INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

APLIKOVANÁ INFORMATIKA

4. LEKCE – DYNAMICKÉ DATOVÉ STRUKTURY

prof. Ing. Radim Farana, CSc.

Ostrava 2013

ISBN 978-80-248-3017-9

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

4 DYNAMICKÉ DATOVÉ STRUKTURY

OBSAH KAPITOLY:

Dynamický seznam

Stromy

MOTIVACE:

Pro efektivní řešení řady složitějších problémů je potřeba dobře pracovat s dostupnou pamětí. K tomu se s výhodou používají dynamické datové struktury, zejména, pokud předem nevíme, jaký objem dat budeme zpracovávat. Mezi nejjednodušší dynamické datové struktury patří lineární seznamy a stromy. Pro jejich využití je třeba se s nimi dobře obeznámit a umět je správně používat.

CÍL:

Naučit se znát vlastnosti a využití jednotlivých dynamických datových struktur. Umět používat lineární seznamy a vědět jak s jejich pomocí realizovat datové struktury jako je fronta, zásobník a setříděný seznam. Umět vkládat i vyjímat prvky z lineárního seznamu i setřídit prvky lineárního seznamu. Rozuměn pojmům z oblasti stromů jako je stupeň stromu, hloubka (výška) stromu, rozhodovací a vyhledávací stromy a další. Znát základní algoritmy pro práci s binárními stromy, umět vkládat prvky do stromu i vyjímat prvky ze stromu, umět zkonstruovat vyvážený strom, AVL strom i optimální strom. Rozumět konstrukci B-stromu a umět ho efektivně využívat.

Jako **dynamické datové struktury** chápeme takové struktury, které během zpracování mění svoji strukturu (velikost zabírané paměti). Tím se výrazně liší od **datových struktur statických**. Výhodou dynamických struktur je skutečnost, že nezabírají paměť stále, ale jen pokud je jich potřeba.

Základem dynamických datových struktur je speciální **datový typ** nazvaný **ukazatel** (*pointer*). Zabírá 4 Byte (případně 2 Byte), které mají význam adresy v paměti, na které se nacházejí vlastní data. Při práci s proměnnou typu ukazatel musíme rozlišit dva případy:

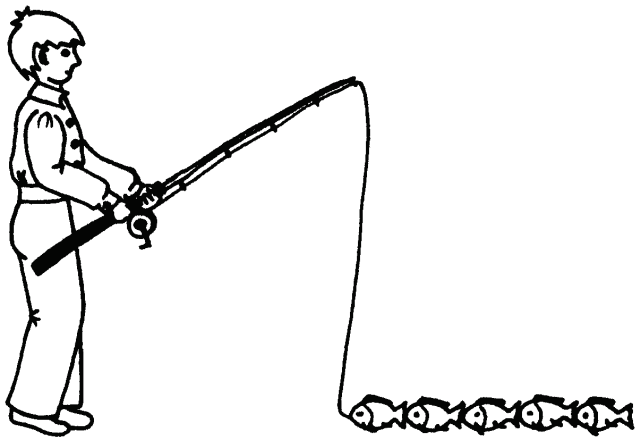
Jednak je to práce přímo s obsahem této proměnné. Odkazujeme se na něj názvem této proměnné, např. p.

Druhým případem je práce s daty, na která proměnná ukazuje. Odkazujeme se na ně názvem proměnné doplněným **znakem ukazatele** (znázorňuje se obvykle stříškou „^“ nebo šipkou nahoru „↑“, pro větší zřetelnost budeme dále používat znak šipky), např. p↑

Data, na která proměnná ukazuje, mohou mít jakoukoliv strukturu. Pomocí ukazatelů můžeme vytvořit pomocné proměnné pro algoritmus, které po jeho skončení opět uvolníme z paměti. Pomocí statických proměnných bychom pomocné proměnné mohli vytvořit dvěma způsoby.

pomocí **globálních proměnných**, pak budou zabírat paměť po celou dobu práce programu,

pomocí **lokálních proměnných**, pak budou zabírat paměť po dobu práce rutiny, která je deklarovala,



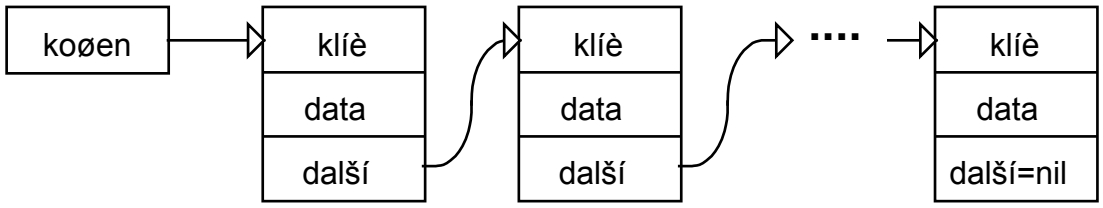
Obrázek 4.1 Dynamický seznam (lineární seznam)

Lokální proměnné se však alokují v paměti pro **zásobník**. Sem se ukládá také informace o volání rutin, takže této paměti není k dispozici příliš mnoho. Hlášení o **přetečení zásobníku** (*stack overflow error*), patří k velmi nepříjemným, neboť si vynucuje změnu přístupu k řešení problému. Použití dynamických proměnných může výrazně pomoci.

Proměnné typu ukazatel obvykle nepoužíváme pro základní datové typy, ale pro **datové typy složené**, ať již jsou to pole, nebo proměnné typu **struktura** (*record*). Pro nás bude zajímavější využití složených datových typů. Umožní nám tvořit datové struktury, které mají skutečně zcela dynamický charakter. Obvykle je možno rozdělit vlastní data na tři části:

1. **klíč** (*index*) – data, podle kterých třídíme informaci,
2. další data, často ukazatel na další datovou strukturu,
3. ukazatel na další proměnnou stejného datového typu.

Zřetězením těchto dat vzniká dynamická datová struktura – **dynamický seznam** (lineární seznam, vázaný seznam). K jeho vytvoření a samozřejmě také k přístupu k seznamu potřebujeme proměnnou typu ukazatel na danou datovou strukturu, nazveme ji **kořen**. Jejím obsahem bude adresa první dynamické proměnné. Ta bude ukazovat na další **prvek seznamu** atd.



Obrázek 4.2 Dynamický seznam (lineární seznam, vázaný seznam)

Velmi důležité je rozpoznání konce struktury. Poslední ukazatel totiž již nemá kam ukazovat. Přitom není možné, aby tento ukazatel měl libovolnou hodnotu, nepoznali bychom, že již neukazuje na další proměnnou. Pro tyto účely je v jazyku PASCAL definována hodnota **nil** pro ukazatel, který je prázdný, tj. neukazuje na žádná data. Toto označení budeme nadále používat i my.

Kromě toho je třeba definovat alespoň dvě základní procedury pro **dynamické přidělování paměti**. Opět využíváme funkcí známých z jazyka PASCAL:

new(p) – Přidělí dynamické proměnné paměť pro její datovou strukturu a adresu této paměti uloží do proměnné **p** typu ukazatel.

dispose(p) – Uvolní paměť přidělenou proměnné **p**. Obsah proměnné p se ale obvykle nemění.

Pozorný čtenář si jistě uvědomil, že obě procedury musí znát potřebnou velikost paměti pro datovou strukturu. Obvykle je problém řešen tak, že při deklaraci proměnné definujeme její typ jako ukazatel na určenou datovou strukturu. Některé jazyky pak nedovolují přiřadit proměnné obsah jiného datového typu. Pro některé aplikace to není vhodné a požadujeme proměnné, které mohou ukazovat na jakoukoliv datovou strukturu. Pro přidělení a uvolnění paměti pak musíme použít procedury, kterým sdělíme také velikost přidělované (uvolňované) paměti, tuto velikost si ovšem musíme stále pamatovat. V jazyku PASCAL jsou to:

GetMem (p, size)

FreeMem (p, size)

V dalším textu budeme pracovat s procedurami `new(p) a dispose(p).`

4.1 DYNAMICKÝ SEZNAM

Nejjednodušší dynamickou strukturou je **lineární seznam**, viz obr.4.2.

Datová struktura je přístupná pomocí proměnné **kořen**. Tím je lineární seznam podobný struktuře **souboru**. Do souboru však obvykle musíme nové prvky ukládat jen na konec souboru a rušit celý soubor najednou. U lineárního seznamu máme možností více. Nyní rozebereme základní operace s lineárním seznamem.

4.1.1 Vkládání prvků

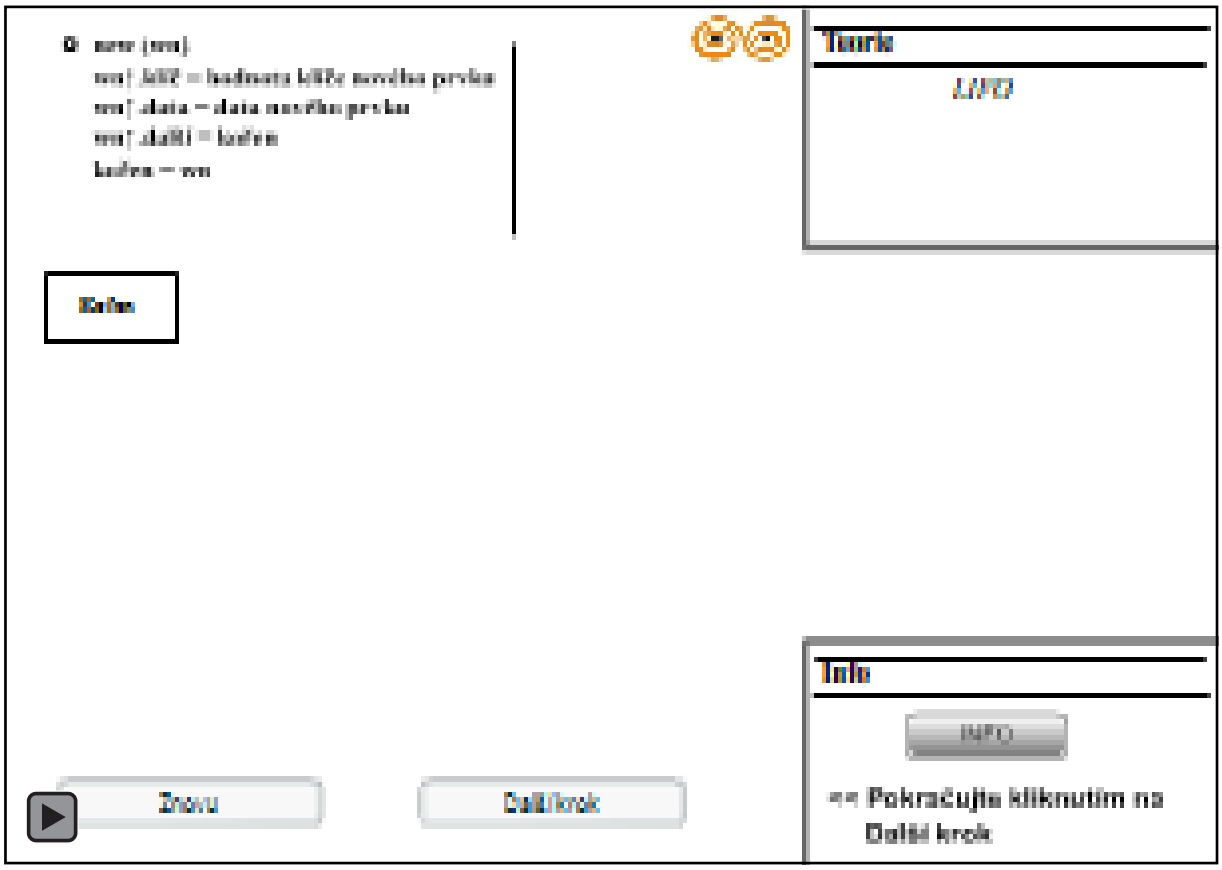
Nejjednodušší operací je vložení nového prvku na začátek seznamu. Kromě proměnné **kořen** potřebujeme jednu pomocnou proměnnou, např. **wn** stejného datového typu.

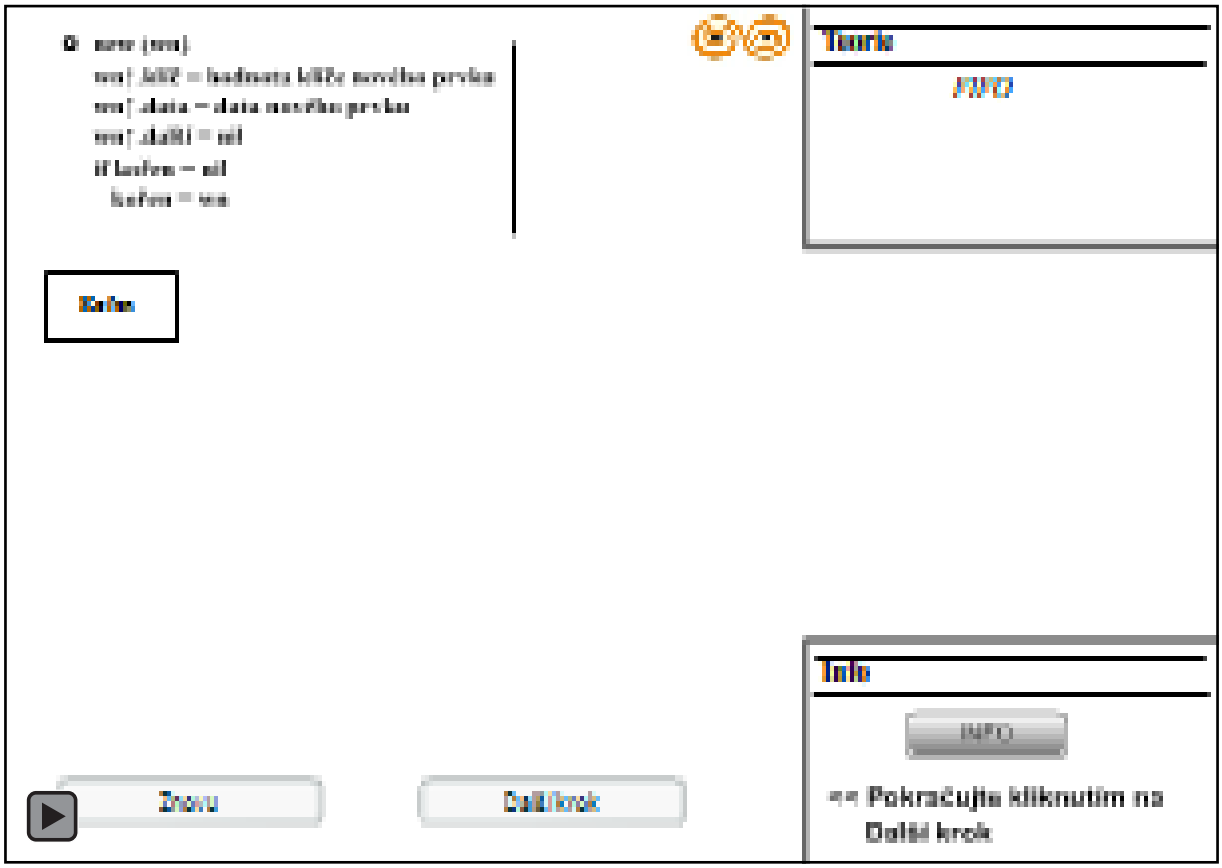
```
new (wn)
```

```
wn↑.klíč = hodnota klíče nového prvku
wn↑.data = data nového prvku
wn↑.další = kořen
kořen = wn
```

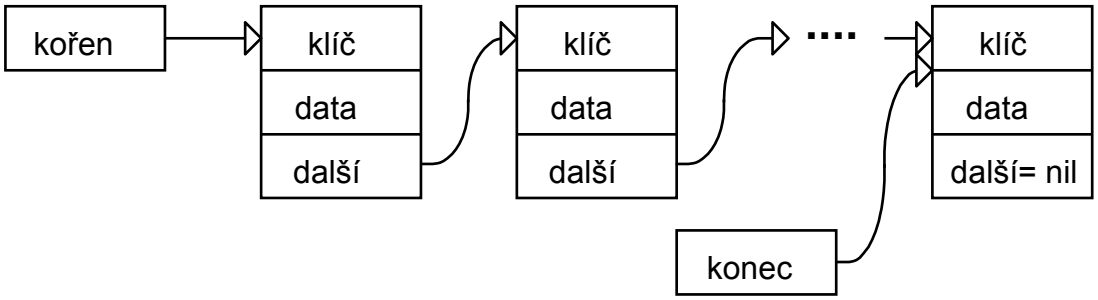
Výsledkem je lineární seznam obsahující prvky seřazené v opačném pořadí, než byly vkládány. K seznamu je přístup jen přes proměnnou **kořen**, takže poslední vložený prvek je první přístupný. Výsledná struktura se nazývá **zásobník**, neboli struktura **LIFO** (z anglického „last in, first out“). Druhou možností je vkládat data na konec seznamu, pak je první vložený prvek také první přístupný. Tato struktura se nazývá **fronta** neboli **FIFO** (first in, first out). Při vkládání na konec seznamu musíme řešit samostatně vkládání prvního prvku do seznamu, neboť má výlučné postavení dané odkazem proměnné **kořen**.

```
new (wn)
wn↑.klíč = hodnota klíče nového prvku
wn↑.data = data nového prvku
wn↑.další = nil
if kořen = nil
  kořen = wn
else
  w = kořen
  while not w↑.další = nil
        w = w↑.další
  end while
  w↑.další = wn
end if
```





Z výpisu je zřejmé, že je nutné vždy dojít na konec seznamu, abychom mohli vložit nový prvek, k tomu jsme použili pomocnou proměnnou **w**. Celý postup výrazně zrychlíme zavedením proměnné **konec**, která bude stále ukazovat na poslední prvek.



