## Sekvenční vyhledávání

* + Vyhledávání v nesetříděném poli nebo seznamu. Nejde jinak než prvky postupně procházet a každý srovnat.
  + Sekvenční vyhledávání se dá vylepšit použitím zarážky, ušetří se tak krok, kde se kontroluje, zda je konec seznamu.
  + Časová složitost: ~~O~~(n) – lineární

*Pseudokód:*

SequentialSearch(A, n, x) [Bez zarážky]

i <- 0

while i < n

if A[i] = x

return i

i <- i+1

return n

SequentialSearchSentinel(A, n, x) [Se zarážkou]

A[n] <- x

i <- 0

while true

if A[i]=x

return i

i <- i+1

## Binární vyhledávání v setříděném poli

* + V setříděném poli kde platí, že prvky jsou a0 ≤ a1 ≤ … ≤ an-1 se dá použít algoritmu binárního vyhledávání, taky zvaný vyhledávání půlením intervalu.
  + Funguje to tak že se vezme prvek, který je v poli uprostřed (je-li počet prvků sudý, vezme se jeden, zpravidla levý). Následně se tento prvek srovná s hledaným prvek a mohou nastat tři situace:

1. Vybraný prvek je hledaný prvek 🡪 vyhledávání skončí
2. Vyhledávaný prvek je menší než prvek vybraný 🡪 znovu se vybere prostřední prvek z levé části a opakuje se od kroku 1.
3. Vyhledávaný prvek je větší než prvek vybraný 🡪 znovu se vybere prostřední prvek z pravé části a opakuje se od kroku 1.
   * Časová složitost: ~~O~~(ln(n)) - logaritmická

*Pseudokód:*

BinarySearch(A, k, l, x) [rekurzivní]

if k>l

return -1

s <- (k+l)/2

if x<A[s]

return BinarySearch(A, k, s-1, x)

if x>A[s]

return BinarySearch(A, s+1, l, x)

return s

BinarySearch(A, n, x) [nerekurzivní, rychlejší]

k <- 0

l <- n-1

while k<l

s <- (k+l)/2

if x < A[s]

l <- s-1

else

if x > A[s]

k <- s+1

else

return s

return -1

## Binární vyhledávací stromy

* Jsou binární stromy v vlastnostmi:
  + V každém uzlu je uložen jeden datový prvek
  + Pro každý uzel *u* platí, že prvky uložené v levém podstromu jsou menší než prvek v *u* a v pravém podstromu jsou větší prvky než v uzlu *u*.
* **Vyhledávání prvku**
  + Aktuální uzel = u
    - Na začátku u je kořen stromu
    - Hledaná hodnota = x
  + Průběžný krok, prvek v uzlu u = c
    - Jestli x < c, tak se musí hledat v levém podstromu. Jako nový aktuální uzel u nastavíme levý podstrom aktuálního uzlu a znovu provedeme krok 2.
    - Pokud není x < c tak srovnáme, zda-li x > c. Pokud ano: tak se musí hledat v pravém podstromu. Jako nový aktuální uzel u nastavíme pravý podstrom aktuálního uzlu a znovu provedeme krok 2.
    - Pokud není x < c a x > c tak zbývá pouze x=c. Což znamená že prvek c je v aktuálním uzlu hledaným prvkem.
  + Časová složitost: ~~O~~(h) - kde h je výška stromu.

*Pseudokód:*

Search(T, x)

u <- T.root

while u != NIL

if x < u.item

u <- u.left

else

if x > u.item

u <- u.right

else

return u

return NIL

* **Přidání prvku**
  + Nejprve proběhne vyhledávání přidávaného prvku x, algoritmus může skončit třemi způsoby:
    - Prvek X byl ve stromu nalezen, přidávání tedy končí, protože ve standardních stromech je prvek pouze jednou.
    - Vyhledávání skončilo v uzlu u s prvkem c kde x < c a uzel u nemá levého následníka, v tomto případě vytvoříme nový uzel jako levého následníka u a do něho vložíme prvek c.
    - Vyhledávání skončilo v uzlu u s prvkem c kde x > c a uzel u nemá pravého následníka, v tomto případě vytvoříme nový uzel jako levého následníka u a do něho vložíme prvek x.

