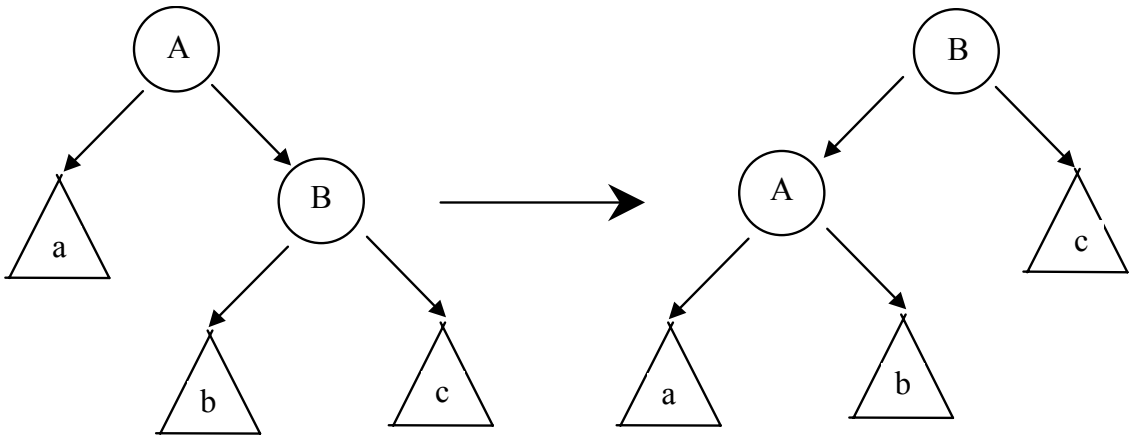


Případ (b)



Pro reorganizaci zavádíme transformace, které modifikují strukturu stromu, avšak zachovávají lexikografické (in-order) uspořádání:

Jednoduchá rotace vlevo

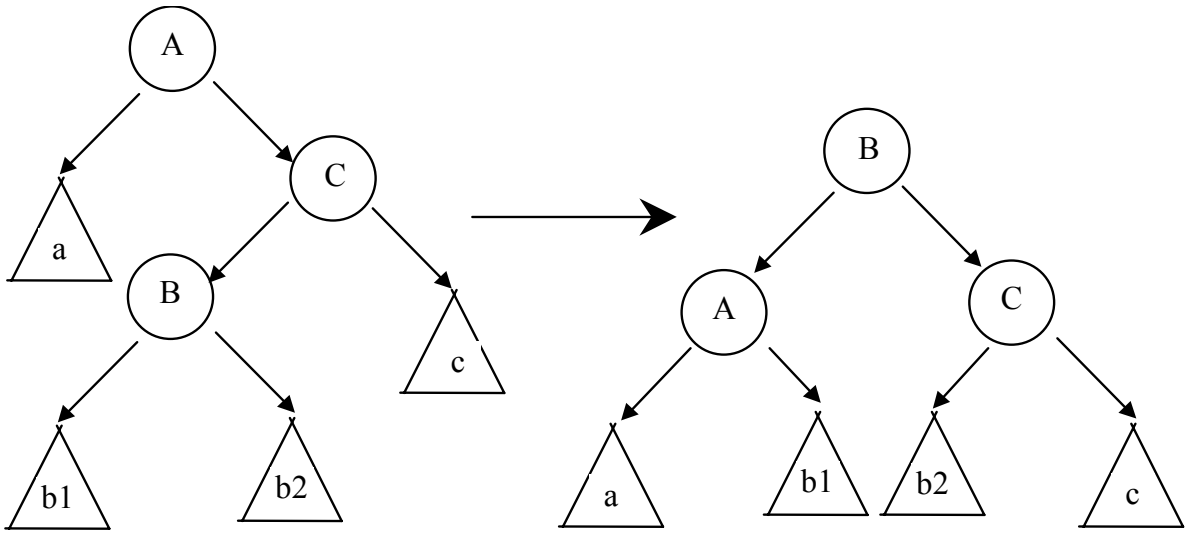



```
procedure lrot(var t:tree); {single left rotation -
RR}
var temp: tree;
begin
  temp := t;
  t:= t^.right;
  temp^.right := t^.left;
  t^.left := temp;
end;
```

Jednoduchá rotace vpravo

```
procedure rrot(var t:tree); {single right rotation -
LL}
var temp: tree;
begin
  temp := t;
  t:= t^.left;
  temp^.left := t^.right;
  t^.right := temp;
end;
```

Dvojitá rotace vlevo




```
procedure dlrot(var t:tree); {double left rotation - LR}
begin rrot(t^.right); lrot(t) end;
```

Symetricky dvojitá rotace vpravo.

Algoritmus vyvažování

Pro efektivní vyvažování zavedeme další položku v definici typu vrchol, tzv. *vyvažovací faktor*:

```
type tNode = record
               value: tValue;
               left,
               right: rNode;
               bal:       (-1 .. +1);
             end;
```

Přidání vrcholu se pak skládá ze tří kroků:

1. Prohledání stromu za účelem zjištění, zda se takový vrchol již ve stromu nenachází.
2. Přidání vrcholu a určení vyvažovacího faktoru.
3. Kontrola vyvažovacího faktoru ve vrcholech ve směru opačném vůči vyhledávání (směrem ke kořenu).