Obrázek 4.3 Dynamický seznam s ukazateli na začátek a konec

Postup ukládání nového prvku na konec seznamu se pak výrazně zjednoduší a zrychlí.

```
new (wn)
wn↑.klíč = klíč
wn↑.data = data
wn↑.další = nil
if kořen = nil
  kořen = wn
  konec = wn
else
  konec↑.další = wn
  konec = wn
end if
```

4.1.2 Vyhledávání a vkládání prvků

V řadě aplikací nejprve hledáme prvek v seznamu a vkládáme ho, jen pokud v seznamu dosud není. Obvykle nový prvek vkládáme na konec seznamu. Při vyhledávání ukončíme činnost pokud nastane jedna ze dvou skutečností

nalezení hledaného prvku, kdy w↑.klíč = nový klíč,

nalezení konce seznamu, kdy w = nil.

V druhém případě budeme přidávat nový prvek na konec seznamu, jenže nám chybí odkaz na poslední prvek seznamu. Nejsnáze ho získáme zavedením pomocné proměnné **wk**. Celý postup bude vhodné rozdělit do dvou fází:

Nejprve budeme hledat, zda se prvek v seznamu nachází. Pro tuto informaci přidáme logickou proměnnou **h**. Pokud prvek nenajdeme, uložíme do proměnné **wk** odkaz na poslední prvek seznamu.

Podle výsledku hledání buď provedeme manipulaci s existujícím prvkem seznamu (zvýšení počtu výskytů apod.) nebo přidáme nový prvek do seznamu. Nesmíme zapomenout, že přidáváme prvek na konec seznamu, takže je nutno respektovat zvláštní způsob vložení prvního prvku do seznamu.

```
w = kořen
h = true
while w<>nil and h
  if w↑.klíč = klíč
        h = false
  else
        wk = w
        w = w↑.další
  end if
end while
if h
  new (wn)
  wn↑.klíč = klíč
  wn↑.data = data
  wn↑.další = nil
  if kořen = nil
        kořen = wn
  else
        wk↑.další= wn
  end if
else
  práce s daty w↑.data
end if
```

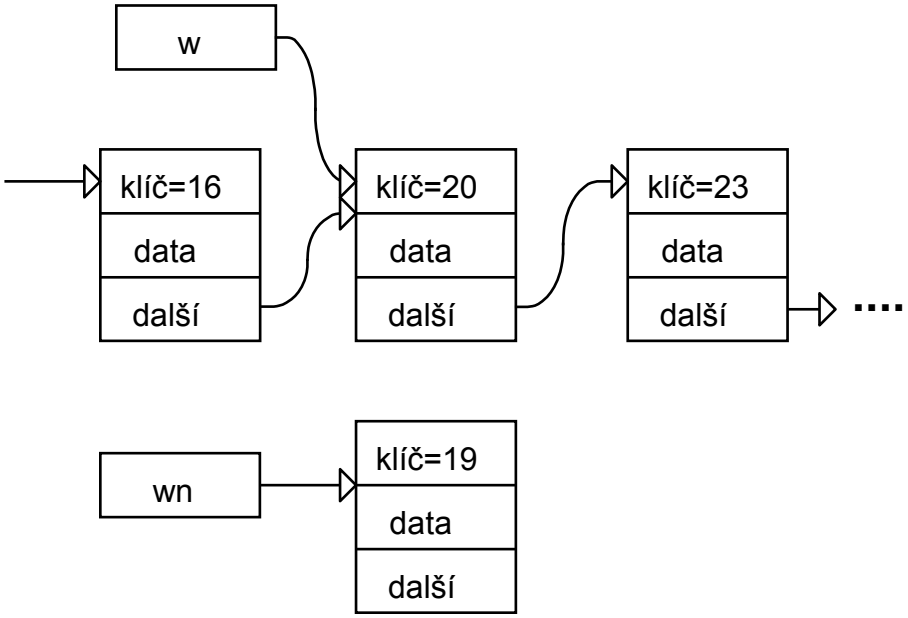
Pokud chceme zjednodušit vkládání, můžeme nový prvek vkládat na začátek seznamu. Vkládání na začátek či konec seznamu ale není právě typické. Daleko častější bude vkládání nového prvku tak, aby indexy prvků v seznamu byly seřazeny (ať již vzestupně nebo sestupně). Přitom může nastat jeden z případů:

je vkládán první prvek do prázdného seznamu,

je vkládán nový prvek na konec seznamu,

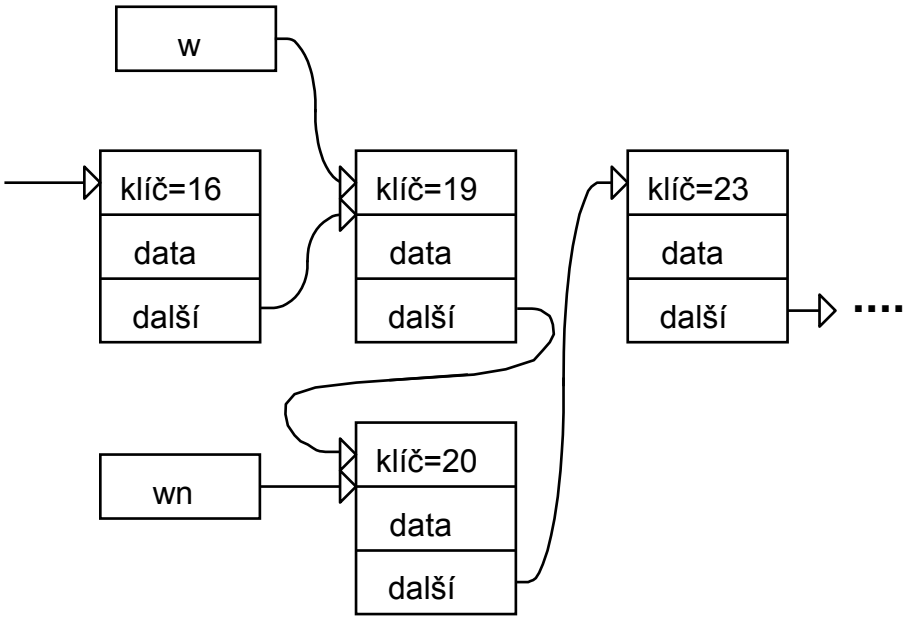
je vkládán nový prvek před nalezený prvek seznamu, neboť nalezený prvek má klíč vyšší hodnoty.

Obsluhu prvních dvou případů známe. Zajímavý je třetí případ. Musíme si uvědomit, že vkládáme nový prvek **před aktuální**! K tomu nám chybí odkaz na předchozí prvek, jehož složka **další** by se měla ukazovat na nově vložený prvek.



Obrázek 4.4 Situace před vložením nového prvku před prvek, na který ukazuje proměnná *w*

To samozřejmě není možné. Zařazení nového prvku dosáhneme jednoduchým trikem. Data proměnné **wn** naplníme daty z proměnné **w**, k proměnné **w** pak vložíme data nového prvku a upravíme ukazatele do výsledné podoby.



Obrázek 4.5 Situace po vložení nového prvku před prvek, na který ukazuje proměnná *w*

Postup vložení nového prvku před prvek w↑:

```
new (wn)
wn↑= w↑                REM přiřazení celé datové struktury
w↑.klíč = klíč
w↑.data = data
```

```
w↑.další = wn
```

Rádi bychom nějak zjednodušili vkládání nového prvku, nejlepší by bylo, kdybychom se dokázali zbavit nutnosti rozeznat jednotlivé případy vkládání (prázdný seznam, dovnitř a na konec seznamu). Na první pohled se to zdá nemožné. Přece však existuje vcelku jednoduchý trik, jak toho dosáhnout. Jmenuje se **zarážka**. Na počátku vložíme do prázdného seznamu ihned jeden prvek zcela libovolného (tedy i nedefinovaného) obsahu, na který bude ukazovat proměnná **zarážka**. Při prohledávání seznamu skončíme nalezením prvku, nebo dosažením situace, kdy prohledávací proměnná w = zarážka.

Postup založení prázdného seznamu se zarážkou

```
new(zarážka)
kořen = zarážka
```

Postup prohledávání a vkládání do seznamu se zarážkou

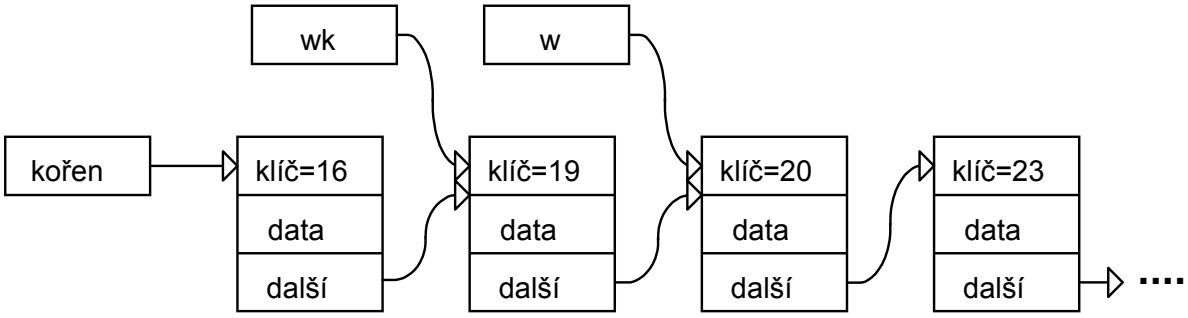
```
w = kořen
h = true
k = true
while h and w<>zarážka
  if w↑.klíč = klíč
        h = false
        k = false
  else
        if w↑.klíč<klíč
              w = w↑. další
        else
              h = false
        end if
  end if
end while
if k
  new (wn)
  wn↑ = w↑
  w↑.klíč = klíč
  w↑.data = data
  w↑.další = wn
  if w = zarážka
        zarážka = w↑.další
  end if
else
  práce s daty w↑.data
end if
```

Jak vidíme, zarážka výrazně zjednodušila algoritmus vkládání nového prvku. Výsledkem po vložení všech prvků je setříděný **indexsekvenční seznam**. Pro vyhledání prvku postupujeme od začátku seznamu až k prvku nebo (pokud neexistuje) na konec seznamu.

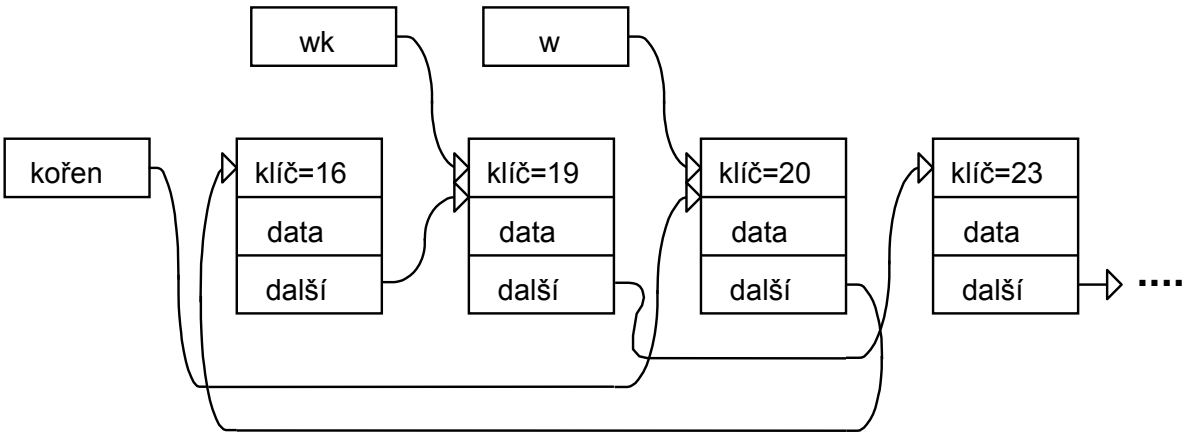
V některých aplikacích bychom rádi vyhledávání v průběhu vkládání zkrátili. Můžeme k tomu využít skutečnosti, že obvykle se jednotlivé prvky vyskytují s různou četností (různě často), v textech mají navíc jednotlivá slova tendenci ke shlukování, tedy opakují se v textu blízko sebe. Zrychlení vyhledávání pak dosáhneme, pokud při zpracování nalezených dat provedeme přesun těchto dat na začátek seznamu. Pokud tam náhodou jsou, tak se nic neděje. Nový prvek vkládáme rovněž na začátek seznamu.

```
w = kořen
```

```
h = true
while w<> nil and h
  if w↑.klíč = klíč
        h = false
  else
        wk = w
        w = w↑.další
  end if
end while
if h
  new (wn)
  wn↑.klíč = klíč
  wn↑. data = data
  wn↑. další = kořen
  kořen = wn
else
  zpracování dat w↑.data
  if w<>kořen
        wk↑.další = w↑.další
        w↑.další = kořen
        kořen = w
  end if
end if
```

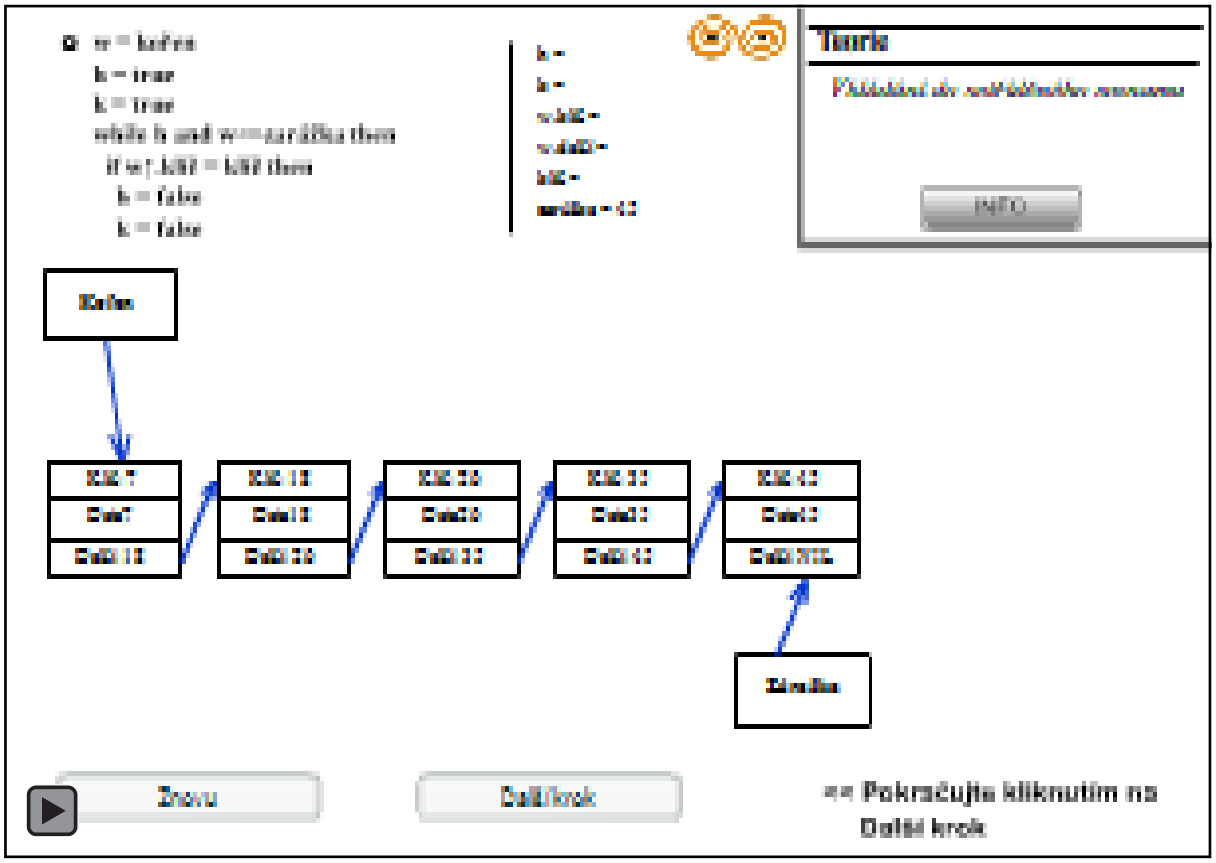


Obrázek 4.6 Situace před přesunem prvku w na začátek seznamu



Obrázek 4.7 Situace po přesunu prvku *w* na začátek seznamu

Budeme-li chtít dále zrychlit vyhledávání prvku, pak již musíme opustit **lineární seznamy** a přejít k **vyhledávacím stromům**, ale o těch se píše o kousek dále.