*Pseudokód:*

NewNode(x)

u <- new Node

u.item <- x

u.left <- u.right <- NIL

return u

Insert(T, x)

if T.root = NIL

T.root <- NewNode(x)

return true

u <- T.root

while true

if x < u.item

if u.left = NIL

u.left <- NewNode(x)

return true

u <- u.left

else

if x > u.item

if u.right = NIL

u.right <- NewNode(x)

return true

u <- u.right

else

return false

* **Odebrání prvku**
  + Označme odebíraný prvek *x*.
  + Vyhledáme prvek *x* ve stromu. Vyhledání může skončit třemi způsoby
    - Prvek *x* nebyl ve stromě nalezen – není co odebrat.
    - Prvek byl nalezen v uzlu V, který má nejvýše jednoho následníka. Tento uzel zrušíme. Pokud rušený uzel V jednoho následníka, jeho následník nyní bude následníkem předchůdce zrušeného uzlu V. Pokud rušený uzel V neměl předchůdce, jeho následník bude novým kořenem.
    - Prvek byl nalezen v uzlu V, který má dva následníky. V tomto případě do uzlu V přesuneme buďto nejpravější (největší) prvek z jeho levého podstromu anebo nejlevější (nejmenší) prvek z jeho pravého podstromu a uzel, z kterého byl prvek přesunut, zrušíme.

*Pseudokód:*

Delete(T, x)

u <- T.root

if u = NIL

return false

if u.item = x

T.root <- DeleteNode(u)

Return true

while true

if x < u.item

if u.left = NIL

return false

if u.left.item = x

u.left <- DeleteNode(u.left)

return true

u <- u.left

else

if u.right = NIL

return false

if u.right.item = x

u.right <- DeleteNode(u.right)

return true

u <- u.right

DeleteNode(u)

if u.left = NIL

return u.right

if u.right = NIL

return u.left

v <- u.right

if v.left = NIL

u.item <- v.item

u.right <- v.right

return u

w <- v.left

while w.left != NIL

v <- w

w <- w.left

u.item <- w.item

v.left <- w.right

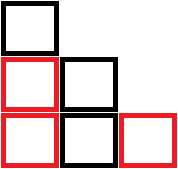
## AVL stromy

* Jsou určitým způsobem vyvážené binární vyhledávací stromy s příznivou (logaritmickou) časovou složitostí všech operací – vyhledání, přidání, odebrání.
* Aby byl strom vyvážený – rozdíl mezi výškami podstromů musí být maximálně o 1
* V každém uzlu je proměnná, která hlídá vyváženost stromu
* Pokud je strom nevyvážený, k jeho vyvážení slouží rotace
  + Jednoduché – RR, LL
  + Dvojité – RL, LR
* Přidání a odebírání probíhá stejně jako u binární vyhledávacího stromu, ale po každém přidání se kontroluje vyváženost stromu.