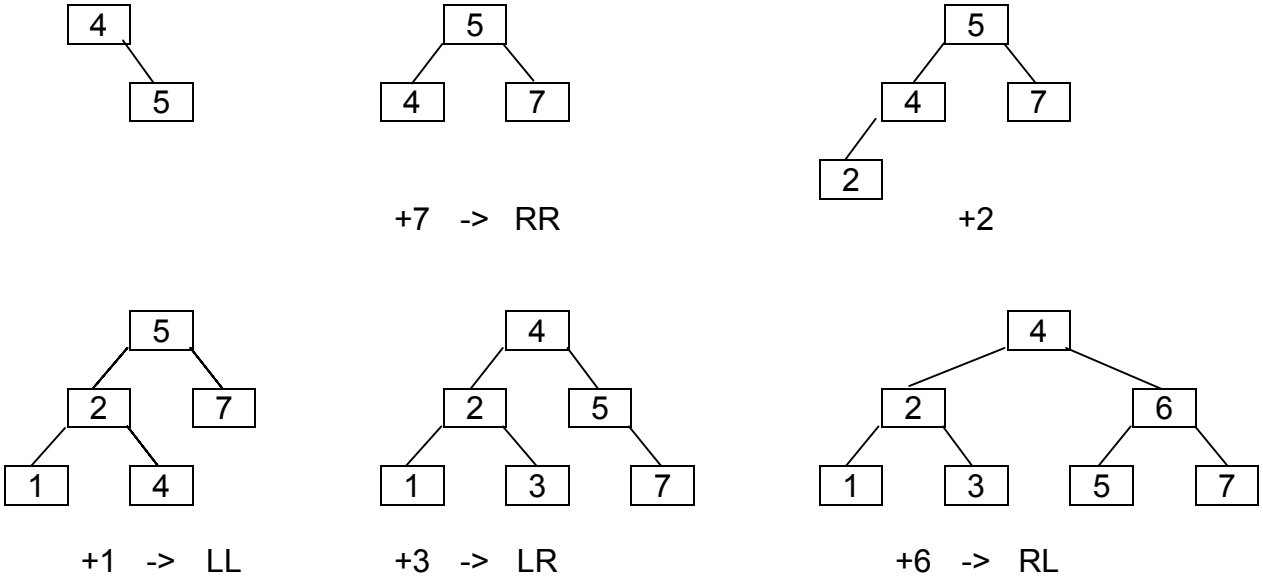
Výsledný algoritmus bude modifikací předchozí procedury `Loc_Add` s tím, že na "zpáteční cestě" rekurze přidáme opravnou operaci. V každém kroku je nutno vrátit informaci, zda se výška podstromu zvětšila, či ne. Proto rozšíříme seznam parametrů procedury o boolský parametr h, který indikuje zvětšení výšky.

Předpokládejme nyní, že se vracíme do vrcholu p^ z jeho levé větve s indikací zvýšení levého podstromu. Mohou nastat 3 výše zmíněné situace:

1. $h_L < h_R$, tj. p^.bal = +1, předcházející nevyváženost se odstraní.
2. $h_L = h_R$, tj. p^.bal = 0, rovnováha se poruší směrem vlevo, avšak AVL-vyváženost ještě zůstává.
3. $h_L > h_R$, tj. p^.bal = -1, je nutné znovuvyvážení.

Ve třetím případě musíme rozlišit nevyváženost typu (a) a (b). Pokud je výška levého podstromu větší než pravého, jde o případ (a), v opačném případě (b). Ve skutečnosti se vyvažování děje vhodnými záměnami ukazatelů, které se cyklicky zaměňují.