4.1.3 Práce se seznamem

Zpracování vytvořeného seznamu je možné prakticky jen postupným procházením seznamu od jeho kořene. Jednotlivé prvky tedy budou zpracovány v pořadí svého umístění v seznamu. Pokud bychom chtěli např. zpracovat prvky od posledního směrem k prvnímu, mohli bychom s výhodou využít **proceduru** (programovou rutinu) s **rekurzívním voláním**, kdy procedura volá sama sebe. Některé programové jazyky takové volání nepřipouštějí a jak záhy ukážeme, pro lineární seznam není tento postup vhodný.

```
průchod(w)
  if w<> nil
        průchod (w↑.další)
        zpracování dat w↑.data
  end if
end průchod
```

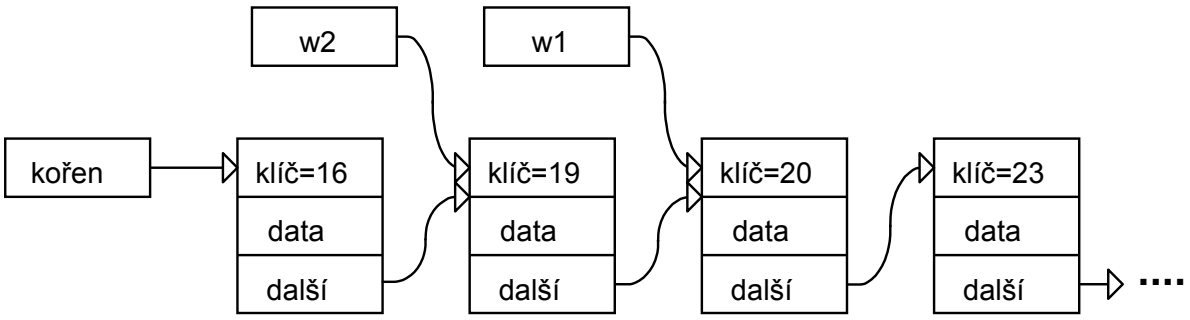
Proceduru budeme volat **průchod(kořen)**. Vidíme, že postupným voláním dojde procedura na konec seznamu a při návratu začne zpracovávat data jednotlivých prvků seznamu.

Vše vypadá velmi krásně, leč rekurzívní algoritmus v sobě nese skryté a to zákeřnější nebezpečí. Tím je **programový zásobník** (*stack*). Je to vyhrazená část paměti, do které se ukládají informace nutné pro návrat z procedury (obsah registrů procesoru, návratová adresa) a vytvářejí se v ní také lokální proměnné procedury a parametry předávané hodnotou. Pokud je seznam delší, hrozí nebezpečí, že vyhrazený paměťový prostor pro zásobník bude nedostatečný, takže dojde k již několikrát zmiňovanému **přetečení zásobníku**. To je závažná chyba, která má obvykle za následek havárii programu. Proto raději používáme **iterační algoritmus** průchodu seznamem, který zpracuje prvky v pořadí od začátku seznamu.

```
w = kořen
while w<>nil
  zpracování dat w↑.data
```

```
    w = w↑.další
  end while
```

Může se stát, že seznam setříděný podle daného klíče potřebujeme setřídit podle jiného **klíče** (jiné složky dat). Přitom máme jen omezený přístup k prvkům seznamu – vždy přes kořen seznamu. Proto můžeme použít jen nejjednodušší a nejméně efektivní metody třídění, jako je **třídění přímým vkládáním** a **třídění přímým výběrem**. Při jejich aplikaci je vhodné brát prvky z jednoho seznamu a vkládat do druhého. Pokud chceme třídit přímo v seznamu, pak můžeme použít **třídění přímou výměnou** známé také jako **bublinkové třídění** (*buble sort*). Při něm porovnáváme hodnoty dvou sousedních prvků a v případě potřeby je vyměníme. Následující program je aplikací tohoto algoritmu při setřídění lineárního seznamu podle hodnoty klíče. Pracuje se dvěma ukazateli **w1** a **w2**, výměna obsahu prvků probíhá pomocí proměnné **wp**. Nesmíme zapomenout, že k tomu, abychom mohli vyměnit pouze odkazy na oba prvky, nám chybí přístup k prvku, který předchází zkoumanou dvojici v seznamu.



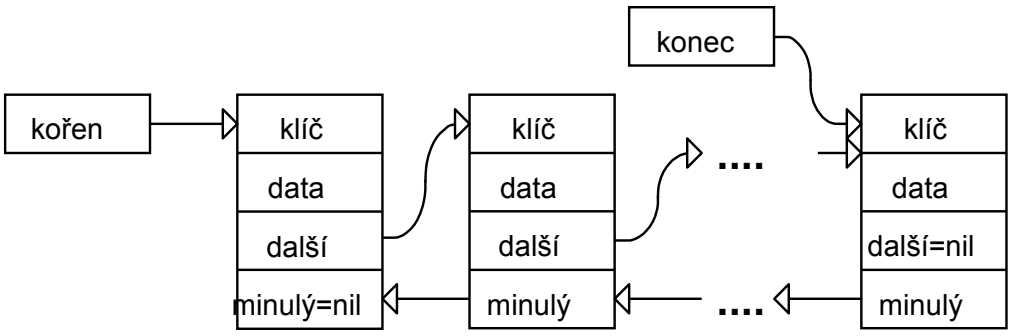
Obrázek 4.8 Situace před výměnou obsahu dvojice prvků v seznamu

K ukončení algoritmu je použita logická proměnná **h**, která oznamuje, zda při průchodu seznamem došlo k výměně.

```
new(wp)
h = true
while h
  h = false
  w1 = kořen
  if w1 <> nil
        w2 = w1
        w1 = w1↑.další
  end if
  while w1<> nil
        if w2↑.klíč>w1↑.klíč
              wp↑.klíč = w2↑.klíč
              wp↑.data = w2↑.data
              w2↑.klíč = w1↑.klíč
              w2↑.data = w1↑.data
              w1↑.klíč = wp↑.klíč
              w1↑.data = wp↑.data
              h = true
        edn if
        w2 = w1
        w1 = w1↑. další
  end while
end while
dispose(wp)
```

Pokud bychom chtěli realizovat podobné **třídění přetřásáním** (*shake sort*) při kterém procházíme seznam střídavě odpředu a odzadu, pak narazíme na zásadní problém. Pro postup odzadu nám chybí informace o předešlém prvku seznamu. Pokud tuto informaci přidáme, získáme **obousměrný seznam**. Vkládání prvků je daleko složitější, nejčastěji se v průběhu vkládání odkazy na předchozí prvky ignorují a nastaví se až po skončení vkládání najednou následujícím algoritmem:

```
wp = nil
w = kořen
while w<>nil
  w↑.minulý = wp
  wp = w
  w = w↑.další
end while
```



Obrázek 4.9 Obousměrný seznam

Výkonnější algoritmy třídění nelze realizovat ani při použití obousměrného seznamu, vyžadují totiž přímý přístup ke všem prvkům seznamu. Z tohoto pohledu je lineární seznam více podobný **sekvenčnímu souboru**, takže je možno na něj aplikovat metody třídění známé ze sekvenčních souborů. Výhody dynamických datových struktur se projevují zejména ve speciálních aplikacích. Široce se využívají např. u metod z teorie grafů, u topologických třídění apod., viz např. [8].

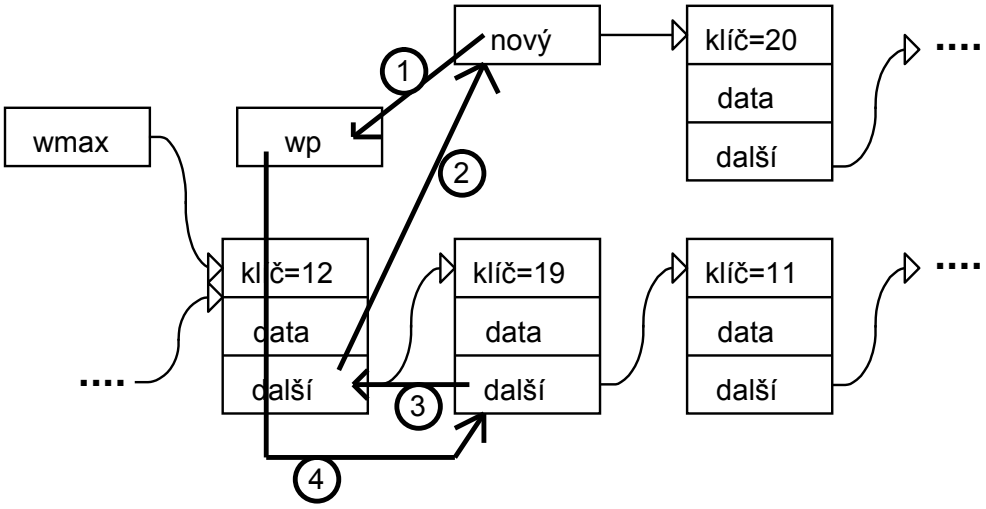
4.1.4 Rušení prvků

Lineární seznam obvykle rušíme celý. Nejjednodušší je rušení prvků na začátku seznamu.

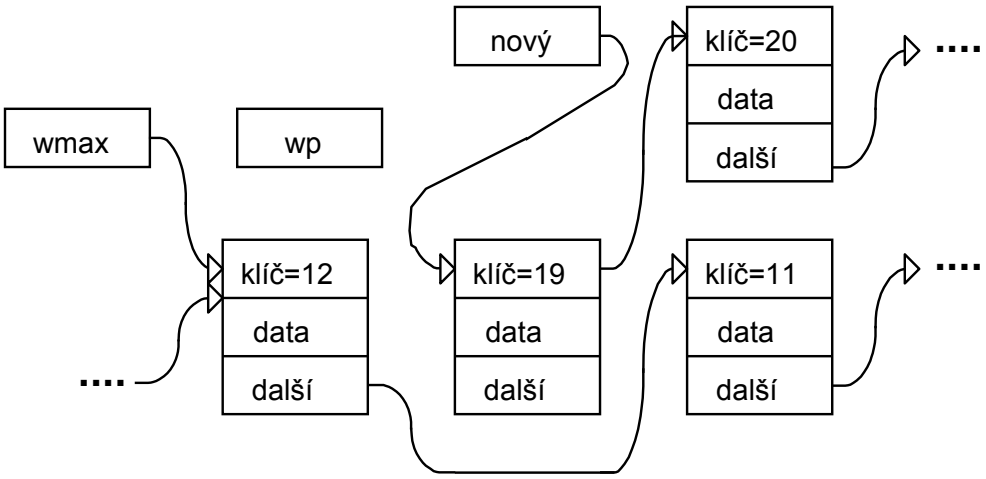
```
while kořen <> nil
  wp = kořen
  kořen = kořen↑.další
  dispose(wp)
end while
```

Všimněte si, že poprvé provádíme průchod seznamem s využitím proměnné **kořen**, u dříve uvedených algoritmů to nebylo přípustné, neboť bychom ztratili přístup k začátku seznamu.

Rušení prvků uvnitř seznamu je složitější, protože musíme zachovat vazby posloupnosti prvků seznamu. Nejjednodušší operací je vynětí následníka prvku, na který ukazuje proměnná **wmax** a jeho přesunutí na začátek jiného seznamu. K přesunu adres přitom použijeme pomocnou proměnnou **wp**.



Obrázek 4.10 Situace před přesunutím následníka prvku, na který ukazuje *wmax*, na začátek seznamu proměnné *nový*, včetně pořadí přesunu informace



Obrázek 4.11 Situace po přesunu prvku na začátek nového seznamu

Při aplikaci uvedeného postupu nesmíme zapomenout, že takto nelze přesouvat první prvek původního seznamu. Upravit algoritmus po přesunu prvního prvku jistě nebude pro čtenáře náročnou operací. Následující algoritmus používá popsaný postup pro třídění seznamu **metodou přímého výběru**. V původním seznamu najdeme prvek s maximální hodnotou klíče (zapamatujeme si adresu jeho předchůdce) a přesuneme ho na začátek nového seznamu. Nový seznam pak bude seřazen vzestupně.

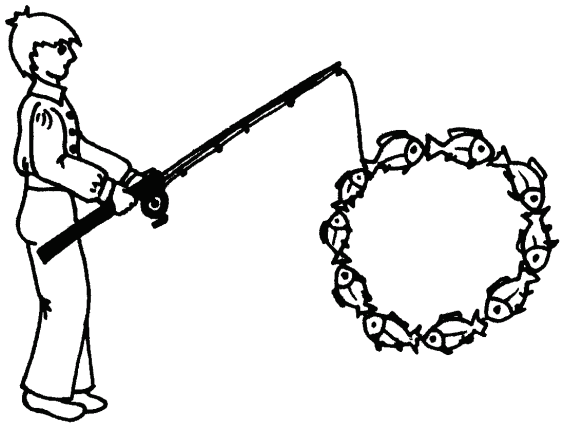
```
nový = nil
while kořen <> nil do
  w1 = kořen
  w2 = nil
  wmax = w2
  kmax = w1↑.klíč
  while w1<>nil
        if w1↑.klíč >= kmax
              wmax = w2
              kmax = w1↑.klíč
        end if
        w2 = w1
        w1 = w1↑.další
```

```
    end while
    if wmax <>nil
          wp = nový
          nový = wmax↑.další
          wmax↑.další = nový↑.další
          nový↑.další = wp
    else
          wp = nový
          nový = kořen
          kořen = nový↑.další
          nový↑.další = wp
    end if
end while
kořen = nový
```

Pokud chceme rušit přímo prvek, na který ukazuje proměnná wmax, pak můžeme využít stejný trik, jako při vkládání nového prvku před existující prvek. Musíme si ale uvědomit, že tímto způsobem nelze zrušit poslední prvek seznamu. Tento nedostatek můžeme opět vyřešit použitím zarážky na konci seznamu.

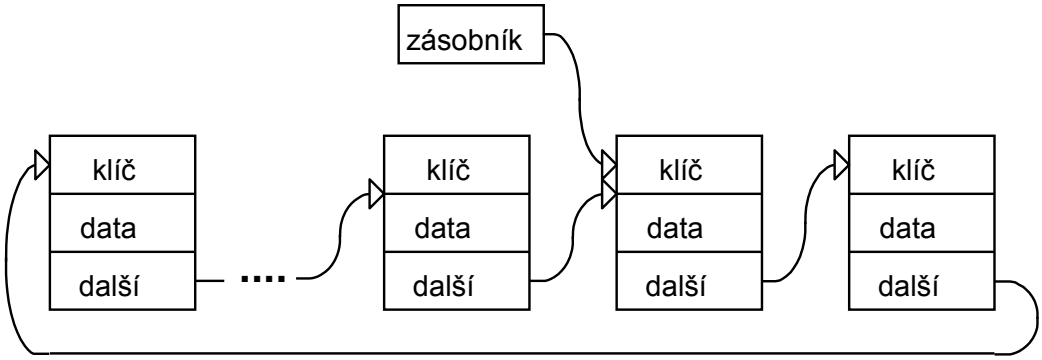
```
wp = wmax↑.další
wmax↑ = wp↑
if wp = zarážka
  zarážka = wmax
end if
dispose(wp)
```

4.1.5 Kruhový zásobník



Obrázek 4.12 Kruhový zásobník

Jak již bylo uvedeno, **zásobník** je datová struktura, do které vkládáme prvky na začátek seznamu a čteme opět od začátku. Poslední vložený prvek je tedy zpracován první (struktura **LIFO**). Pod pojmem **kruhový zásobník** se však skrývá trochu jiná datová struktura. Má obvykle pevný počet prvků, u nichž postupujeme tak, že přečteme hodnotu prvku, nahradíme ji novou a přesuneme se na následující prvek. Protože jsou prvky seřazeny v kruhu, má následující prvek vždy aktuálně nejstarší hodnotu.