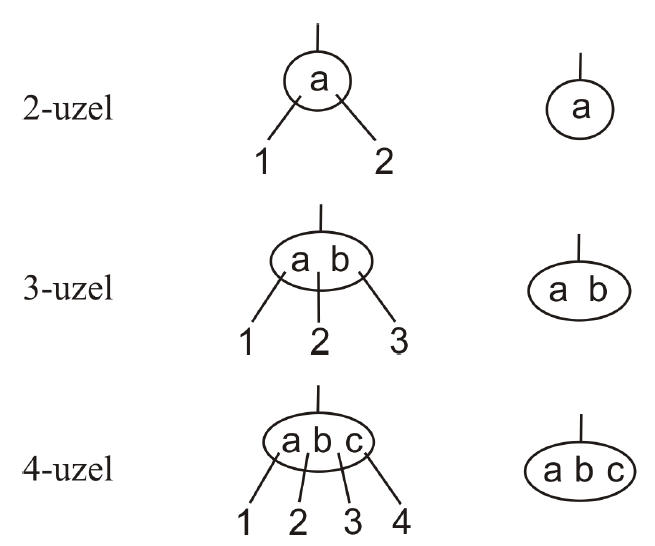
## B stromy

* Mají v uzlech uloženo více prvků
* Vlastnosti B stromů:
  + Kapacita uzlu (počet prvků, které lze do uzlu uložit) je u všech uzlů stromu stejná a volíme ji před začátkem vytváření stromu.
  + Všechny uzly (kromě kořene) musí být aspoň z poloviny zaplněny. Kořen musí obsahovat aspoň jeden prvek
  + Prvky uložené v uzlu jsou v něm seřazeny vzestupně dle velikosti.
  + Uzel je buď list anebo má o jednoho následníka více, než je počet prvků v něm.
  + Listy jsou v B-stromu jen v jeho poslední (nejspodnější) úrovni.
  + Časová složitost: Logaritmická
* **Vyhledávání**
  + Počáteční krok
    - Uzel, který je v daném okamžiku vyhledávání aktuální, budeme označovat u. Na začátku jím bude kořen stromu. Hledaná hodnota nechť je x.
  + Průběžný krok
    - Vyhledáme hodnotu x mezi prvky uloženými v aktuálním uzlu u. Protože prvky jsou v uzlu setříděné, lze k tomu použít binární vyhledávání. Vyhledání může skončit třemi způsoby:
      * Prvek byl v aktuálním uzlu u nalezen, čímž vyhledávání úspěšně končí.
      * Prvek nebyl v aktuálním uzlu u nalezen a tento uzel je list. Tím vyhledávání končí – hledaný prvek není ve stromu obsažen.
      * Prvek nebyl v aktuálním uzlu u nalezen a tento uzel je nelistový. V tom případě vyhledávání skončilo v místě, kde je odkaz na následníka, ve kterém vyhledávání má pokračovat. Tohoto následníka učiníme novým aktuální uzlem a opět se udělá krok 2.
* **Přidávání prvku**
  + Chceme-li do B-stromu přidat prvek, znamená to najít příslušný uzel, do kterého nový prvek patří, a následně ověřit, zda přitom nedošlo k přeplnění, a pokud ano, provést rozdělení uzlu.
* **Odebírání prvku**
  + Chceme-li z B-stromu odebrat prvek, znamená to vyhledat uzel, ve kterém se prvek nachází, prvek z něho odstranit a následně ověřit, zda odebráním prvku nepoklesl počet prvků v daném uzlu pod přípustnou mez, a pokud ano, je nutné to vyřešit.

## Červeno-černé stromy

* Jsou vyvážené vyhledávací stromy.
* Dají se přepsat jako B-stromy 4. řádu:
* Vlastnosti červeno-černých stromů
  + Kořenový uzel je vždy černý
  + Mezi kořenem a libovolným listovým uzlem je stejný počet černých hran (a tím i černých uzlů)
  + Ve stromu nikdy nenásledují dvě červené hrany po sobě
  + Nechť mezi kořenem a listem je m černých hran. Pak mezi kořenem a libovolným listem je nejvýše m+1 červených hran.
* **Přidávání**
  + Přidání probíhá standardním způsobem jako v běžném binárním vyhledávacím stromu. Nově přidaný prvek má červenou barvu. Po každém přidání prvku se musí zkontrolovat, jestli je strom vyvážený nebo jestli nejsou dva červené uzly (hrany) po sobě. Pokud není, musí se vyvážit. K tomu slouží přebarvování anebo rotace jako u AVL stromů.
* **Odebírání**
  + Probíhá stejným způsobem jako odebírání u binárního vyhledávacího stromu s tím rozdílem, že když odebíráme červený prvek, prostě jej odebereme, ale když odebíráme černý prvek, ve stromu musí zůstat stejný počet černých hran viz. Dvojitý černý uzel. Pokud je uzel obarven dvojitě černou barvou – odstraní se přebarvováním nebo rotacemi.