Než uvedeme celý algoritmus v programové formě, ukažme postup na jednoduchém příkladu:



```
procedure Loc_AddB( x: tValue; var p: rNode;
                    var h: boolean);
  var p1, p2: rNode;
begin
  if p = nil then begin
    {záznam nenalezen - přidat}
    new(p);     p^.Value:= x.Value;
    p^.left:= nil;  p^.right:= nil;
    h:= true;     bal:= 0;
  end else
  if t^.Value.Key<x.Key then begin
    Loc_AddB(x,t^.left,h);
    if h then {levá větev se zvýšila}
    case p^.bal of
     1: begin p^.bal:=0; h:= false; end;
     0: p^.bal:= -1;
    -1: begin {je nutno vyvažovat}
          p1:= p^.left;
          if p1^.bal = -1 then
          begin {jednoduchá LL rotace}
            p^.left:= p1^.right;    p1^.right:= p;
            p^.bal:= 0;             p:= p1;
          end else
          begin {dvojitá LR rotace}
            p2:= p1^.right;         p1^.right:= p2^.left;
            p2^.left:= p1;          p1^.left:= p2^.right;
            p2^.right:= p;
            if p2^.bal=-1   then p^.bal:=+1
                            else p^.bal:=0;
            if p2^.bal=+1   then p1^.bal:=-1
                            else p1^.bal:=0;
            p:= p2;
          end;
          p^.bal:= 0; h:= false;
        end {vyvažování}
    end {case};
  end else
  if t^.Value.Key>x.Key then begin
    Loc_AddB(x,t^.right,h);
    if h then {pravá větev se zvýšila}
    case p^.bal of
    -1: begin p^.bal:=0; h:= false; end;
     0: p^.bal:= +1;
     1: begin {je nutno vyvažovat}
          p1:= p^.right;
          if p1^.bal = +1 then
          begin {jednoduchá RR rotace}
            p^.right:= p1^.left;    p1^.left:= p;
            p^.bal:= 0;             p:= p1;
          end else
          begin {dvojitá RL rotace}
```

```
            p2:= p1^.left;          p1^.left:= p2^.right;
            p2^.right:= p1;         p1^.right:= p2^.left;
            p2^.left:= p;
            if p2^.bal=+1   then p^.bal:=-1
                              else p^.bal:=0;
            if p2^.bal=-1   then p1^.bal:=+1
                              else p1^.bal:=0;
            p:= p2;
          end;
          p^.bal:= 0; h:= false;
        end {vyvažování}
    end {case};
  end else
    h:= false
end;  {Loc_AddB}
```

Operace rušení uzlu ve vyváženém stromu je variací na rušení uzlu v obyčejném binárním stromu, doplněná o vyvažování pomocí stejných záměnných operací.

Binární vyhledávací stromy vs. AVL stromy

Operace vkládání:

AVL	Red-Black strom
Dva průchody - dolů k vkládanému uzlu - zpět nahoru s vyvažováním	Dva průchody - dolů k vkládanému uzlu - zpět nahoru s vyvažováním

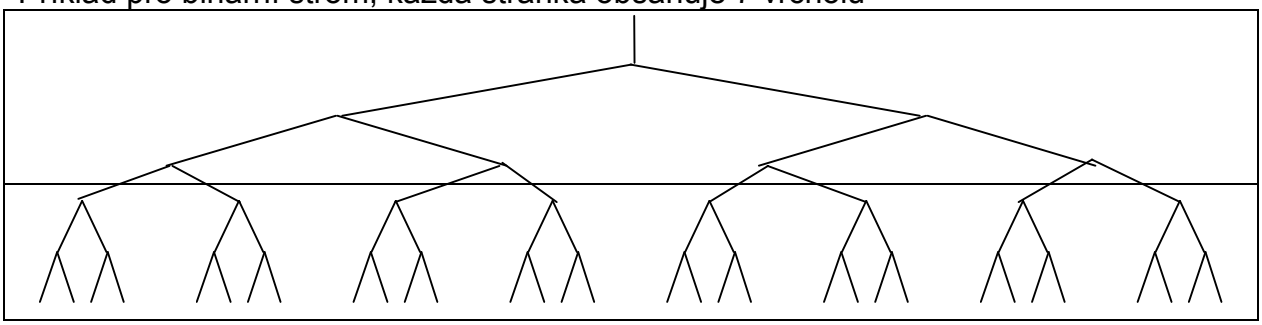
V literatuře:

- Cormen T.H.: Introduction to algorithms. Cambridge, Mass : MIT Press; New York : McGraw-Hill, 1990
  --> bez preferencí,
- Wirth, N: Algorithm + Data Structures = Program. Prentice-Hall, New Jersey 1975 (a další vydání)
  --> naopak upřednostňuje AVL stromy, avšak
- Weiss, M.A., Algorithms, Data Structures and Problem Solving with C++, Addison-Wesley, 1996,
  --> ukazuje řešení s R-B stromy v jednom průchodu!

Vícecestné (m-ární) stromy

Motivace - rozsáhlé stromy, které nutno rozdělit na **stránky** uložené na disku.

Příklad pro binární strom, každá stránka obsahuje 7 vrcholů

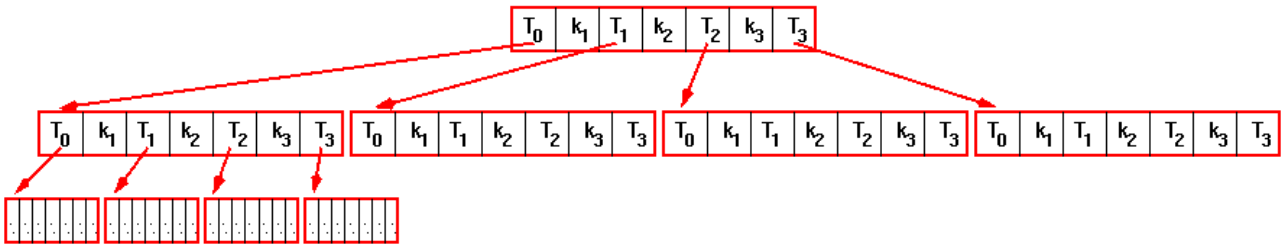


Výhody:

- Redukce hloubky stromu na úroveň $O(\log_m n)$ pro m-cestné stromy, m potomků, m-1 klíčů ve stránce --> pak pro m = 10 je 106 klíčů v 6 úrovních vs. 20 v binárním stromu
- Uvažme množinu dat s 10^6 elementy. Čistý binární vyhledávací strom by vyžadoval až $\log_2 10^6 \sim 20$ přístupů na disk. Budou-li stránky obsahovat po 100 elementech bude zapotřebí jen max.
  $\log_{100} 10^6 = 3$ přístupy. Stromy však nemůžeme nechat růst náhodně.

ale

- Je nutné v každé stránce procházet m-1



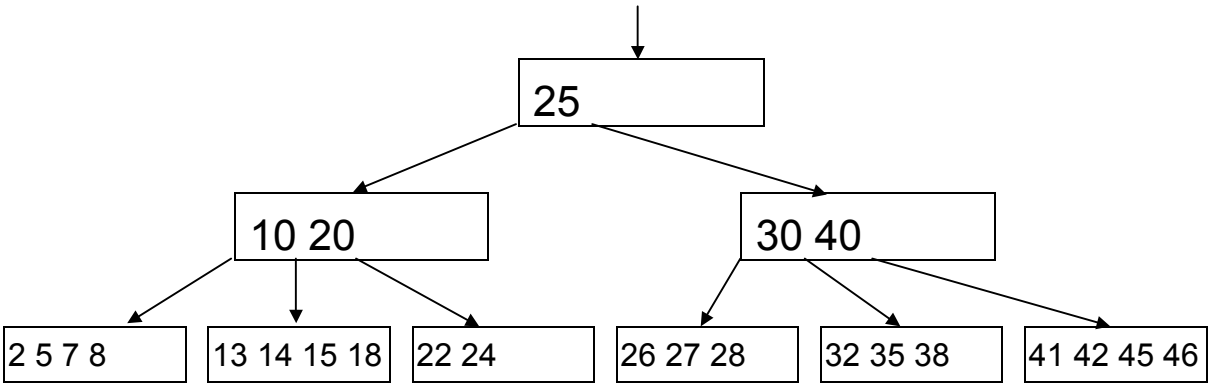
B-stromy

[BAYER 1970]: každá stránka (až na jednu) musí obsahovat *n* až 2*n* uzlů, kde *n* je zvolená konstanta. Je-li známa *velikost* uzlu a velikost stránky, pak 2*n* uzlů se musí vejít do stránky a stránka musí být aspoň z 50% plná. **B-strom** řádu *n* tedy splňuje následující kriteria:

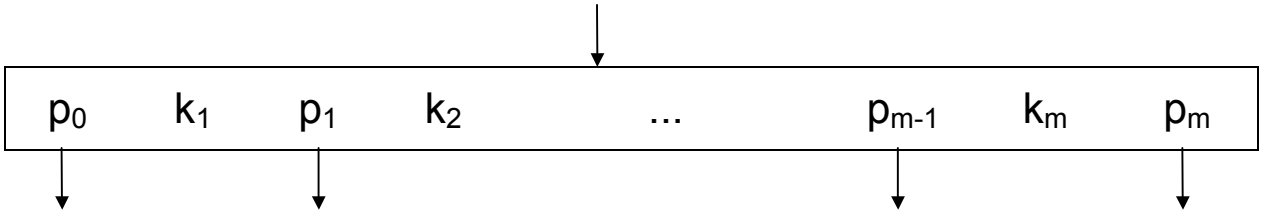
1. Každá stránka obsahuje nejvýše 2*n* položek (=klíčů).
2. Každá stránka *s výjimkou kořene* stromu obsahuje aspoň *n* položek.
3. Každá stránka je buď listem stromu (a pak nemá následníky) nebo má právě *m*+1 následníků, kde *m* je skutečný počet položek ve stránce.
4. Všechny listové stránky jsou na jedné úrovni.

Potom pro nejhorší případ je nutno $log_a a$ přístupů.

B-strom řádu 2



Klíče jsou ve stránkách uspořádány vzestupně a "prostřídány" s ukazateli na následníky:



Vyhledávání v B-stromu je jednoduché: Načteme stránku do paměti a v ní vyhledáváme nějakou metodou pro hledání v uspořádaném poli (sekvenčně nebo půlením). Pokud klíč *x* v poli nenajdeme, pak:

1. $k_i < x < k_{i+1}$ pro $1 <= i < m$. Vyhledávání pokračuje na stránce p_i^.
2. $k_m < x$. Vyhledávání pokračuje na stránce p_m^.
3. $x <. k_1$. Vyhledávání pokračuje na stránce p_0^.