Obrázek 4.13 Kruhový zásobník

Kruhový zásobník pro n prvků (n ≥ 1) vytvoříme algoritmem

```
new (zásobník)
w1 = zásobník
inicializace dat w1.data
i = 1
while i< n
  new(w1↑.další)
  w1 = w1↑.další
  inicializace dat w1↑.data
  i = i+1
end while
w1↑.další = zásobník
```

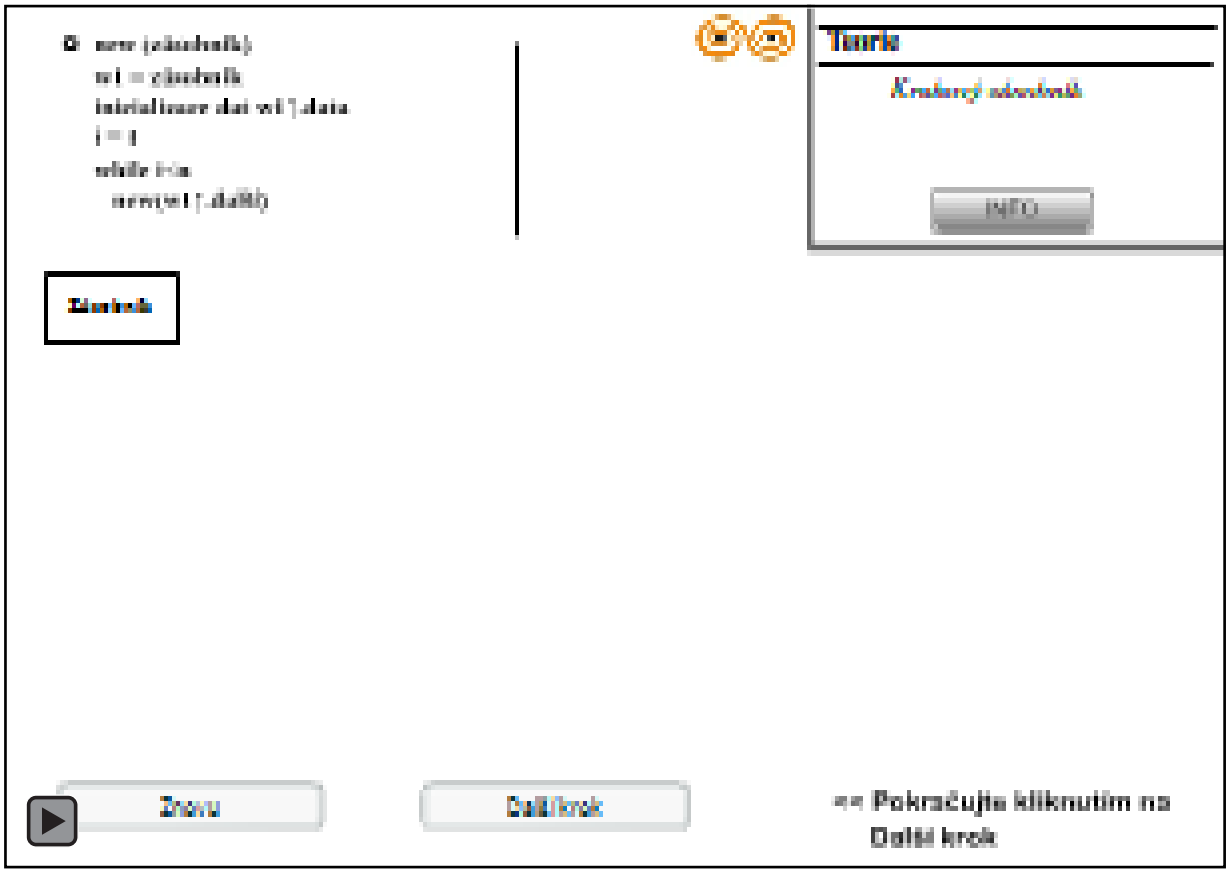
vlastní průchod zásobníkem pak probíhá následovně

```
přečtení dat zásobník↑.data
uložení nových dat zásobník↑.data
zásobník = zásobník↑.další
```

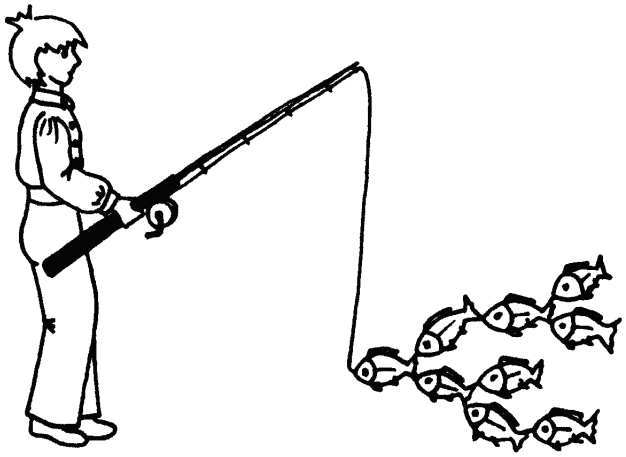
Zrušení kruhového zásobníku provedeme takto (pokud má zásobník alespoň jeden prvek). Test na konci opakování je důležitý pro správnou činnost pokud má zásobník právě jeden prvek. Pokud máme jistotu, že kruhový zásobník má alespoň dva prvky, můžeme test přesunout na začátek opakování:

```
w1 = zásobník
repeat
  wp = w1
  w1 = w1↑.další
  dispose(wp)
until w1 = zásobník
```

Kruhový zásobník se využívá v některých speciálních algoritmech, jako je komprese dat nebo realizace zpoždění o daný počet kroků apod.



4.2 STROMY



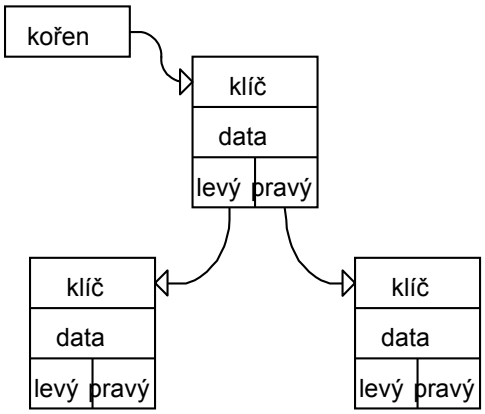
Obrázek 4.14 Stromová struktura

Název datové struktury ***strom*** byl zvolen pro grafickou analogii se skutečnými stromy, s jejich rozvětvováním od kmene přes větve až po listy. Základním kamenem stromu je datová struktura ***prvek*** (uzel, vrchol), která kromě vlastní informace obsahuje konečný počet připojených stromových struktur stejné datové struktury (datového typu) – ***podstromů***.

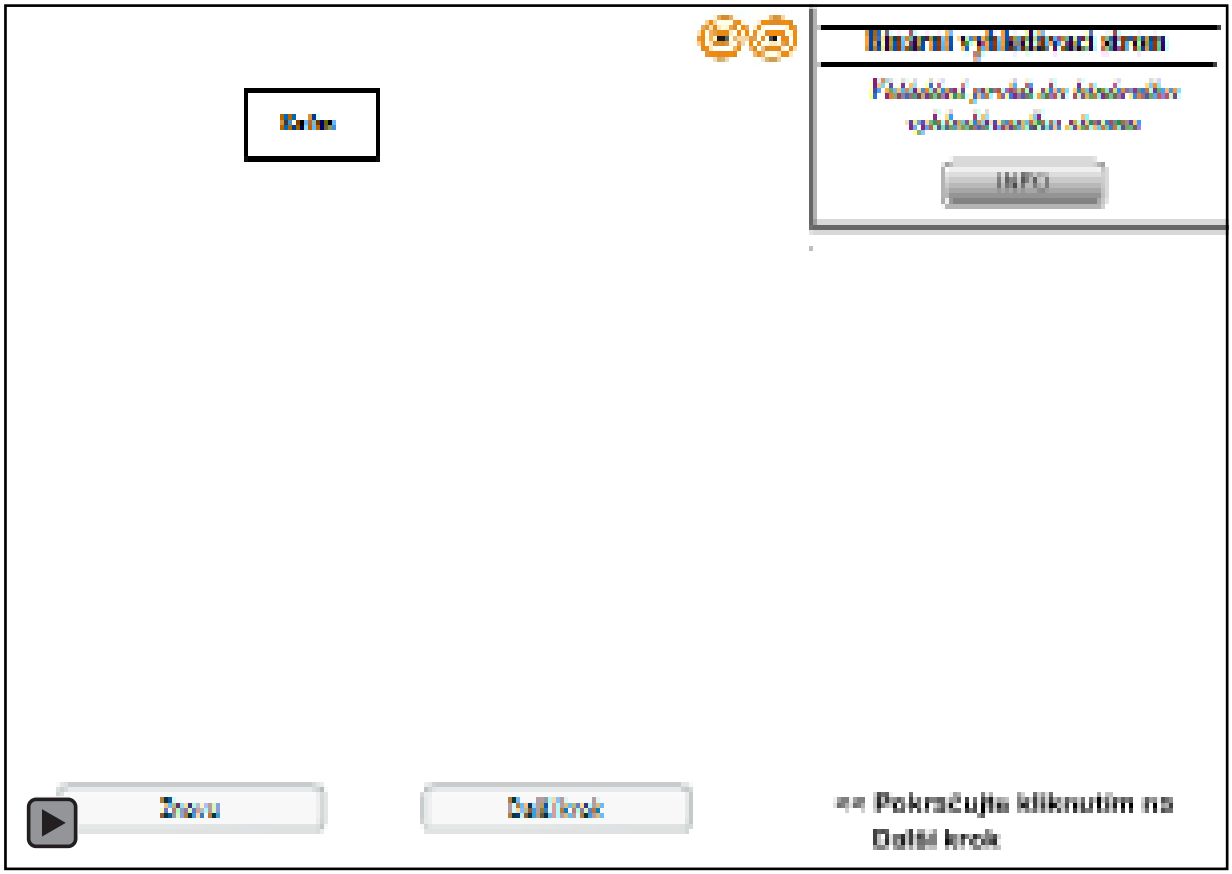
Z uvedené definice je zřejmé, že ***lineární seznam*** můžeme chápat také jako strom, ve kterém má každý prvek nejvýše jeden podstrom. Většinou ho však chápeme jako ***degenerovaný strom***, jak uvidíme dále, může k jeho vytvoření vlivem nepříznivé situace dojít. My se však budeme dále zabývat stromy s více než jedním podstromem. Maximální počet podstromů, který se u prvků stromu vyskytuje, přitom nazýváme ***stupeň stromu***. Pokud tedy žádný prvek

nemá více než dva podstromy, jedná se o strom druhého stupně. Tento typ stromu má v praxi velmi časté použití, proto v dalším textu této kapitoly budeme popisovat výhradně pouze stromy druhého stupně. Pro jejich označení se často používá pojem ***binární stromy***.

Vlastní data každého prvku rozdělíme na informaci o ***klíčové položce*** (klíči, indexu), podle které budeme prvky řadit a zbytek dat. Odkazy na podstromy budeme v souladu se zvyklostmi a analogií grafického znázornění stromu označovat jako levý a pravý podstrom.



Obrázek 4.15 Strom



Základním prvkem stromu je ***kořen***. Liší se od ostatních prvků stromu tím, že nemá žádného ***předchůdce***, ale jen nejvýše dva přímé ***následníky*** (levý a pravý podstrom). Tedy prvky, které na něj bezprostředně navazují. Obecně je ***následník*** takový prvek, který leží v některém podstromu daného prvku. Prvky, které nemají žádné následníky, označujeme za ***koncové prvky***, nebo také ***listy***. Ostatní prvky, které nejsou ani koncové ani kořen, nazýváme ***vnitřní***

prvky. Jako ***prázdný strom*** pak označujeme takový strom, který nemá ani kořen (proměnná ***kořen = nil)***

Stromy většinou zobrazujeme tak, že se rozvíjejí od kořene směrem dolů (označení koncových prvků jako listy pak působí poněkud překvapivě, ale označení kořínky či podobné je matoucí a nepoužívá se). Prvky, které mají stejný počet předchůdců, kreslíme ve společné vodorovné linii. Říkáme, že tyto prvky leží na stejné ***úrovni*** (úrovni vnoření do stromu). Kořen je přitom na 1. úrovni, jeho přímí následníci pak na 2. úrovni atd. Maximální úroveň, které dosahuje některý prvek stromu, nazýváme ***hloubka stromu*** (méně často výška stromu).

Dalším významným pojmem je ***délka cesty*** k prvkům. Je to počet spojnic prvků (hran) nebo prvků (uzlů, vrcholů), které musíme projít na nejkratší cestě ke kořenu stromu. Pro co nejrychlejší přístup k prvku je třeba, aby každý měl co nejkratší cestu. Pokud sečteme délku cest všech prvků, dostaneme ***délku cesty stromu*** (délku vnitřní cesty stromu). Je zřejmé, že ***minimální délku*** bude mít strom tehdy, jestliže na každé úrovni (kromě poslední) bude mít maximální možný počet prvků. Takový strom má současně ***minimální hloubku*** (výšku) takže při použití rekurzívních algoritmů je nejmenší nebezpečí přetečení zásobníků.

Základní manipulace se stromy je opět vkládání, vyhledávání, a rušení prvků. I když své uplatnění mají i statické stromy, budeme se zabývat především dynamickými stromy, jejichž struktura se během zpracování mění. Vzhledem k vlastnostem stromů, které lze definovat rekurzívně, je možno velmi úspěšně řešit operace na stromech pomocí rekurzívních algoritmů. Na známé nebezpečí ***rekurze*** (přetečení zásobníku) přitom samozřejmě nezapomínáme.

Stromy mají řadu aplikací, například ***rozhodovací stromy***, u kterých uzly představují dílčí řešení. Jejich aplikací je strom Huffmanova kódu, který se používá jako jedna z metod komprese dat. V dalším textu budeme rozebírat ***vyhledávací stromy,*** které se využívají zejména při třídění dat.

4.2.1 Binární vyhledávací stromy

Vyhledávací stromy jsou stromy, u kterých zařazujeme prvky podle klíčové položky tak, že levý následník má vždy nižší hodnotu klíče než prvek a pravý následník hodnotu vyšší. Zařazení prvku je obvykle spojeno s jeho vyhledáním a zařazuje se jen tehdy, pokud nebyl nalezen.

Algoritmus ***vyhledání a vložení prvku*** do stromu budeme realizovat rekurzívní procedurou. Předávat jí budeme odkaz na aktuální prvek stromu. Protože budeme do stromu také vkládat, musíme předávat skutečně odkaz na prvek a ne pouze jeho hodnotu.

```
hledej (klíč, REF prvek)
  w = prvek
  if w = nil
        new(w)
        w↑.klíč = klíč
        w↑.data = data
        w↑.levý = nil
        w↑.pravý = nil
  else
        if klíč < w↑.klíč
              hledej (w↑.levý)
        else
              if klíč > w↑.klíč
                    hledej (w↑.pravý)
              else
                    zpracování dat w↑.data
```

```
                end if
          end if
    end if
end hledej
```

Vyhledávání i vkládání bude probíhat rychleji než u lineárního seznamu. Pokud však budou prvky vstupovat již seřazené (ať vzestupně nebo sestupně) degeneruje strom v lineární seznam. Okamžitě se objevuje také nebezpečí přetečení zásobníku při rekurzívním volání procedury ***hledej***. Jak toto nebezpečí odstranit budeme řešit v kapitole věnované ***vyváženým stromům***.

Nalezení konkrétního prvku v binárním stromu, je možno realizovat také iteračním algoritmem. Stejně jako u lineárních seznamů používáme logickou proměnnou ***h*** k ukončení vyhledávání. Po skončení algoritmu obsahuje proměnná ***w*** odkaz na hledaný prvek nebo hodnotu ***nil***, pokud ve stromu hledaný prvek není.

```
h = true
w = kořen
while w <> nil and h
    if w↑.klíč = klíč
          h = false
    else
          if w↑.klíč > klíč
                w = w↑.levý
          else
                w = w↑.pravý
          end if
    end if
end while
```

Uvedený postup je možno použít i při převodu rutiny ***hledej(REF prvek)*** z rekurzívního na iterační tvar.

Sestavený strom budeme často ***procházet*** a ***zpracovávat***. Opět je vhodný rekurzívní algoritmus. Pokud chceme zpracovávat prvky v jejich seřazeném pořadí, můžeme použít následující proceduru.

```
projdi (prvek)
    if prvek <> nil
          projdi (prvek↑.levý)
          zpracování dat prvek↑.data
          projdi (prvek↑.pravý)
    end if
end projdi
```

Tento algoritmus snadno upravíme pro ***zrušení celého stromu***. Je však nutno nejprve projít a zrušit oba následníky (a také části stromu, které na ně navazují, neboli oba podstromy) a pak zrušit tento prvek. Opět nesmíme zapomenout předávat prvky odkazem.

```
zruš (REF prvek)
    if prvek <> nil
          projdi (prvek↑.levý)
          projdi (prvek↑.pravý)
          dispose(p)
    end if
end zruš
```

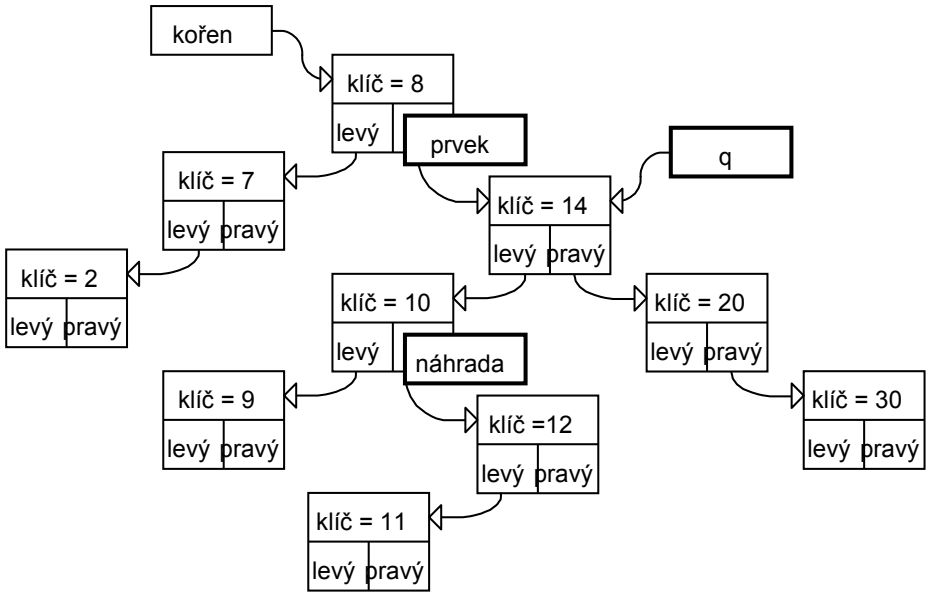
Inverzní problém k vyhledávání a vkládání prvků je ***vyhledávání a rušení prvků***. Jeho řešení je výrazně složitější. Musíme rozlišovat následující případy:

prvek s hledaným klíčem ve stromě není, pak vyhlásíme chybu, nebo prostě neuděláme nic,

prvek s hledaným klíčem má nejvýše jednoho následníka, pak na místo nalezeného prvku přesuneme tohoto následníka (a s ním spojený podstrom),

prvek s hledaným klíčem má dva následníky, pak nalezený prvek nahradíme buď nejpravějším prvkem jeho levého podstromu (nebo nejlevějším prvkem pravého podstromu), viz obr. 4.16.