## 2-3-4 stromy

* 2-3-4 stromy jsou B stromy řádu 4, proto se také dají transformovat na červeno-černé stromy
* Časová složitost: logaritmická
* Jsou tři druhy uzlů:
* **Vyhledávání** – probíhá stejně jako u B stromu
* **Přidávání**
  + V průběhu hledání uzlu, do kterého má být prvek vložen, jsou plně obsazené uzly (4-uzly) na cestě od kořene k příslušnému listu preventivně štěpeny tak, aby při dosažení cílového listu tento nebyl plně obsazen (byl 2-uzel nebo 3-uzel). Přidání prvku do 2-3-4 stromu reprezentuje jen průchod dolů, neboť rozdělováním plně obsazených uzlů při tomto průchodu se zajistí, že po přidání prvku do listu nedojde k překročení jeho kapacity.
* **Odebírání**
  + Probíhá stejně jako u B stromu.

## Číslicové vyhledávací stromy

* lze použít pro prvky, které jsou reprezentovány v binárním tvaru (celá čísla, řetězce)
* Nejjednodušší způsob číslicového vyhledávání vychází z obecných binárních vyhledávacích stromů
* Větvení je založeno na hodnotě bitu:
  + 0 🡪 prvek patří do levého podstromu
  + 1 🡪 prvek patří do pravého podstromu
* **Vyhledávání:**
  + Uzel, který je v daném okamžiku vyhledávání aktuální, budeme označovat u. Na začátku jím bude kořen stromu. Hledaný prvek nechť je x. Aktuální index (pořadí) bitu označíme d, první bit má index 0.
  + Vezmeme prvek obsažený v aktuálním uzlu u, označme ho c.
    - Nejprve srovnáme, zda je x = c. Pokud ano, hledaný prvek je nalezen a vyhledávání tím úspěšné končí.
    - Jestliže x ≠ c, zjistíme hodnotu bitu d prvku x. Pokud je hodnota 0, je nutné v hledání pokračovat v levém podstromu. Jako nový aktuální uzel u položíme levého následníka současného aktuálního uzlu, zvýšíme hodnotu d o 1 a opět provedeme krok 2. Pokud uzel nemá levého následníka, vyhledávání končí – hledaný prvek není ve stromu obsažen.
    - Je-li hodnota bitu d rovna 1, pokračujeme obdobně ve vyhledávání v pravém podstromu, pokud aktuální uzel u má pravého následovníka.
* **Přidávání:**
  + Přidání prvku probíhá obdobně jako v binárním vyhledávacím stromě. Přidávaný prvek x nejprve vyhledáme. Pokud nebyl nalezen, vytvoříme příslušného následovníka v uzlu, ve kterém vyhledávání skončilo, a přidávaný prvek x do něho vložíme.
* **Odstranění:**
  + Prvek nejprve vyhledáme. Pokud byl nalezen, pak se rozlišují dva případy:
    - Prvek je v uzlu, který nemá více než jednoho následníka. Tento uzel odstraníme.
    - Prvek je v uzlu, který má dva následníky – je nahrazen kterýmkoliv prvkem z levého nebo pravého podstromu, jehož uzel lze odstranit (má nejvýše jednoho následníka).

## Hašování

Datová struktura je tabulka složená z řádků. Počet přihrádek (řádků) označíme m. Kapacita je tedy 0 až m-1. Jako velikost tabulky je dobré míti prvočíslo.

Základem je hašovací funkce, zobrazení, které hodnotě prvku přiřadí číslo v tabulce. Hašovací funkce se typicky skládá ze dvou funkcí. Ta první hodnotu prvku zobrazí na celé nezáporné číslo a druhá mu přiřadí číslo přihrádky z intervalu.