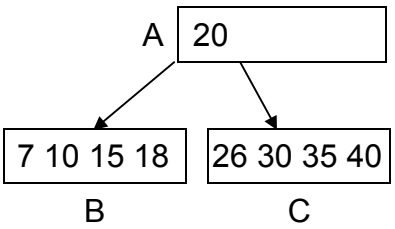
Přidávání do B-stromu

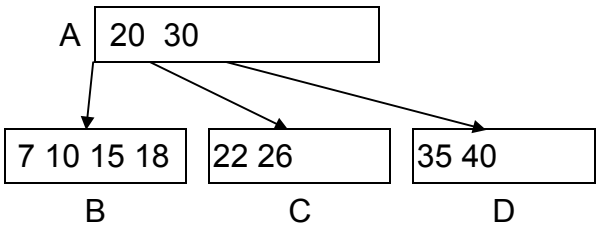
Přidávání do B-stromu je také jednoduché, pokud je ve stránce místo. Pokud není, pak musíme strom lokálně přeuspořádat.

Ukažme přidání položky s klíčem 22 do B-stromu řádu 2:

1. Klíč 22 chybí, přidání do stránky C není možné (zaplněna)
2. Stránka C se rozdělí na dvě stránky (tj. vytvoří se nová stránka D)
3. Přetékající stránku rovnoměrně rozdělíme do C a D a "prostřední" hodnotu klíče přesuneme o úroveň výše, do stránky předchůdců A.

Toto může rekurzivně probíhat až do kořene stromu. Pokud přeteče kořen, výška stromu se zvětší.





Obráceným postupem položky rušíme. Pokud při zrušení položky klesne počet položek ve stránce pod *n*, musíme dvě stránky spojit v jednu. Pokud rušený prvek není v listu stromu, musíme ho nahradit některým z lexikograficky sousedních prvků z následníků dané stránky. Budeme volit vždy tak, aby následnická stránka (pokud možno) nepodtekla.

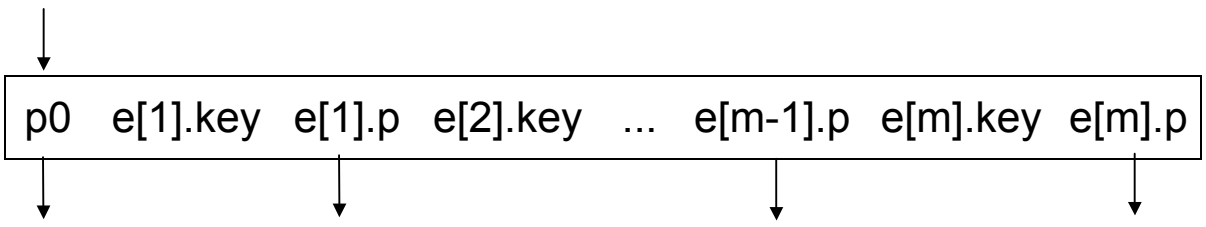
Příklad datové reprezentace

```
const nn = 2*a;
type  tRef = ^tStranka;
type  tIndex = 0..nn;

type tPrvek = record
            key:  integer;
            p:    tRef;
            value:   {ostatní položky dat};
          end;

type tStranka = record
            m:    tIndex;
            p0:   tRef;
            e:    array [1..nn] of tPrvek
          end;
```

Uvědomme si, že každá stránka představuje prostor pro 2*n* prvků. Položka *m* určuje skutečný počet prvků ve stránce. Je-li $m \geq n$ (vyjma kořenové stránky), máme zaručeno alespoň 50% využití paměti.



Algoritmus vyhledávání a přidávání:

```
procedure Loc_Add(x:integer; refStr:tRef;
                  var h:boolean; var u:tPrvek);
begin if refStr = nil then
  begin {x není ve stromu}
    priřaď x prvku u, nastav h:=true - příznak
    putování prvku u nahoru stromem
  end else
  with refStr^ do {přesun stránky do paměti}
  begin {vyhledej x ve strance refStr^}
    prohledej stranku (pole);
    if (x nalezen) then
       {nalezen}
    else begin
       Loc_Add(x, nasledovník, h, u);
       if h then {prvek postoupil nahoru}
          if (počet prvku v refStr^)<2n then
          přidej u do stránky refStr^, h:=false
       else
          rozděl stránku a přesuň střední prvek
    end
  end
end;
```

Algoritmus připomíná jednoduché vyhledávání v binárním stromu vyjma rozhodování o větvení, které není založeno na binárním výběru. Vyhledávání v rámci stránky může být realizováno metodou binárního půlení.

Algoritmus přidávání se provádí ve chvíli, kdy je indikován (*h=true*) pohyb prvku směrem vzhůru (ke kořenu). Příznak *h* má obdobnou funkci jako v případě přidávání do vyváženého stromu, kde se signalizoval nárůst podstromu. Má-li *h* hodnotu true, pak parametr *u* reprezentuje prvek, putující vzhůru.