Všimněte si zejména toho, že díky předávání proměnných odkazem způsobí změna obsahu proměnné také změnu struktury stromu, protože proměnná je totožná s konkrétním ukazatelem některého prvku stromu. Pro zvýraznění byly proměnné algoritmu orámovány tučnou čarou.



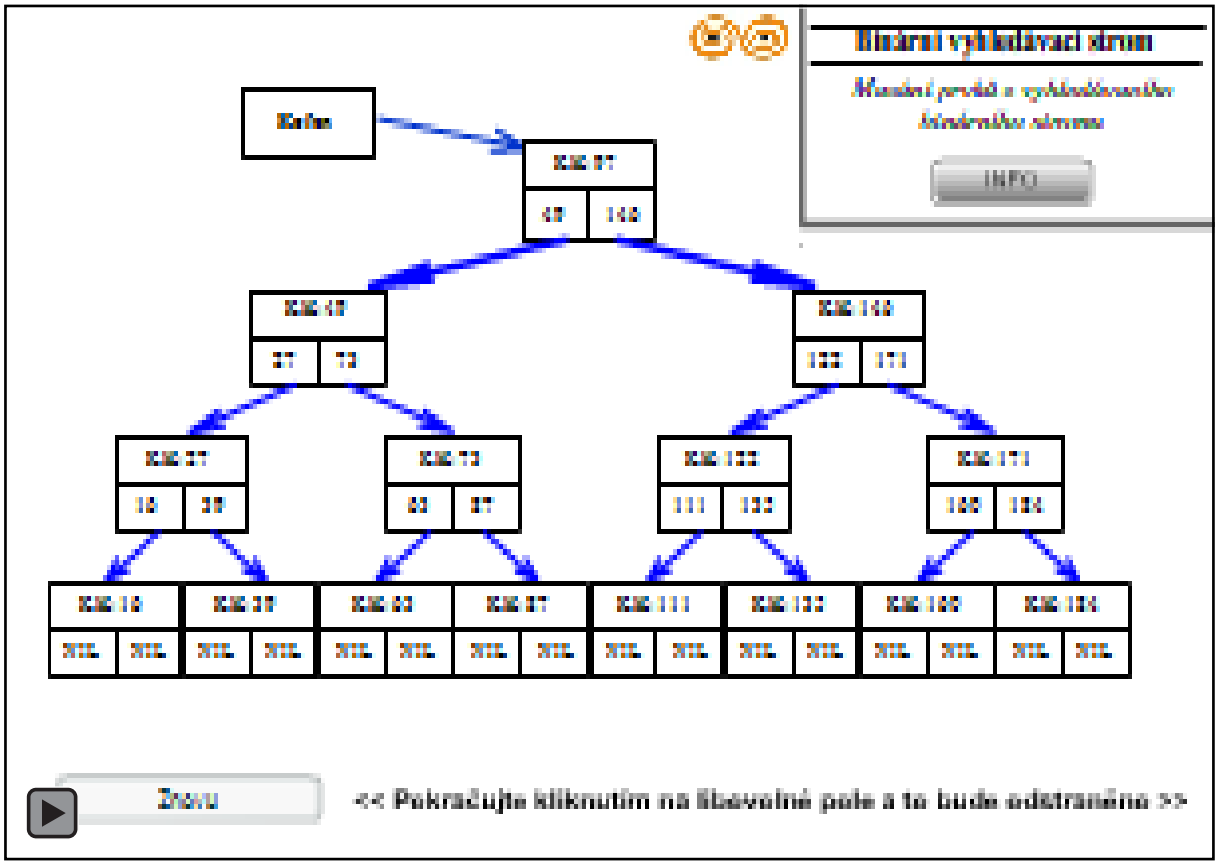
Obrázek 4.16 Stav před zrušením prvku 14 a jeho náhradou prvkem 12

```
vezmi (klíč, REF prvek)
  if prvek = nil
        hledaný prvek ve stromu není
  else
        if klíč<prvek↑.klíč
              vezmi (klíč, prvek↑.levý)
        else
              if klíč>prvek↑.klíč
                    vezmi (klíč, prvek↑.pravý)
              else
                    hledaný prvek byl nalezen
                    q = prvek
                    if q↑.pravý = nil
                          prvek = q↑.levý
                    else
                          if q↑.levý = nil
                                prvek = q↑.pravý
                          else
                                najdi (q↑.levý)
```

```
                        end if
                    end if
                    dispose (q)
                end if
            end if
        end if
    end vezmi
```

Jak vidíme, pro nejsložitější případ je použita procedura ***najdi***, která do prvku ***q*** dosadí nejpravější koncový prvek z levého podstromu prvku ***q*** (proměnná ***q*** tedy musí být deklarována jako globální, jinak ji musíme předávat jako parametr procedury, samozřejmě odkazem).

```
najdi (REF náhrada)
  if náhrada↑.pravý<>nil
        najdi (náhrada↑.pravý)
  else
        q↑.klíč = náhrada↑.klíč
        q↑.data = náhrada↑.data
        q = náhrada
        náhrada = náhrada↑.levý
  end if
end najdi
```



4.2.2 Vyvážené stromy

V předchozí kapitole bylo upozorněno na nebezpečí degenerování ***binárního vyhledávacího stromu*** na ***linenární seznam*** a s tím spojené nebezpečí havárie rekurzívního algoritmu.

Nejlepším řešením by byla konstrukce stromu s ***minimální hloubkou***, neboť hloubka stromu určuje také počet vnoření při volání rekurzívního algoritmu.

Nejjednodušeji toho dosáhneme, když budeme jednotlivé prvky umisťovat rovnoměrně na levou i pravou stranu stromu. Tento postup nazýváme ***strukturování stromu***. Pro známý počet vrcholů n můžeme definovat strom s minimální hloubkou pomocí tří pravidel:

1. zvolíme jeden z prvků jako kořen stromu,
2. vytvoříme levý podstrom s počtem ***nl = n div 2*** prvků.
3. vytvoříme pravý podstrom s počtem ***nr = n - nl -1*** prvků.

Výsledný strom bude ***dokonale vyvážený,*** neboť pro každý prvek platí, že počet prvků v jeho levém a pravém podstromu se liší nanejvýš o 1. V praxi je použití dokonale vyváženého stromu dosti problematické. Obvykle předem neznáme počet prvků a navíc požadujeme tvorbu ***vyhledávacího stromu***, jehož prvky nejsou umístěny náhodně. Pak bychom při každém vložení nového prvku do vyhledávacího stromu (tzv. ***náhodném vkládání***) museli provést restrukturalizaci stromu, která je velmi neefektivní.

Ze slepé uličky se dostaneme, pokud přijmeme méně přísnou definici ***vyváženého stromu***, jak ji formulovali Adelson-Velskij a Landis. Zní: „***Strom je vyvážený právě tehdy, když se výšky dvou podstromů každého prvku liší nejvýše o 1***“. Takto konstruované stromy nazýváme ***AVL-stromy***, my je budeme dále označovat jako vyvážené stromy, neboť se jinými zabývat nebudeme.

Autoři dokázali, že AVL-strom bude v nejhorším případě o 45% vyšší, než dokonale vyvážený strom se stejnými prvky. Přitom je možno se složitostí ***O(log n)*** provést všechny základní operace, tedy vložení, vyhledání i zrušení prvku s daným klíčem.

4.2.2.1 Vkládání prvku do AVL-stromu

Při tvorbě vyváženého stromu budeme postupovat tak, že po vložení nového prvku vždy, když je to nutné, opravíme vyváženost stromu. Vložením nového prvku do levého podstromu konkrétního prvku mohou nastat následující případy:

1. vyváženost se zlepší, pokud výška levého byla menší než u pravého podstromu,
2. vyváženost se zhorší, ale neporuší, pokud byly obě výšky stejné,
3. vyváženost se poruší, pokud výška levého podstromu byla větší než u pravého, vyváženost je třeba upravit.

K opravě vyváženosti stromu slouží ***rotace prvků***. Při nich samozřejmě nesmí dojít k porušení návaznosti klíčů u prvků vyhledávacího stromu.

K rozhodnutí o tom, která situace nastala, potřebujeme informaci o předchozím stavu vyváženosti stromu. Zjišťovat ji přímo ze struktury stromu je značně nepraktické, výhodnější bude doplnit datovou oblast prvků o složku ***vaz*** nabývající hodnot {-1, 0, 1} s následujícím významem:

-1 - levý podstrom je delší,

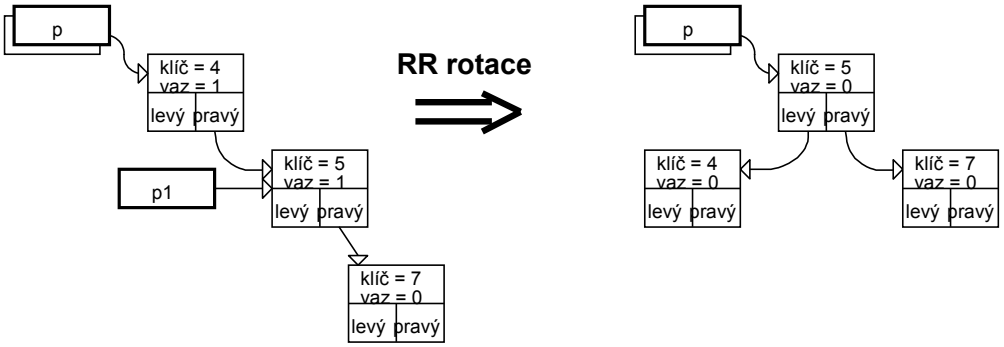
0 - oba podstromy mají stejnou délku,

1 - pravý podstrom je delší.

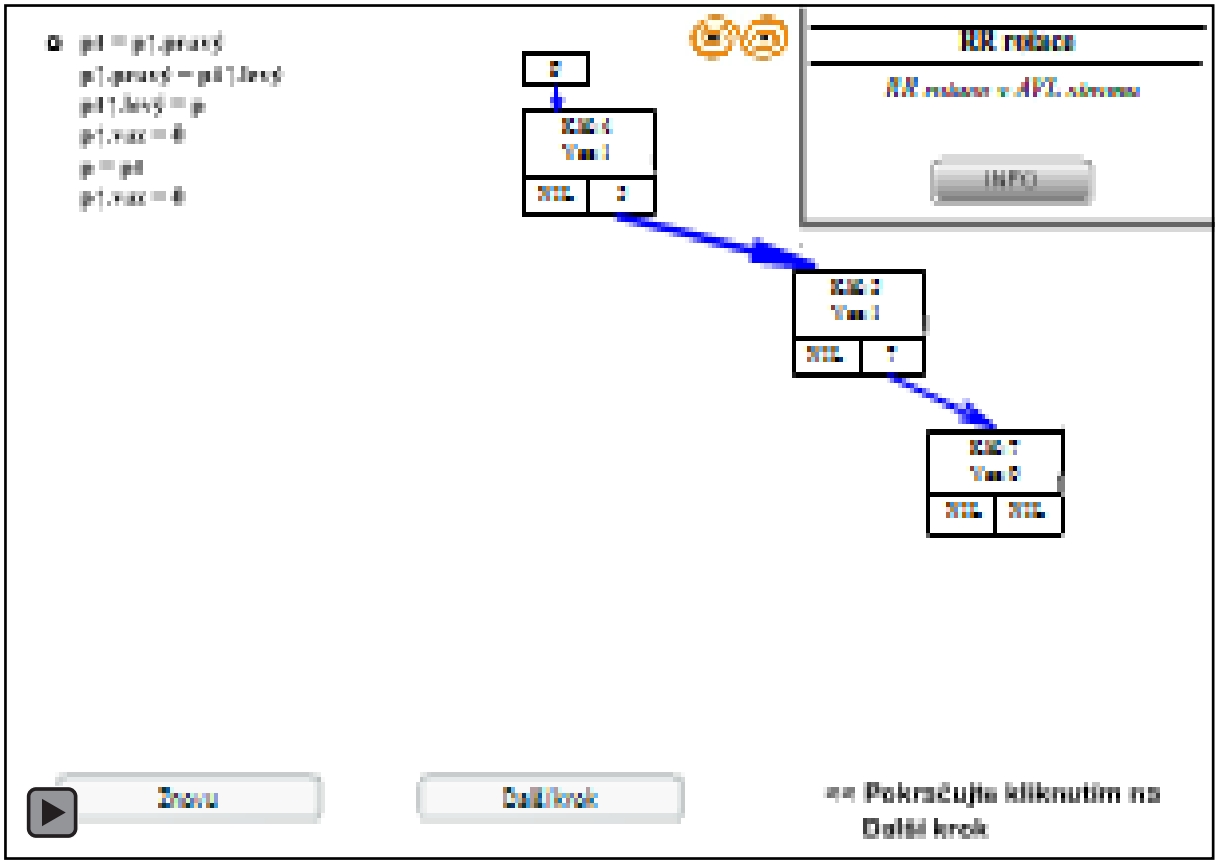
Po vložení nového prvku opravíme hodnotu proměnné ***vaz*** a v případě potřeby opravíme vyváženost stromu. Nesmíme zapomenout na rekurzívní volání, a proto musí rutina vracet informaci o tom, zda se vložením prvku zvětšila délka podstromu nebo ne. Využijeme k tomu logickou proměnnou ***h***, kterou budeme předávat odkazem, stejně jako aktuální prvek.

Ukažme si nyní, jaké případy nevyváženosti mohou nastat a jak je budeme řešit. Pro zjednodušení budeme kreslit jen vazby mezi prvky a u prvků hodnoty jejich klíčů a proměnné ***vaz***. Na prvek, u kterého došlo k porušení vyvážení, bude ukazovat proměnná ***p***, předávaná odkazem, ostatní proměnné budou pomocné.

Do prázdného stromu vložíme nejprve prvek s klíčem 4, ten bude kořenem, pak prvek s klíčem 5. Vložením prvku s klíčem 7 dojde k porušení vyvážení u prvku s klíčem 4. Úpravy dosáhneme jednoduchou RR rotací pravého podstromu.

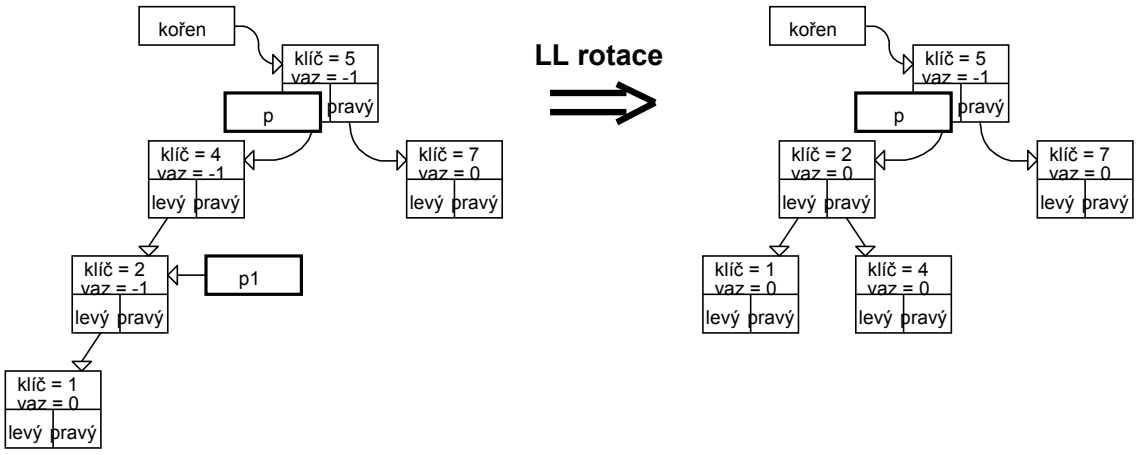


Obrázek 4.17 Rotace RR

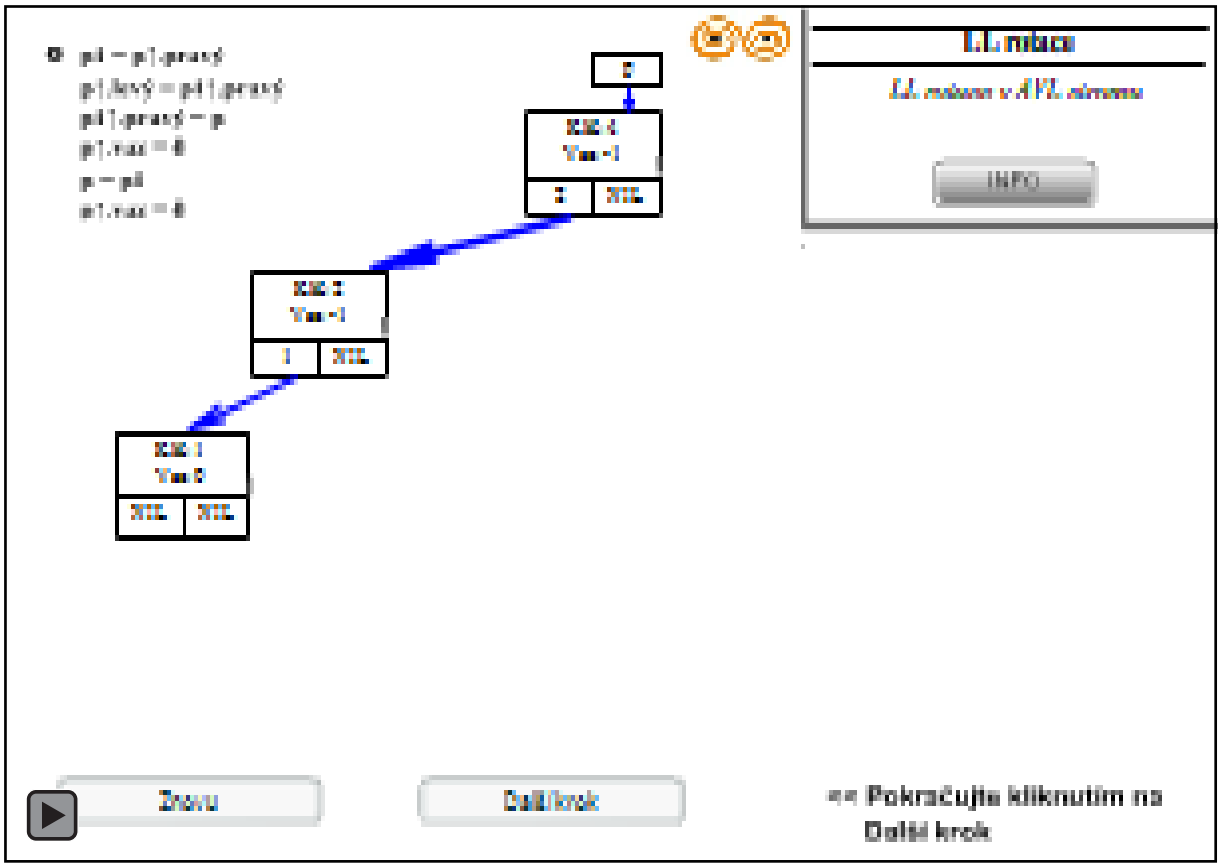



```
p1 = p↑.pravý
p↑.pravý = p1↑.levý
p1↑.levý = p
p↑.vaz = 0
p = p1
p↑.vaz = 0
h = false
```

Dalším vložením prvku s klíčem 2 a pak 1 dojde k podobné situaci v levé větvi opět u prvku s klíčem 4. Opravu dosáhneme jednoduchou rotací LL.