Cílem hašovací funkce je rovnoměrné rozmístění prvků v tabulce. Tzn. že první funkce, by měla mít vlastnosti:

* Zobrazit hodnoty prvku na co největší počet různých čísel
* Zobrazení na celá čísla by měla být rovnoměrná
* Aby nebyl výpočet příliš náročný

### Otevřené adresování

Když je pozice v tabulce vypočítaná hašovací funkci obsazena, počítá se další pozice tak dlouho, dokud se nenajde volná pozice, anebo se nezjistí, že tabulka je zaplněna.

### Lineární hledání

Nejjednodušší, nové pozice počítáme funkcí:

H(x, i) = (h(x) + i )mod m

h(x) … hašovací funkce

i … celočíselný parametr

m … rozsah tabulky

Je-li tedy pozice H(x, 0) obsazena, prohledávají se další pozice, H(x, 1) atp.

Lineární umisťování za sebou vede k vytváření nežádoucích shluků. (Pozn. že je tabulka plná se dá zjisti až když vyzkoušíme pozici H(x, m))

*Pseudokód:*

SearchLin(T,m ,x)

h <- h0 <- hash(x)

do

if T[h] = NIL

return -1

if T[h] = x

return h

h <- (h+1) mod m

while h != h0

return -1

### Kvadratické hledání

Používá se jako vylepšená verze lineárního hledání. U kvadratického hledání stále vznikají shluky, ale v menší míře. Jednoduchá hashovací funkce:

H(x, i) = (h(x) + i2 )mod m

h(x) … hašovací funkce

i … celočíselný parametr

m … rozsah tabulky

Problémem této funkce je, že se může dostat znovu na stejnou pozici, která už byla procházena, aniž by byla obsazena celá tabulka.

*Pseudokódy:*

SearchQuad(T, m ,x)

h0 <- hash(x)

for i <- 0 to m/2

h <- (h0 + i\*i)mod m

if T[h] = NIL

return -1

if T[h] ≠ FREE and T[h] = x

return h

return -1

InsertQuadr(T, m, x)

h0 <- hash(x)

for i <- 0 to m/2

h <- (h0 + i\*i) mod m

if T[h] = NIL or T[h] = FREE

T[h] <- x

return

error

DeleteQuadr(T, m, x)

h0 <- hash(x)

for i <- 0 to m/2

h <- (h0 + i\*i) mod m

if T[h] = NIL

return false

if T[h] ≠ FREE and T[h] = x

T[h] <- FREE

return true

return false

### Dvojí hašování

Ještě propracovanější řešení. Obecný tvar funkce:

H(x, i) = (h(x) + i \* h2(x))mod m

h(x) … primární hašovací funkce

h2(x) … sekundární hašovací funkce (Pozn. Nenabývá hodnotu 0, jenom 1 až m-1)

i … celočíselný parametr

m … rozsah tabulky

Nejčastěji je sekundární hašovací funkce vytváří využitím primární hašovací funkce, která je tvořena z nějaké výchozí funkce c(x) a má tvar:

h(x) = c(x) mod m

Použije se základní část (funkce c(x)) a vytvoří se h2(x) ve tvaru:

h2(x) = 1 + (c(x) mod (m-1))

*Pseudokód:*

SearchDHash(T, m ,x)

h <- hash(x)

h2 <- hash2(x)

for i <- 0 to m-1

if T[h] = NIL

return -1

if T[h] = x

return h

h <- (h + h2) mod m

return -1

### Vyhledávání v tabulce

Nejdřív se vypočítá hodnota hašovací funkce pro hledaný prvek. Pak se projde přihrádka, kterou určila hodnota hašovací funkce. Přihrádka může být buď prázdná (hledaný prvek není v tabulce) nebo obsazená. Pokud je v přihrádce jiný prvek než hledaný, začnou se postupně počítat další možné pozice a porovnávat prvky s hledaným prvkem, dokud se nenajde nebo se nenarazí na prázdnou přihrádku.