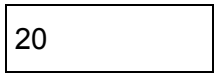
Napišme nyní hlavní program

```
var root, q: tRef; y,u: tPrvek; h: boolean;
begin
  root := nil;
  while not eof(input) do begin
    read(y);
    Loc_Add(y, root, h, u);
    if h then
    begin {vytvoř a připoj novou stránku}
       q:= root;
       new(root);
       with root^ do
          begin m:=1; p0:=q; e[1]:=u end;
    end;
  end
end.
```

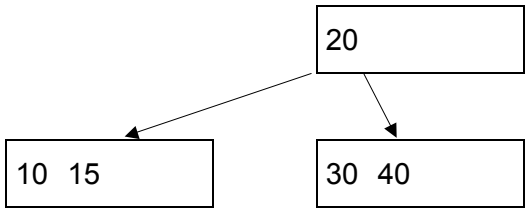
Jestliže má parametr *h* při návratu z procedury *Loc_Add* hodnotu true, znamená to, že se dělí kořenová stránka. Její rozdělení je zvlášť ošetřeno a spočívá pouze z vytvoření nové kořenové stránky a z vložení jediného prvku *u*.

Příklad růstu B-stromu pro následující posloupnost přidávaných klíčů:

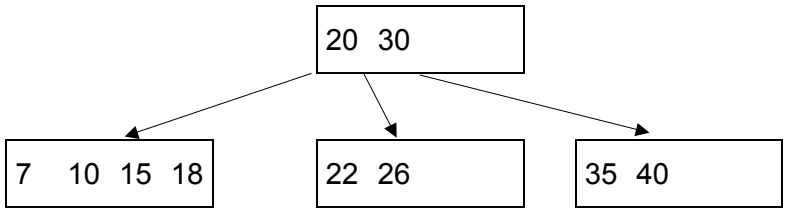
20



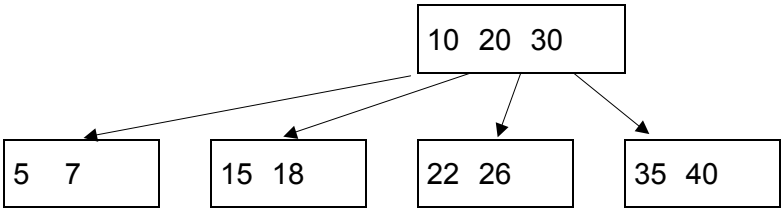
40, 10, 30, 15



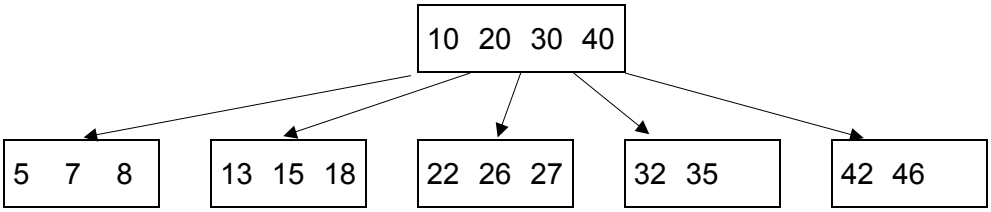
35, 7, 26, 18, 22



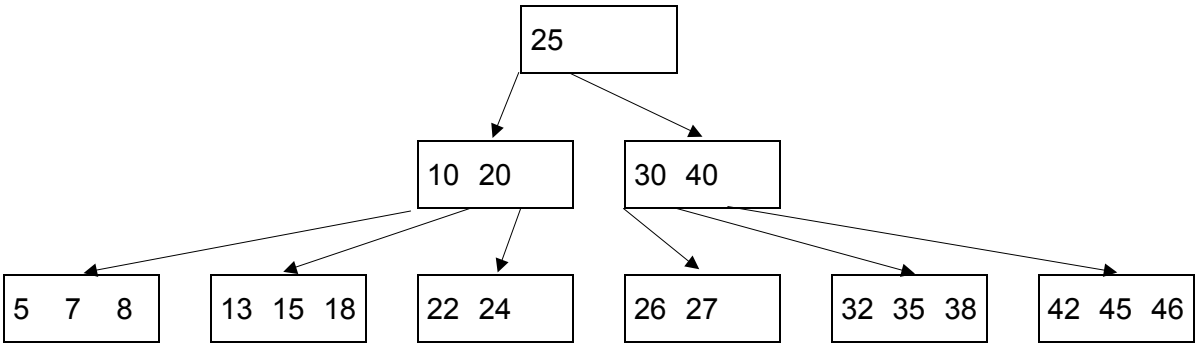
5



42, 13, 46, 27, 8, 32



38, 24, 45, 25



Všimněme si praktických důsledků z hlediska paměťové náročnosti:

Uvnitř procedury *Loc_Add* pracujeme se stránkou, odkazovanou *refStr.* Z hlediska efektivity programu bude výhodné, podaří-li se stránku *refStr^* umístit do operační paměti. Protože tak každé volání procedury *Loc_Add* znamená vytvoření jedné stránky v paměti, bude potřeba nejvýše $k=log_nN$ rekurzivních volání. Pro strom o velikosti *N* prvků musíme být schopni umístit v operační paměti alespoň *k* stránek o velikosti *2n*. Ve skutečnosti jsou nároky ještě vyšší, neboť při přidávání prvku může nastat proces dělení, a tím vznik nových stránek. Přirozeným důsledkem je požadavek na trvalé umístění kořenové stránky v operační paměti, protože každý dotaz je zahájen bezprostředně v ní.