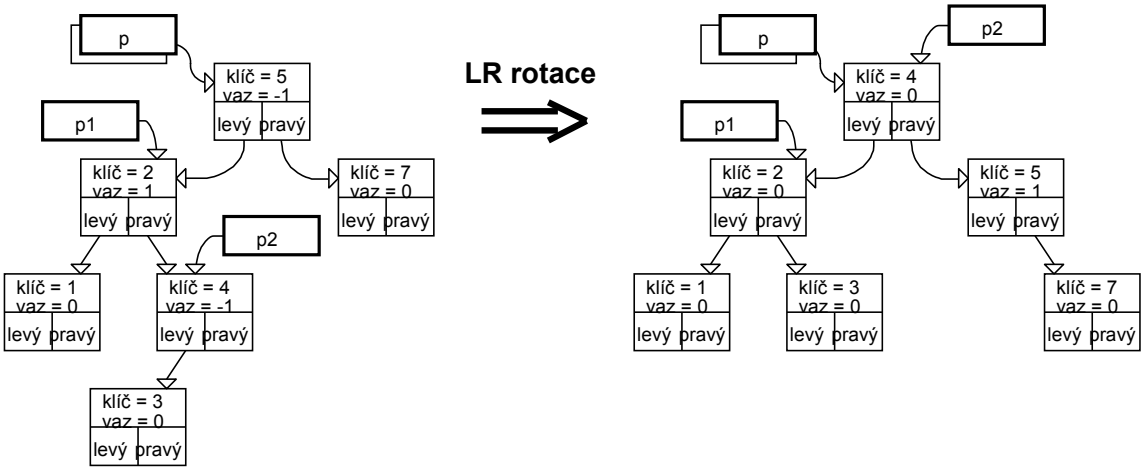


Obrázek 4.18 Rotace LL

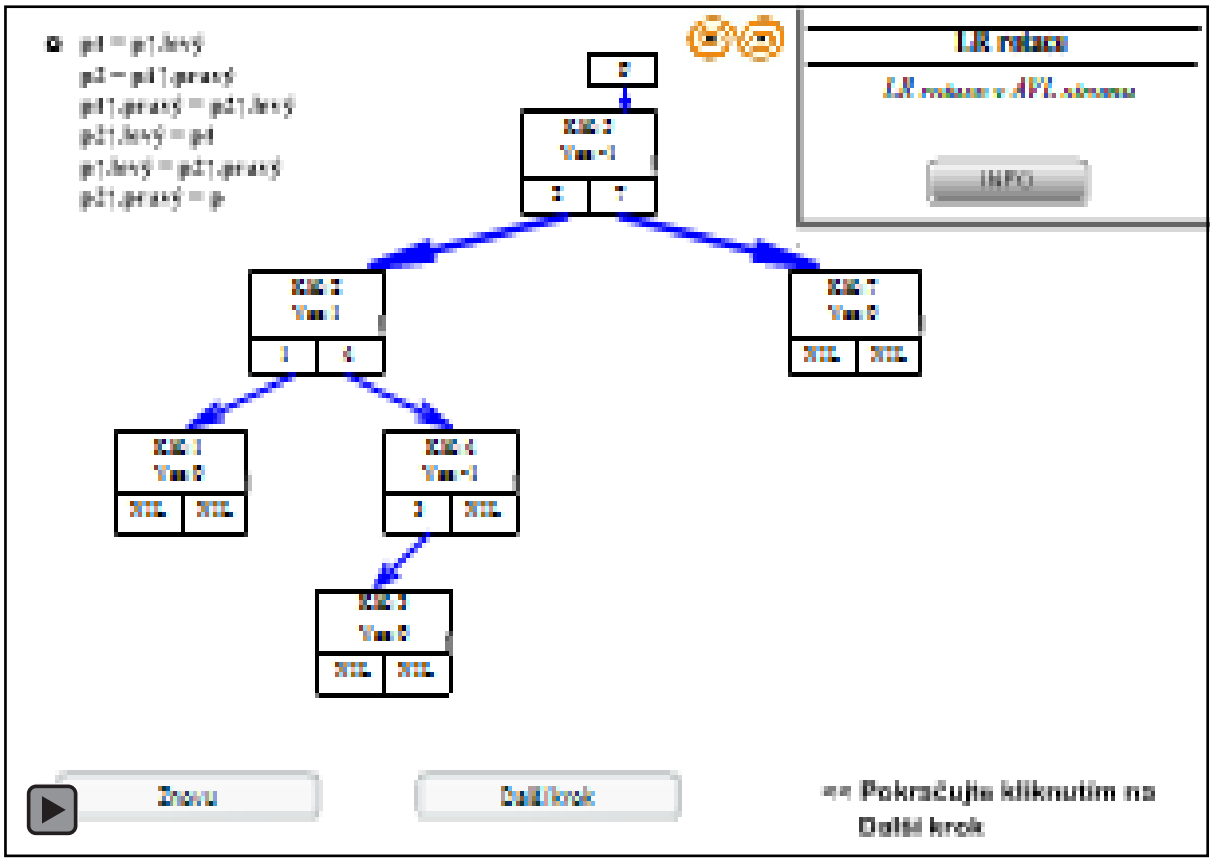



```
p1 = p↑.pravý
p↑.levý = p1↑.pravý
p1↑.pravý = p
p↑.vaz = 0
p = p1
p↑.vaz = 0
h = false
```

Jako další vložíme prvek s klíčem 3. Tím se poruší vyváženost u prvku s klíčem 5. Tentokrát je situace složitější. K opravě je nutná dvojitá LR rotace



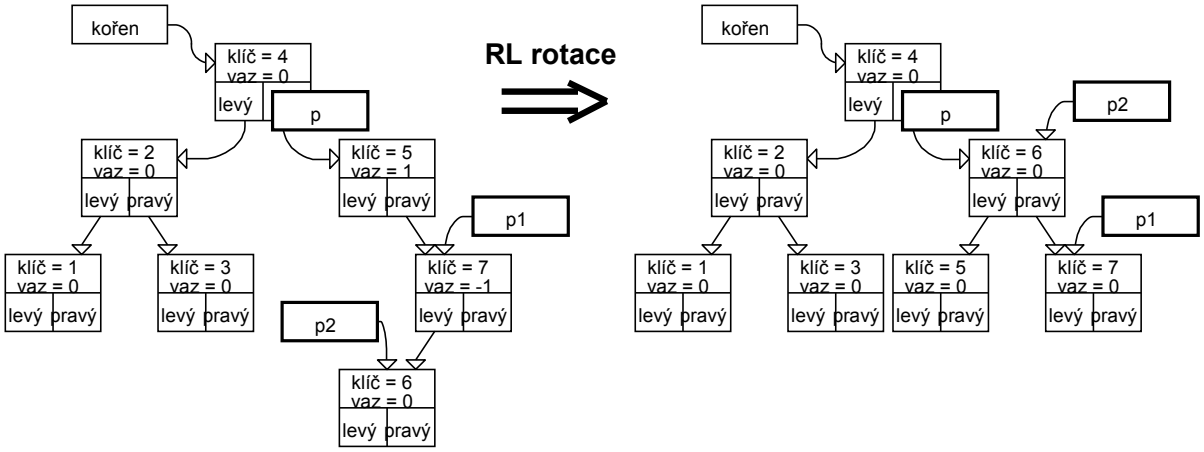
Obrázek 4.19 Dvojitá rotace LR



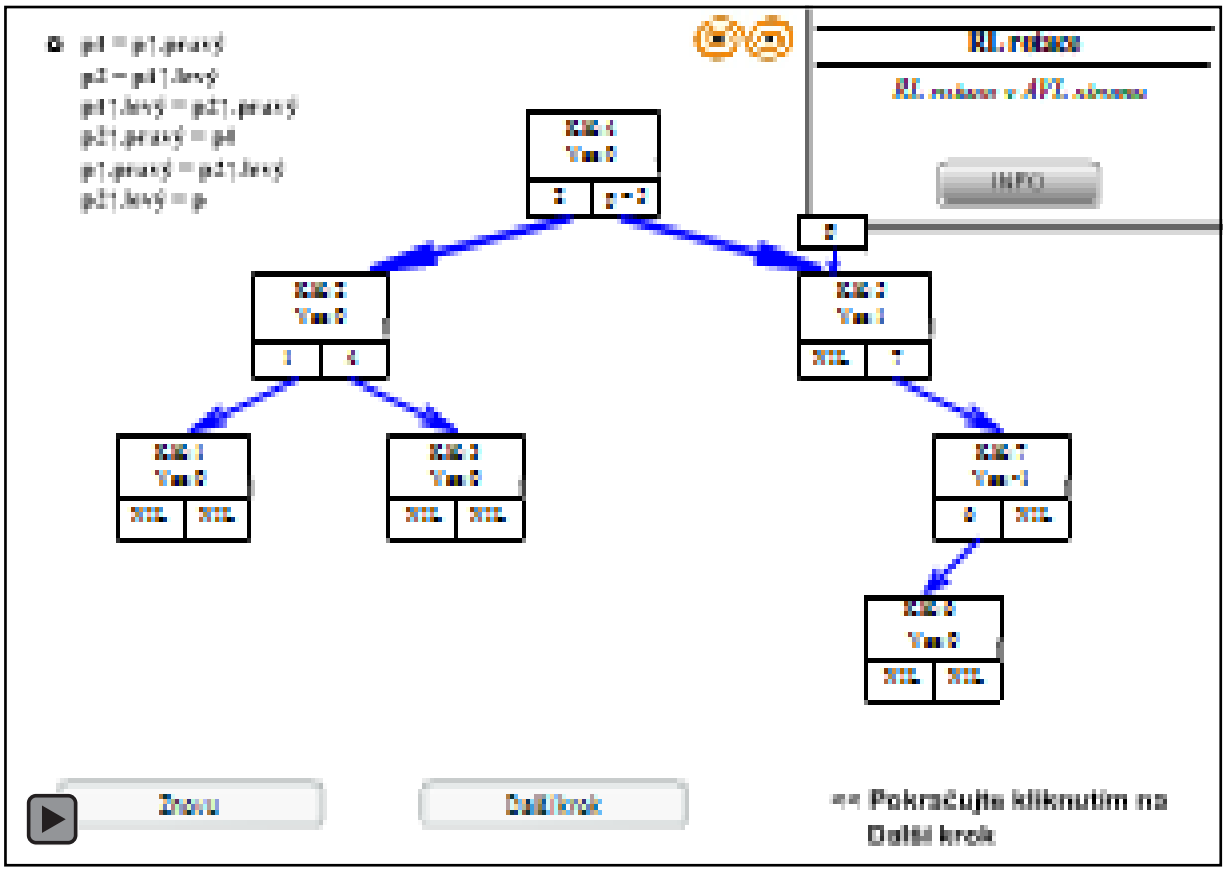

```
p1 = p↑.levý
p2 = p1↑.pravý
p1↑.pravý = p2↑.levý
p2↑.levý = p1
p↑.levý = p2↑.pravý
p2↑.pravý = p
if p2↑.vaz = -1
  p↑.vaz = 1
else
  p↑.vaz = 0
```

```
end if
if p2↑.vaz = 1
  p1↑.vaz = -1
else
  p1↑.vaz = 0
end if
p = p2
p↑.vaz = 0
h = false
```

Poslední úpravou bude vložení prvku s klíčem 6, které způsobí porušení vyvážení u prvku s klíčem 5. Nutná je dvojitá rotace RL.



Obrázek 4.20 Dvojitá rotace RL



```
p1 = p↑.pravý
p2 = p1↑.levý
p1↑.levý = p2↑.pravý
p2↑.pravý = p1
p↑.pravý = p2↑.levý
p2↑.levý = p
if p2↑.vaz = 1
  p↑.vaz = -1
else
  p↑.vaz = 0
end if
if p2↑.vaz = -1
  p1↑.vaz = 1
else
  p1↑.vaz = 0
end if
p = p2
p↑.vaz = 0
h = false
```

Z rozboru jednotlivých situací je zřejmé také rozpoznání jednoduché a dvojité rotace. Pokud mají na počátku rotace prvky ***p*** a ***p1*** stejnou hodnotu ***vaz***, jedná se o jednoduchou rotaci, jinak je potřebná dvojitá rotace. Nyní již můžeme sestavit celou rekurzívní rutinu pro vyhledávání a vkládání prvků do vyváženého stromu. U popisů jednotlivých typů rotací byly orámovány ty části algoritmů, které nyní uvádíme pouze názvem rotace.

```
hledej1 (REF p, REF h)
if p = nil
  vložení nového prvku do proměnné p, h = true
else
  if klíč < p↑.klíč
        hledej1 (p↑.levý,h)
        if h
              case p↑.vaz = 1
                    p↑.vaz = 0
                    h = false
              case p↑.vaz = 0
                    p↑.vaz = -1
              case p↑.vaz = -1
                    p1 = p↑.levý
                    if p1↑.vaz = -1
                          rotace LL
                    else
                          rotace LR
                    end if
                    p↑.vaz = 0
                    h = false
              end case
        end if
  else
        if klíč > p↑.klíč
              hledej1 (p↑.pravý,h)
```

```
                if h
                      case p1↑.vaz = -1
                            p1↑.vaz = 0
                            h = false
                      case p1↑.vaz = 0
                            p↑.vaz = 1
                      case p1↑.vaz = 1
                            p1↑.vaz = pravý
                            if p1↑.vaz = 1
                                  rotace RR
                            else
                                  rotace RL
                            end if
                            p1↑.vaz = 0
                            h = false
                      end case
                end if
          else
                zpracování dat p↑.data, h = false
          end if
    end if
end if
end hledej1
```

4.2.2.2 Práce s AVL – stromy

Vyhledávání prvků, stejně jako ***průchod vyváženým stromem*** je zcela shodné se stromy nevyváženými. Obdobně jako u nich je pak operace ***vyhledávání*** a ***rušení*** prvků výrazně náročnější než ***vyhledávání*** a ***vkládání***. Kromě zrušení nalezeného prvku je třeba provést nové vyvážení stromu.

4.2.3 Tisk struktury stromu

Jako samostatnou kapitolu nyní vkládáme řešení speciálního problému, kterým je zobrazení struktury stromu. Přitom chceme, aby bylo zřejmé, na jaké ***úrovni*** prvek leží a nejlépe také ***vazby*** mezi prvky. Nejčastější výstupní zařízení bude zřejmě monitor počítače, tiskárna či textový soubor. Úroveň na jaké se prvek nachází, můžeme naznačit jeho odsazením o příslušný počet mezer. Pro jejich počítání zavedeme proměnnou ***u***. Jednotlivé prvky budeme zobrazovat pod sebou na řádcích. Proto bude vhodné začít nejpravějším prvkem stromu a postupovat směrem k nejlevějšímu

```
zobraz (w, u)
  if w <>nil
        zobraz (w↑.pravý, u+1)
        tiskni u mezer a klíč w↑.klíč
        zobraz (w↑.levý, u+1)
  end if
end zobraz
```

Složitější situace nastane, pokud chceme graficky znázornit také ***vazby mezi prvky*** pomocí spojovacích čar. Při tisku informace o aktuálním prvku potřebujeme znát také počet přechodů na levý či pravý podstrom, kterými jsme se k prvku dostali. Ty musíme jednak předávat při volání procedury, k tomu použijeme znakovou proměnnou ***s***. Potřebujeme však znát všechny předchozí hodnoty. Pro předem neznámou hloubku stromu by bylo vhodné je ukládat do

lineárního seznamu. Pokud známe hloubku stromu, můžeme použít pole ***uk*** se znakovými prvky, označenými 1, 2 až po maximální hloubku stromu.