### Zřetězení

Metoda odstraňuje nevýhody otevřeného adresování (Omezená velikost hašovací tabulky a procházení prvků, které mají jinou hodnotu hašovací funkce než hledaný prvek)

K ukládání dalších prvků se stejnou hodnotou využívá seznamy. Hašovací tabulka obsahuje ukazatele na začátky jednotlivých seznamů.

*Pseudokódy:*

Search(T, x)

u <- T[hash(x)]

while u ≠ NIL

if u.item = x

return u

u <- u.link

return NIL

Insert(T, x)

u <- new Node

u.item <- x

h <- hash(x)

u.link <- T[h]

T[h] <- u

### Odebírání prvku z hašovací tabulky

Kvůli řešení kolizí je při přidávání a odebírání dobré používat metoda zřetězení. Při odebrání se prvek vyhledá a zruší se uzel v seznamu, ve kterém je prvek obsažen.

*Pseudokódy:*

Delete(T, x)

h <- hash(x)

u <- T[h]

if u = NIL

return false

if u.item = x

T[h] <- u.link

delete u

return true

while u.link ≠ NIL

v <- u.link

if v.item = x

u.link <- v.link

delete v

return true

u <- u.link

return false

## Rozšiřitelné hašování

* Před vytvořením hašovací tabulky si nemusíme vytvářet tabulku pevné velikosti
* Metoda hašování nazývaná extendible hashing nevyžaduje počáteční stanovení velikosti hašovací tabulky a navíc poskytuje konstantní časovou složitost vyhledání prvku bez ohledu na to, kolik prvků je v hašovací struktuře uloženo.
* Hašovací datová struktura se skládá ze dvou částí:
  + Adresáře
  + Přihrádek
* Adresář je pole odkazů na přihrádky. Jeho velikost je 2d, kde d se dynamicky zvětšuje podle počtu ukládaných prvků.
* **Vyhledávání:**
  + Vypočítáme hodnotu hašovací funkce hledaného prvku.
  + Vezmeme posledních d bitů bd…b1 vypočtené hodnoty hašovací funkce. Ty jsou indexem v poli adresáře. V prvku pole adresáře, který odpovídá tomuto indexu, zjistíme odkaz na příslušnou přihrádku.
  + V přihrádce procházíme v ní uložené prvky a srovnáváme je s hledaným prvkem, dokud nenalezneme prvek shodný s hledaným prvkem nebo je všechny neprojdeme (pak hledaný prvek není nalezen)
* **Přidávání:**
  + Vypočítáme hodnotu hašovací funkce přidávaného prvku.
  + Vezmeme posledních d bitů bd…b1 vypočtené hodnoty hašovací funkce. Ty jsou indexem v poli adresáře. V prvku pole adresáře, který odpovídá tomuto indexu, zjistíme odkaz na příslušnou přihrádku.
  + Je-li v přihrádce místo pro uložení přidávaného prvku, prvek do přihrádky vložíme. Jinak je nutné přihrádku rozdělit.
  + **Rozdělení:**
    - Pokud mají všechny prvky uložené v přihrádce stejnou hodnotu posledních d bitů bd…b1 hodnot svých hašovacích funkcí, přihrádku rozdělíme na dvě a do první dáme prvky, jejichž hašovací funkce má hodnotu dalšího bitu bd+1 rovnu 0. Do druhé dáme prvky, jejichž hodnota bitu bd+1 je 1. Po rozdělení přihrádek následně i zdvojnásobíme velikost adresář (hodnotu d zvýšíme o 1). Při zdvojnásobení velikosti adresáře zachováme všechny odkazy na přihrádky, které se nerozdělily
* **Odebírání:**
  + Vyhledáme odebíraný prvek.
  + Pokud byl nalezen, odstraníme ho z dané přihrádky.
  + Při větším odebírání, kdy vznikne více prázdných přihrádek, lze některé přihrádky sloučit. Případně po větším sloučení je někdy možné zmenšit velikost adresáře na polovinu. Jednoduchým způsobem lze sloučit prázdnou přihrádku, na kterou je jen jeden odkaz z adresáře, s přihrádkou, na kterou je opět jen jeden odkaz z adresáře a zároveň hodnoty indexů těchto odkazů v adresáři se liší jen v jednom bitu.