Odebírání prvků z B-stromu

V procesu odebírání je nutné rozlišit dva případy - klíč *x*, který odebíráme

1. nachází se v listové stránce (pak je algoritmus odebrání prvku jednoduchý a jasný)
2. není v listové stránce - pak se musí nahradit jedním ze dvou lexikograficky sousedících prvků, který je možné jednoduše odebrat, pokud se nachází v listové stránce.

Způsob hledání sousedního klíče je analogický s metodou odebírání prvků z binárního stromu.

Proto budeme postupovat takto:

1. sestoupíme ve směru pravého ukazatele až do nejlevější listové stránky *P*,
2. nahradíme prvek, který se má odebrat, nejlevějším prvkem stránky *P* a
3. snížíme velikost obsazení *P*

(v podstatě hledáme nejmenší větší klíč, resp. následovníka z uspořádané posloupnosti klíčů). Alternativně lze sestupovat ve směru levého ukazatele k nejpravějšímu prvku.

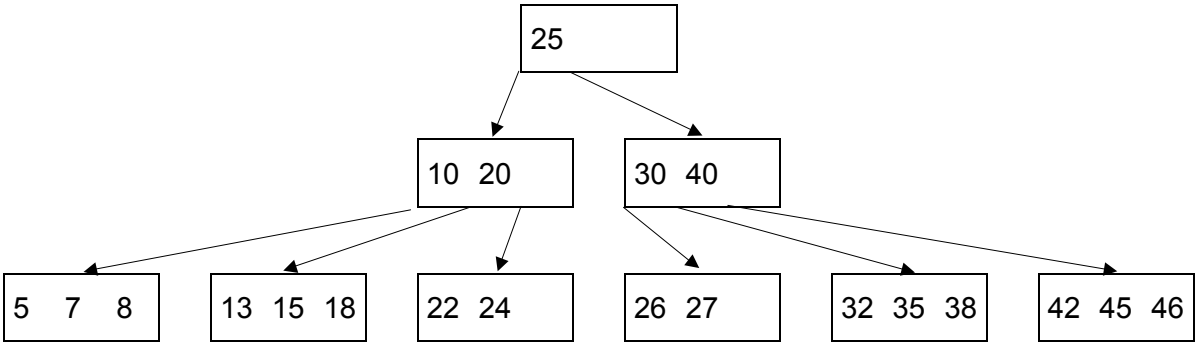
Po snížení velikosti stránky musí následovat kontrola počtu prvků *m* ve stránce. Je-li *m* < *n* (nezaplnění stránky), je porušena základní charakteristika B-stromů a je nutné provést reorganizaci. Řešením je přesunutí prvku ze sousední stránky *Q* do stránky *P*. Vzhledem k tomu, že toto vyžaduje načtení celé stránky *Q* do paměti, provádí se obvykle více než přesun jednoho prvku - obsazení stránek *P* a *Q* se vyrovnává = *vyvažování*.

V případě, že stránka *Q* již dosáhla své minimální velikosti *n*, pak celkový počet prvků na stránkách *P a Q* je *2n-1* a stránky můžeme sloučit:

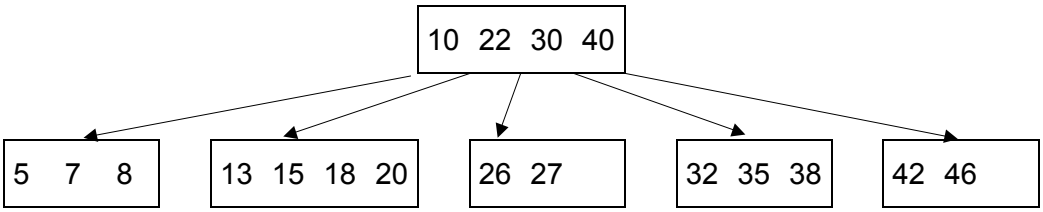
1. do stránky *P* přesuneme "prostřední" prvek stránky předchůdců *P* a *Q*
2. do stránky *P* přesuneme prvky ze stránky *Q* a
3. stránku *Q* zrušíme.

Takovéto odstranění prostředního klíče ze stránky předchůdců může způsobit zmenšení velikosti stránky pod přípustnou mez *n* a tím další vyvažování či slučování stránek na vyšší úrovni. V extrémním případě se tento proces může rozšířit až po kořen B-stromu. Když se velikost kořenové stránky zmenší na nulu, je třeba ji odstranit. Toto je v podstatě jediný způsob zmenšení výšky B-stromu.