Pro volbu vykreslovaných znaků znázorňujících vazby prvků platí následující pravidla:

Na poslední úrovni budeme volit pro

P - pravý podstrom - znak ´┌´

C - kořen stromu - znak ´—´

L - levý podstrom - znak ´└´

Na vyšších úrovních budeme volit mezi znakem mezera a ´|´, který prodlužuje spojení podstromu podle toho, jak jdou jednotlivá volání za sebou. Čáru kreslíme, pokud za Pravým podstromem následuje Levý a za Levým následuje Pravý, jinak kreslíme mezeru.

Proceduru pro zobrazení struktury celého stromu budeme volat ***zobraz(kořen, 1, ´C´)***

```
zobraz (w, u, s)
  if w <>nil
        uk[u] = s
        zobraz (w↑.pravý, u+1,´P´)
        s1 = ´ ´
        for i = 1 to u - 1 step 1
              if uk[i] = uk[i+1]
                    s1 = s1 + ´ ´
              else
                    s1 = s1 + ´|´
              end if
        end for
        case s = ´L´
              s1 = s1 + ´└´
        case s = ´P´
              s1 = s1 +´┌´
        case s = ´C´
              s1 = s1 + ´—´
        end case
        tiskni řetězec s1 a w↑.klíč
        zobraz (w↑.levý, u+1, ´L´)
  end if
end zobraz
```

Úpravu algoritmu pro ukládání informace o postupu stromem do ***lineárního seznamu*** ponecháváme na čtenáři. Je třeba si uvědomit, že při vstupu do procedury přidáváme prvek na konec seznamu a před jejím opuštěním ho opět vyjímáme. Protože víme, že zrušení posledního prvku vyžaduje přístup k předposlednímu prvku, bude vhodnější vkládat prvky na začátek lineárního seznamu, neboť rušení prvního prvku je velmi jednoduché. Jen nesmíme zapomenout, že informaci musíme zpracovávat od konce seznamu, což při použití rekurzívního algoritmu pro tvorbu řetězce ***s1*** nebude problém.

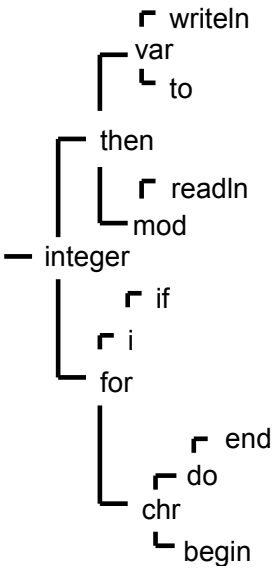
Výsledkem činnosti algoritmu bude zobrazení stromu, na které je třeba se dívat z pravé strany, aby pojmy levý a pravý podstrom odpovídaly zobrazení. Použitý přístup vychází ze zkušenosti, že obvykle mají stromy velké množství prvků při relativně malé ***hloubce*** (zvláště AVL – stromy):

Ukázka použití AVL-stromu pro rozbor slov v textu

rozebíraný text:

```
var i:integer;
begin
  for i:=1 to 255 do
    begin
      writeln(i:4,' ',chr(i));
      if i mod 10 = 0 then readln;
    end;
end.
```

struktura výsledného AVL-stromu:



V literatuře se můžeme setkat také se zobrazením stromu, při kterém jsou prvky na stejné úrovni vnoření umístěny na jedné vodorovné linii. K tomu je třeba zřetězit prvky na stejné úrovni a určit polohu na vodorovné linii. Výsledné zobrazení roste velmi rychle do šířky, což je nepraktické, proto tento algoritmus neuvádíme.

4.2.4 Optimální stromy

V úvodní kapitole jsme definovali ***délku cesty stromu*** (délku vnitřní cesty stromu) jako součet délek cesty ke všem prvkům stromu. Pro co nejrychlejší přístup k náhodně zvolenému prvku pak konstruujeme stromy s ***minimální délkou***. Přitom vycházíme z předpokladu, že pravděpodobnost výskytu je pro všechny prvky stejná.

V praxi se často setkáváme s aplikacemi, ve kterých se jednotlivé prvky vyskytují různě často, tedy ***pravděpodobnost výskytu*** prvků je různá. Jestliže pravděpodobnost přístupu k *i*-tému prvku stromu označíme p_i, pak platí

$$p_i \geq 0; \quad i = 1,2...n, \tag{4.1}$$

$$\sum_{i=1}^{n} p_i = 1, \tag{4.2}$$

kde je n - počet prvků ve stromu.

Pravděpodobnosti výskytů prvků můžeme chápat také jako ***váhy*** těchto prvků. S jejich znalostí je vhodné konstruovat stromy s ***minimální váženou délkou cesty***:

$$P_I = \sum_{i=1}^{n} p_i h_i \longrightarrow \min, \tag{4.3}$$

kde je h_i - úroveň i-tého prvku stromu,

p_i - pravděpodobnost přístupu k i-tému prvku,

P_I - vážená délka cesty stromu.

V řadě aplikací určujeme pravděpodobnost výskytu prvku jako poměr počtu výskytů tohoto prvku k počtu všech prvků v souboru prvků. Je zřejmě nutné, aby celkový počet prvků v souboru byl konečný.

$$p_i = \frac{m_i}{\sum_{i=1}^{n} m_i} \tag{4.4}$$

kde je m_i - počet výskytů i-tého prvku,

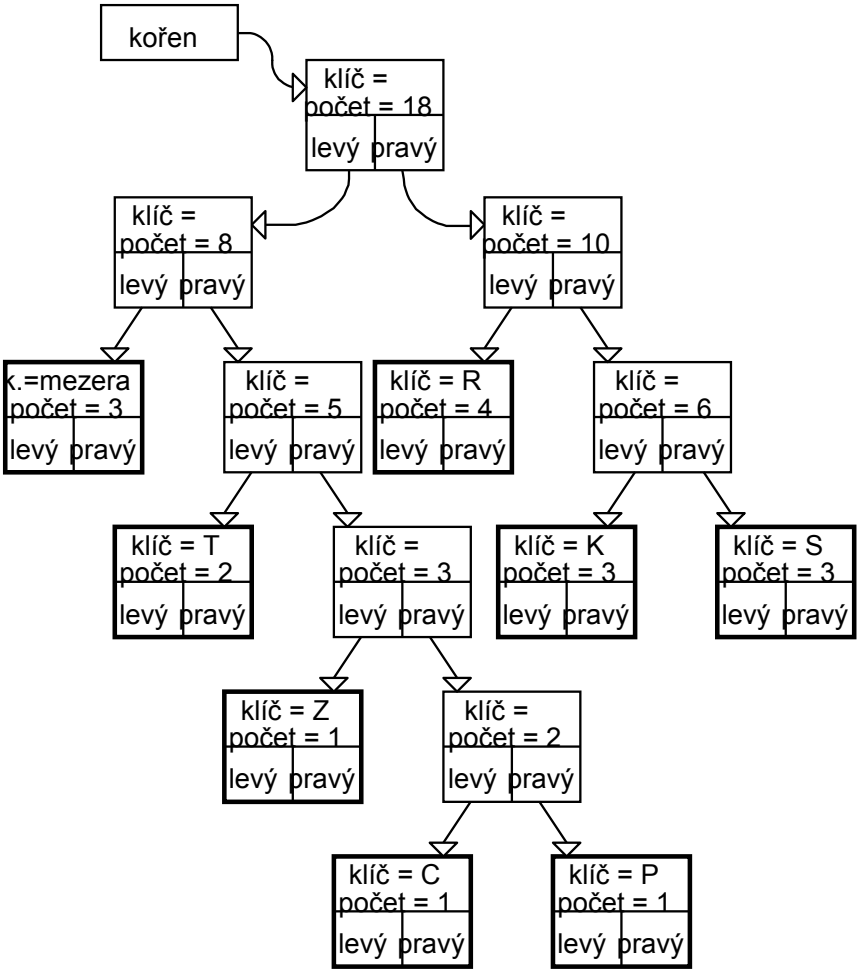
n - počet různých prvků v souboru.

Ze vztahu (4.4) je zřejmé, že beze ztráty obecnosti můžeme pracovat přímo s počtem výskytů prvku, pro který platí pouze podmínka (4.1), tedy $m_i \geq 0$ pro $i = 1, 2, \ldots n$, takže ***váženou délku cesty stromu*** můžeme v těchto aplikacích vyjadřovat upraveným vztahem (4.3) v podobě:

$$P_I = \sum_{i=1}^{n} m_i h_i \,. \tag{4.5}$$

Strom, u kterého dosahuje ***vážená délka cesty*** minimální hodnoty, nazýváme ***optimální strom***. Jeho konstrukce je dosti komplikovaná a časově náročná, složitost algoritmu je řádově $O(n^2)$ a velikost potřebné paměti (paměťová složitost) rovněž $O(n^2)$[8](tento algoritmus při konstrukci uvažuje také pravděpodobnosti výskytu prvků, které se ve stromu nevyskytují). Pokud se spokojíme s kvazioptimálním stromem, můžeme využít velmi efektivní algoritmus, který má složitost ***O(n.log n)*** a paměťové nároky úměrné ***O(n)***.

Jako ukázku aplikace ***optimálního stromu*** nyní uvedeme algoritmus tvorby stromu pro konstrukci ***Huffmanova kódu***. Jedná se o ***prefixový kód*** (žádné klíčové slovo není začátkem jiného kódového slova) minimální délky. Kódová slova mají tím kratší délku, čím mají větší pravděpodobnost výskytu. Huffmanův kód je vhodné definovat pomocí optimálního stromu (pro binární kód rovněž binárním). Informaci přitom nesou jen ***koncové prvky***, všechny vnitřní prvky jsou jen pomocné, jinak by kód nebyl prefixový. Kódové slovo pak tvoříme podle cesty ke koncovému prvku - postup k levému následníku zastupujeme znakem 0, k pravému 1. Následující obrázek znázorňuje optimální strom pro konstrukci Huffmanova kódu pro soubor znaků: "STRC PRST SKRZ KRK"



Obrázek 4.21 Strom Huffmanova kódu

Každý koncový prvek stromu (je nakreslen tučnou čarou) nese informaci o znaku a počtu jeho výskytů, vnitřní prvek pak součet výskytů všech jeho následníků. Tabulka uvádí počty výskytů jednotlivých prvků a přiřazená kódová slova.

Tabulka 4.1

prvek	poč. výskytů	kód. slovo	prvek	poč. výskytů	kód. slovo
C	1	01110	S	3	111
K	3	110	T	2	010
P	1	01111	Z	1	0110
R	4	10	mezera	3	00

Využití Huffmanova kódu je jednou z metod ***komprese dat***. Znakům souboru přidělíme kódová slova Huffmanova kódu a s jejich pomocí zakódujeme. Protože pro každý soubor konstruujeme speciální kód, nesmíme opomenout informaci o vytvořeném souboru uložit do ***komprimovaného souboru***. Postup tvorby stromu Huffmanova kódu můžeme rozdělit do tří částí. Pro uložení informace o znacích a počtu jejich výskytu využijeme dynamickou datovou strukturu. Každý prvek bude obsahovat informaci:

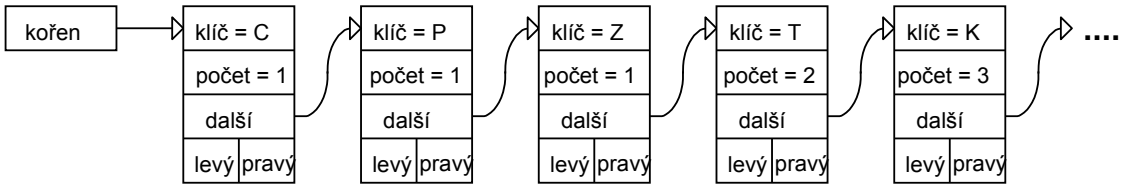
klíč = znak souboru (u koncových prvků),

počet = počet výskytů v souboru,

další = ukazatel na další prvek lineárního seznamu,

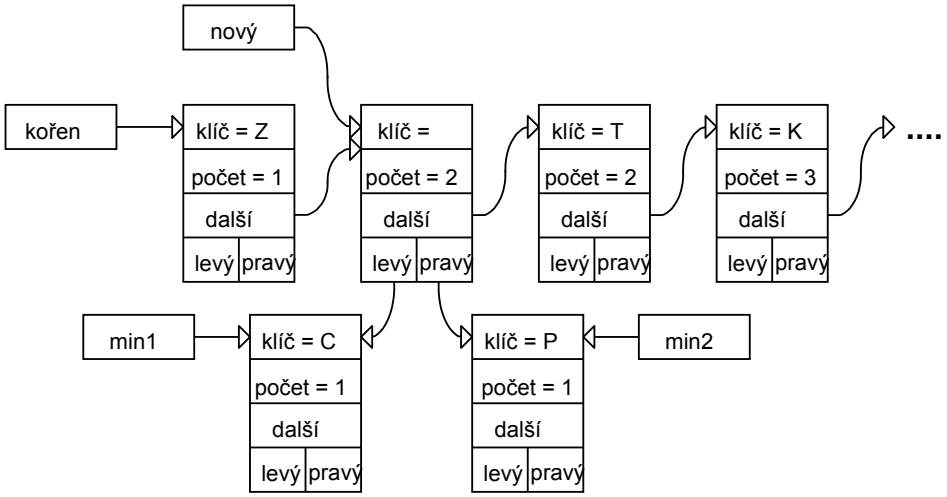
levý, pravý = ukazatele na následníky ve stromu.

4. Určíme, které znaky se v souboru vyskytují a kolikrát. Pokud je maximální počet různých znaků známý a nepříliš velký, můžeme využít statické ***pole*** (znaky obvykle kódujeme ASCII kódem, který obsahuje 256 kódových slov). Pro větší počet prvků (např. pro celá slova) bude vhodnější dynamická datová struktura. Můžeme využít ***lineární seznam***, nebo ***vyhledávací strom*** organizovaný podle klíče. (znaků).
5. Jednotlivé prvky seřadíme do ***lineárního seznamu*** podle počtu výskytů od nejmenšího k největšímu. Pokud jsme pro určení počtu výskytů prvků využili lineární seznam, provedeme jejich nové setřídění. Nejlépe pomocí stabilního algoritmu třídění. Tak zajistíme, že prvky se stejným počtem výskytů budou setříděny v definovaném pořadí. Nesmíme zapomenout položkám levý a pravý přiřadit hodnotu nil.



Obrázek 4.22 Lineární seznam seřazený podle počtu výskytů znaků

6. Pokud má lineární seznam více než jeden prvek (kořen↑.další <> nil), pak vyjmeme ze seznamu první dva prvky (označíme je ukazateli ***min1*** a ***min2***), vytvoříme nový prvek, prvky ***min1*** a ***min2*** uložíme jako jeho následníky. Počet výskytů nového prvku určíme jako součet výskytů jeho přímých následníků. Nakonec nový prvek zařadíme do lineárního seznamu a celou činnost zopakujeme.



Obrázek 4.23 Lineární seznam po vložení prvního vnitřního prvku budoucího stromu Huffmanova kódu

Algoritmus tvorby stromu Huffmanova kódu z lineárního seznamu:

```
if kořen <> nil
  while kořen↑.další <> nil
        min1 = kořen
        kořen = kořen↑.další
        min2 = kořen
        kořen = kořen↑.další
        new (nový)
        nový↑.levý = min1
```

```
            nový↑.pravý = min2
            nový↑.počet = min1↑.počet + min2↑.počet
            zařazení prvku nový do seznamu
      end while
   end if
```

4.2.5 B-stromy

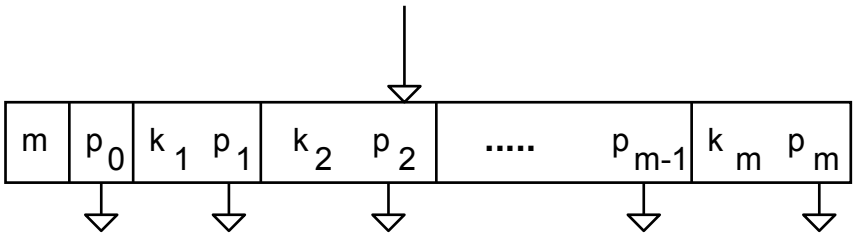
B-stromy jsou speciálním případem stromů ***vyššího stupně*** (každý prvek může mít více následníků). Uvádíme je pro jejich dobré vlastnosti, zejména jednoduché kritérium, usměrňující růst stromu. Navrhli je Rudolf Bayer & Ed McCreight v roce 1972, s názvem B jako „*ballanced*“, ale možná jako „*Bayer*“ nebo „*Boeing*“, neboť pracovali pro „*Boeing Scientific Research Labs*“.

Strom je složen ze ***stránek***. Každá stránka musí obsahovat ***n*** až ***2n*** prvků pro danou konstantu ***n***. (Kromě kořenové stránky, jak uvidíme dále.) Za těchto podmínek bude strom s ***N*** prvky a maximální velikostí stránky ***2n*** vyžadovat maximálně $\log_n N$ přístupů ke stránce pro vyhledání prvků. Přitom máme jistotu, že paměť bude využita nejméně z 50%.

I za uvedených podmínek budou algoritmy vyhledávání, vkládání a rušení prvků poměrně jednoduché. Datovou strukturu nazývanou B-strom definujeme následujícími pravidly. Parametr n přitom označujeme jako ***řád*** stromu.

7. Každá stránka obsahuje nejvýše ***2n*** prvků (klíčů).
8. Každá stránka kromě kořenové obsahuje alespoň ***n*** prvků.
9. Každá stránka je buď koncová (nemá následníky), nebo má ***m + 1*** následníků, kde ***m*** je počet prvků stránky.
10. Všechny koncové stránky jsou na stejné úrovni.

Pro uložení informace o jedné stránce zvolíme následující datovou strukturu. Informaci o jednotlivých prvcích zatím zúžíme na uložení hodnoty klíče, podle kterého je B-strom organizován.



Obrázek 4.24 Datová struktura stránky B-stromu

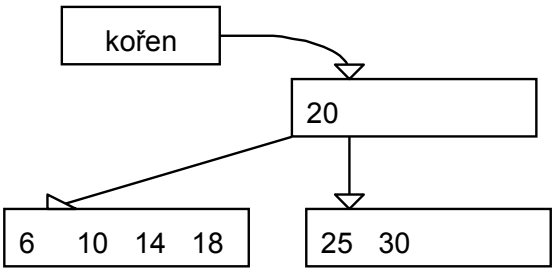
Vyhledávání prvku je rozšířením vyhledávání prvku v binárním stromu. Protože stránka obsahuje více než dva prvky, můžeme využít kromě ***sekvenčního prohledávání*** také rychlejší ***binární prohledávání***, zejména pro větší ***řády*** B-stromu. Pokud prvek najdeme v aktuální stránce, pak nastane jeden ze tří případů:

11. ***klíč*** < k_1 - vyhledávání pokračuje ve stránce $p_0\uparrow$,
12. k_i < ***klíč*** < k_{i+1} - vyhledávání pokračuje ve stránce $p_i\uparrow$,
13. k_m < ***klíč*** - vyhledávání pokračuje ve stránce $p_m\uparrow$.

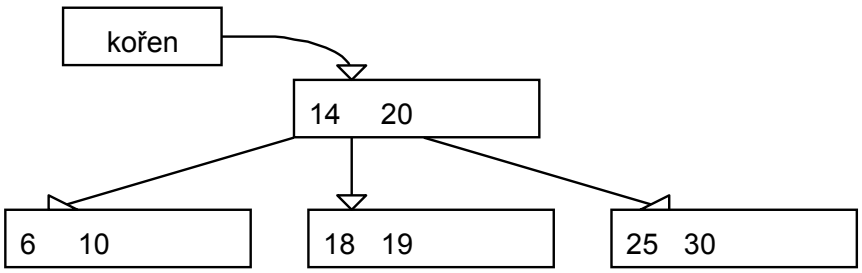
V případě, že ukazatel na další stránku má hodnotu nil, znamená to, že prvek ve stromu není a je nutno ho přidat do B-stromu. Přitom mohou nastat dva případy:

14. ***m*** < ***2n*** - datová struktura není plná, pak se prvek přidá do této stránky.

15. ***m*** = ***2n*** - stránka je plná, rozdělí se na dvě stránky po ***n*** prvcích (jedna obsahuje prvních ***n*** prvků, druhá posledních ***n*** prvků). Zbývající prvek (ze středu posloupnosti prvků) je přesunut do vyšší stránky. Pokud dojde k přeplnění vyšší stránky, opět dojde k jejímu rozdělení. Tento proces může pokračovat až ke ***kořenové stránce***. Jejím rozdělením vzroste hloubka stromu (výška) a vznikne nová ***kořenová stránka*** obsahující jediný prvek. Jak vidíme, B-strom roste zvláštním způsobem, od koncových prvků ke kořenům.



Obrázek 4.25 B-strom před uložením prvku s klíčem 19



Obrázek 4.26 B-strom po vložení prvku s klíčem 19 - došlo k rozdělení plné stránky

Algoritmus vyhledávání a přidávání prvků do B-stromu je realizován jako procedura hledej. Vstupují do ní parametry ***klíč*** a aktuální stránka (***a***). Odkazem je předávána logická proměnná ***h***, která informuje o tom, že došlo k rozdělení stránky a v proměnné ***u*** je pak předáván prvek, který je přesouván do vyšší stránky.

Pro uložení stránky B-stromu využijeme následující datové struktury:

stránka - structure

m - počet prvků ve stránce

p0 - ukazatel na prvního následníka

l [1 až 2n] -prvky stránky

end structure

prvek - structure

klíč - položka klíče

data - ostatní data spojená s prvkem

p - ukazatel na další stránku

end structure

```
hledej (klíč, a, REF h,REF u)
if a = nil
  klíč není v B-stromu, prvku u přiřadíme nový klíč
  u.klíč = klíč nového prvku
```

```
    u.data = data nového prvku
    h = true
    prvek u bude putovat nahoru v B-stromu
  else
    hledáme prvek s udaným klíčem ve stránce a pomocí
  binárního prohledávání
    if prvek nalezen
          manipulace s daty nalezeného prvku a↑.e[i]
    else
          hledej (klíč, následník, h, u)
          if h
                prvek postupuje nahoru
                if a↑.m < 2n
                      prvek u přidáme do stránky a↑
                      h = false
                else
                      stránku a↑ rozdělíme na dvě stránky
                      střední prvek dosadíme do u
                      h = true
                end if
          end if
    end if
  end if
  end hledej
```

Již bylo zmíněno, že kořenová stránka má výhradní postavení, její dělení je nutno naprogramovat samostatně. Nejjednodušší bude zařadit proceduru ***hledej*** do procedury ***vkládání***. V případě, že proměnná ***h*** bude mít po návratu z procedury ***hledej*** hodnotu true, znamená to, že je nutno rozdělit kořenovou stránku. K tomu použijeme ukazatel ***q***. Spolu s proměnnými ***h*** (logická) a ***u*** (prvek stránky) může být definována v proceduře ***vkládání*** jako lokální.

```
  vkládání (klíč)
    hledej (klíč, kořen, h, u)
    if h
          q = kořen
          new (kořen)
          kořen↑.m = 1
          kořen↑.p0 = q
          kořen↑.e[1] = u
    REM prvek u obsahuje informaci o klíči a ukazatel
    REM na stránku následníků
    end if
  end vkládání
```

Průchod B-stromem je opět rozšířením průchodu lineárním stromem. Následující procedura zpracuje prvky B-stromu v pořadí hodnot jejich klíčů.

```
  průchod (p)
    if p <> nil
          průchod (p↑.p0)
          for i = 1 to p↑.m step 1
                zpracování dat p↑.e[i].data
                průchod (p↑.e[i].p)
          end for
```

```
  end if
end průchod
```

Nejčastěji budeme rušit celý B-strom najednou. K tomuto účelu upravíme proceduru pro průchod stromem tím, že po zpracování celé stránky dojde k jejímu zrušení.

```
zrušení (p)
  if p <> nil
        zrušení (p↑.p0)
        for i = 1 to p↑.m step 1
              zrušení (p↑.e[i].p
              zrušení dat p↑.e[i].data)
        end for
        dispose (p)
end if
end zrušení
```

Na závěr kapitoly věnované B-stromům se ještě zmíníme o řešení jednoho problému. Tím je konstrukce vyhledávacích stromů, které svým rozsahem přesahují ***velikost dostupné operační paměti***. V tomto případě je nutno strom ukládat na jiné paměťové médium, které obvykle neumožňuje současný přístup ke všem částem informace (typickým příkladem je ***sekvenční soubor***). B-stromy nám nabízejí dobré řešení. Do souboru budeme ukládat jednotlivé stránky B-stromu. Přitom využijeme skutečnost, že stránka má známou velikost. Jako hodnoty ukazatelů na stránky pak budeme používat pozici stránky v souboru. Nové stránky budou zřejmě ukládány na konec souboru, jinak by bylo nutno upravovat hodnoty ukazatelů na odsouvané stránky při vložení nové stránky do souboru. Velikost stránky (***řád*** B-stromu) budeme volit tak, aby bylo možno celou stránku přesunout do operační paměti. V proceduře ***hledej*** je orámována část algoritmu, při níž je potřeba mít v paměti aktuální stránku. Při tomto řešení můžeme pro vyhledání prvku ve stránce využít rychlejší ***binární prohledávání***. Pokud bychom i s aktuální stránkou pracovali v souboru, museli bychom využívat ***sekvenční prohledávání***.

POUŽITÁ LITERATURA

[1] Arlow, J. & Neustadt, I. *UML a unifikovaný proces vývoje aplikací*. 1. Vyd. Brno, Computer Press, 2003, 388 s. ISBN 80-7226-947-X.

[2] Barton, D. P. & Pears, A. N. Application of Evolutionary Computation. In *Proceedings of First International Conference on Genetic Algorithms "Mendel '95"*. Red. Ošměra, P. Brno, VUT 1995, s. 15 - 21.

[3] Bayer, R. & McCreight, E. M. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1, 1972.

[4] Březina, T. *Informatika pro strojní inženýry I*. 1. vyd. Praha ČVUT 1991, 187 s.

[5] Brodský, J. & Skočovský, L. *Operační systém UNIX a jazyk C*. 1. vyd. Praha, SNTL 1989, 368 s.

[6] Cockburn, A. *Use Case – Jak efektivně modelovat aplikace*. 1. vyd. Brno, CP Books a.s., 2005, 262 s. ISBN 80-251-0721-3.

[7] Častová, N. & Šarmanová, J. *Počítače a algoritmizace*. 3. vyd. Ostrava, skriptum VŠB 1983, 190 s.

[8] Donghui Zhang. *B Trees*. Northeastern University, 22 pp. Dostupný z webu: http://zgking.com:8080/home/donghui/publications/books/dshandbook_BTree.pdf

[9] Drózd, J. & Kryl, R. *Začínáme s programováním*. Praha, Grada 1992, 312 s.

[10] Drozdová, V. & Záda, V. *Umělá inteligence a expertní systémy*. 1. vyd. Liberec, skriptum VŠST 1991, 212 s.

[11] Farana, R. *Zaokrouhlovací chyby a my*. Bajt 1994, č. 9, s 243 – 244.

[12] Flaming, B. *Practical data structures in C++*. New York, USA, Wiley, 1993.

[13] Hodinár, K. *Štandardné aplikačné programy osobných počítačov*. 1. vyd. Bratislava, Alfa 1989, 272 s.

[14] Holeček, J. & Kuba, M. *Počítače z hlediska uživatele*. Praha, SPN 1988, 240 s.

[15] Honzík, J. M., Hruška, T. & Máčel, M. *Vybrané kapitoly z programovacích technik*. 3. vyd. Brno, skriptum VUT 1991, 218 s.

[16] Hudec, B. *Programovací techniky*. Praha, ČVUT 1990.

[17] Jackson, M. A. *Principles of Program Design*. New York (USA), Academic Press 1975.

[18] Jandoš, J. *Programování v jazyku GW BASIC*. Praha, NOTO - Kancelářské stroje 1988, 164 s.

[19] Kačmář, D. *Programování v jazyce C++. Objektová a neobjektová rozšíření jazyka*. Ostrava, ES VŠB-TU 1995, 92 s.

[20] Kačmář, D. & Farana, R. *Vybrané algoritmy zpracování informací*. 1. vyd. Ostrava: VŠB-TU Ostrava, 1996. 136 s. ISBN 80-7078-398-2.

[21] Kaluža, J., Kalužová, L., Maňasová, Š. *Základy informatiky v ekonomice*. 1. vyd. Ostrava, skriptum VŠB 1992, 193 s.

[22] Kanisová, H. & Müller, M. *UML srozumitelně*. 1. vyd. Brno, Computer Press, 2004. 158 s. ISBN 80-251-0231-9.

[23] Kapoun, K. & Šmajstrla, V. *Základní fyzikální problémy - programy v jazyce BASIC a FORTRAN*. 1. vyd. Ostrava, skriptum VŠB 1987, 312 s.

[24] Kelemen, J. aj. *Základy umelej inteligencie*. 1. vyd. Bratislava, Alfa 1992, 400 s.

[25] Knuth, D. E. *The Art of Computer Programming*. Volumes 1-4A, 3rd ed. Reading, Massachusetts, Addison-Wesley, 2011, 3168 pp. ISBN 0-321-75104-3.

[26] Kopeček, I. & Kučera, J. *Programátorské poklesky*. Praha, Mladá fronta 1989, s. 150-155.

[27] Krček, B. & Kreml, P. *Praktická cvičení z programování. FORTRAN*. 1. vyd. Ostrava, skripta VŠB 1986, 199 s.

[28] Kučera, L. *Kombinatorické algoritmy*. 2. vyd. Praha, SNTL 1989, 288 s.

[29] Kukal, J. *Myšlením k algoritmům*. 1. vyd. Praha, Grada 1992, 136 s.

[30] Marko, Š. - Štěpánek, M. *Operačné systémy mikropočítačov SMEP*. 2. vyd. Bratislava/Praha, Alfa/SNTL 1988, 264 s.

[31] Medek, V. & Zámožík, J. *Osobný počítač a geometria*. 1. vyd. Bratislava, Alfa 1991, 256 s.

[32] Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*. 1. vyd. Heidelberg, Springer-Verlag Berlin 1992, 250 s.

[33] *Microsoft Developer Network*. Development Library January 1995. Microsoft Corporation 1995, CD - ROM.

[34] Molnár, Ľ. & Návrat, P. *Programovanie v jazyku LISP*. 1. vyd. Bratislava, Alfa 1988, 264 s.

[35] Molnár, Ľ. *Programovanie v jazyku Pascal*. Bratislava/Praha, Alfa/SNTL 1987, 160 s.

[36] Molnár, Z. *Moderní metody řízení informačních systémů*. Praha, Grada 1992, s. 211 - 221.

[37] Moos, P. *Informační technologie*, 1. vyd. Praha, ČVUT 1993, 200 s.

[38] Nešvera, Š., Richta, K. & Zemánek, P. *Úvod do operačního systému UNIX*. 1. vyd. Praha, ČVUT 1991, 185 s.

[39] Olehla, J. & Olehla, M. aj. *BASIC u mikropočítačů*. 1. vyd. Praha, NADAS 1988, 386 s.

[40] Ošmera, P. Použití genetických algoritmů v neuronových modelech. In *Sborník konference "EPVE 93"*. Brno VUT 1993, s. 88 - 95.

[41] Paleta, P. *Co programátory ve škole neučí aneb Softwarové inženýrství v reálné praxi*. 1. vyd. Brno, Computer Press, 2003, 337 s. ISBN 80-251-0073-1.

[42] Petroš, L. *Turbo Pascal 5.5* – Uživatelská příručka. 1. vydání. Zlín, MTZ 1990, s. 20 – 21.

[43] Plávka, J. *Algoritmy a zložitosť*. Košice, TU Košice, 1998. ISBN 80-7166-026-4.

[44] Podlubný, I. *Počítať na počítači nie je jednoduché*. PC World, 1994, č. 2, s. 112 – 115.

[45] Rawlins, G. J. E. *Compared to what – an introduction to the analysis of algorithms*. Computer Science Press, New York, 1992.

[46] Reverchon, A. & Ducamp, M. *Mathematical Software Tools in C++*. West Sussex (England) John Wiley & Sons Ltd. 1993, 507 s.

[47] Rychlík, J. *Programovací techniky*. České Budějovice, KOPP 1992, 188 s.

[48] Sedgewick, R. *Algorithms*. 1st ed. Addison-Wesley. ISBN 0-201-06672-6.

[49] Sirotová, V. *Programovacie jazyky*. 1. vyd. Bratislava, skriptum SVTŠ 1985, 138 s.

[50] Soukup, B. SGP verze 2.30. *Referenční a uživatelská příručka systému*. Uherské hradiště, SGP Systems 1991, s. 25-55.

[51] Synovcová, M. *Martina si hraje s počítačem*. 1. vyd. Praha, Albatros 1989, 144 s.

[52] Šarmanová, J. *Teorie zpracování dat*. Ostrava, FEI VŠB-TU Ostrava, 2003, 160 s.

[53] Šmiřák, R. *Unified Modeling Language*. Softwarové noviny, 2004, č. 12, s. 76 – 77.

[54] Tichý, V. *Algoritmy I.* Praha, FIS VŠE v Praze, 2006, 190 s. ISBN 80-245-1113-4.

[55] Vejmola, S. *Hry s počítačem*. 1. vyd. Praha, SPN 1988, 256 s.

[56] Virius, M. *Základy algoritmizace*. Praha, ČVUT 2008, 265 s. ISBN 978-80-01-04003-4.

[57] Víteček, A. aj. *Využití osobních počítačů ve výuce*. 1. vyd. Ostrava, ČSVTS FS VŠB Ostrava 1986, 202 s.

[58] Vítečková, M., Smutný, L., Farana, R. & Němec, M. *Příkazy jazyka BASIC*. Ostrava, katedra ASŘ VŠB Ostrava 1989, 44 s.

[59] Vlček, J. *Inženýrská informatika*. 1. vyd. Praha, ČVUT 1994, 281 s.

[60] Wirth, N. *Algoritmy a štruktúry udajov*. 2. vydání. Bratislava, Alfa 1989, s. 19 – 89.

[61] *Základy algoritmizace a programové vybavení*. 2. vyd. Praha, Tesla Eltos 1986, 168 s.

[62] Zelinka, I., Oplatková, Z., Šeda, M., Ošmera, P. & Včelař, F. *Evoluční výpočetní techniky. Principy a aplikace*. 1. vyd. Praha, BEN – Technická literatura, 2009. ISBN 978-80-7300-218-3.