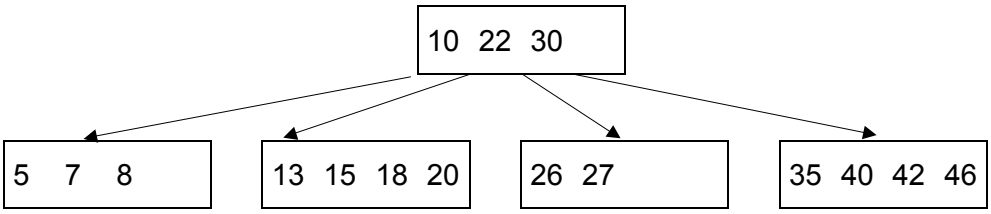
Postupný zánik B-stromu budeme ilustrovat na příkladě sekvenčního odebírání klíčů:



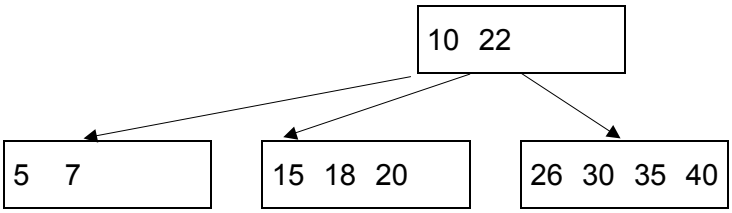
25, 45, 24



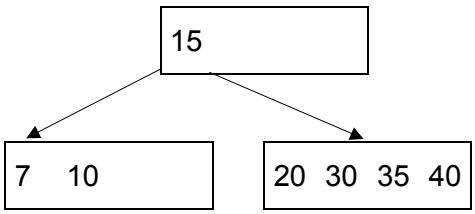
38, 32



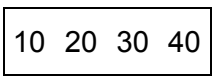
8, 27, 46, 13, 42



5, 12, 18, 26

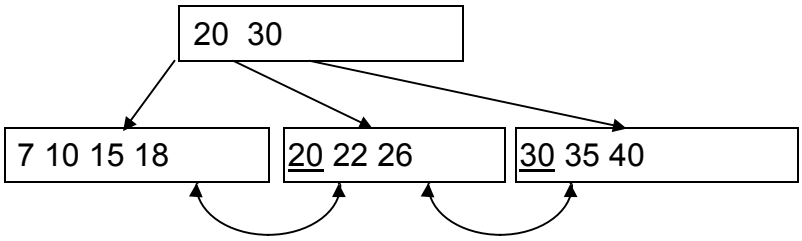


7, 35, 15

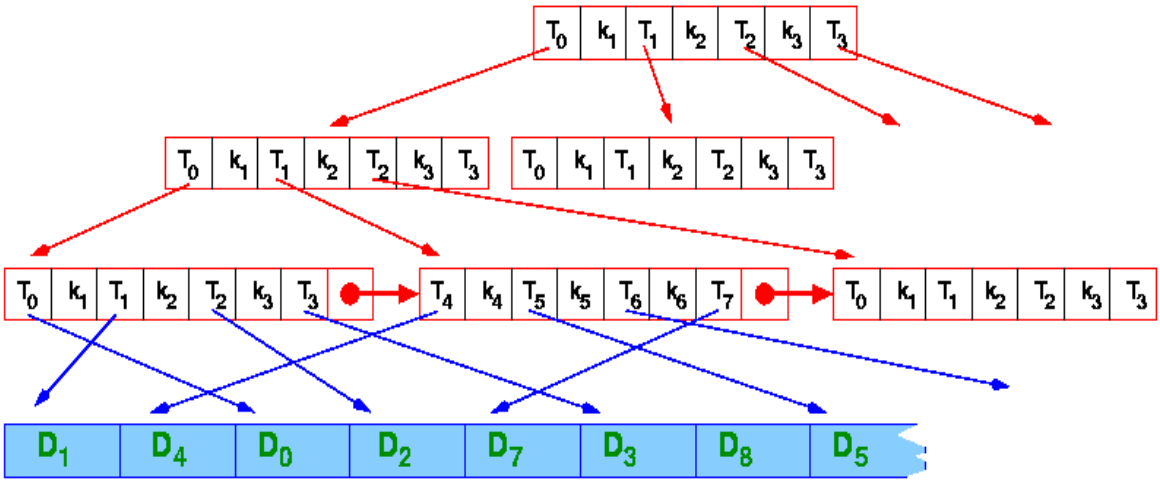


B+ stromy

Jsou modifikací B-stromů tak, že klíče jsou zopakovány v následnících a listové stránky jsou navzájem propojeny obousměrnými spojkami:



B+ stromy se používají v moderních databázích jako forma indexních souborů (např. dBase nebo FoxPro). Při klasickém vyhledávání se postupuje prakticky stejně jako u obyčejných B-stromů. Dokonce **ukazatele** (diskové adresy dat) **se nacházejí jen v listech**.



B+ stromy jsou velmi výhodné z hlediska vytváření uspořádaných pohledů na data. Stačí procházet listy stromu a tím získávat data setříděná podle konkrétního klíče.

Z důvodů optimalizace přístupové doby korespondují stránky s diskovými bloky (ms vs. ns).

Nevýhodou je opakování klíčů a složitější údržba stromů.