## Trie

* Mají prvky uložené jen v listový uzlech
* Ve stromu dále platí, že v každém listovém uzlu je právě jeden prvek
* **Vyhledávání:**
  + Uzel, který je v daném okamžiku vyhledávání aktuální, budeme označovat u. Na začátku jím bude kořen stromu. Hledaný prvek nechť je x. Aktuální index (pořadí) bitu označíme d, první bit má index 0.
  + Je-li aktuální uzel nelistový, zjistíme hodnotu bitu d prvku x.
    - Je-li tato hodnota 0, ověříme, zda uzel u má levého následovníka. Pokud ano, učiníme ho novým aktuálním uzlem, zvýšíme hodnotu d o 1 a opět provedeme krok 2. Pokud uzel levého následníka nemá, vyhledávání končí, hledaný prvek není ve stromu obsažen.
    - Je-li hodnota bitu d rovna 1, pokračujeme obdobně ve vyhledávání v pravém podstromu, pokud aktuální uzel u má pravého následovníka.
  + Je-li aktuální uzel u list, srovnáme, zda je hledaný prvek x roven prvku, který je uložen v tomto uzlu. Pokud ano, hledaný prvek byl v tomto uzlu nalezen, jinak vyhledávání končí neúspěšně.
* **Přidávání:**
  + Prvek x nejprve vyhledáme. Pokud nebyl nalezen, mohou nastat dva případy:
    - Vyhledávání skončilo v nelistovém uzlu, protože v něm nebyl příslušný následník, aby vyhledávání mohlo pokračovat. Zde tohoto následníka vytvoříme a přidávaný prvek x do něho vložíme.
    - Vyhledávání skončilo v listovém uzlu, ve kterém je uložen prvek c. Před tento list přidáváme nelistové uzly pro všechny bity až po bit (včetně tohoto bitu), jehož hodnoty jsou v prvcích x a c různé. Následně vytvoříme nový uzel, do kterého dáme prvek x a tento uzel učiníme následníkem posledního přidaného nelistového uzlu. Druhým následníkem tohoto uzlu je uzel s prvkem c.
* **Odebírání** je stejné jako u číslicových vyhledávacích stromů

## Patricia trie

* Stromy trie mají nevýhody:
  + Vnitřní a listové uzly jsou rozdílné (ve vnitřních uzlech prvek uložen není, v listových ano), což komplikuje efektivní implementaci.
  + Ve stromu se vyskytují za sebou i sekvence vnitřních uzlů, ve kterých není žádné větvení, což citelně zvyšuje počet uzlů ve stromu.
* Patricia trie tyto nevýhody odstraňují
* Vlastnosti:
  + V každém uzlu je uvedeno, podle kterého bitu je v tomto uzlu větvení (index bitu větvení)
  + Každý uzel má dva odkazy na další uzly (odkaz může být i na sebe)
  + Všechny uzly (nelistové i listové) mají stejnou strukturu:
    - datový prvek + index bitu větvení + 2 odkazy na další uzly
  + Kořenem stromu je specifický uzel, ve kterém je uložen nulový prvek a je v něm odkaz na první z uzlů, ve kterém jsou uloženy prvky. Kořen budeme označovat jako hlavu stromu. Nulový prvek (ve všech bitech má hodnotu 0) má v patricia trie specifickou úlohu. Do stromu proto nelze vložit prvek, který má stejnou hodnotu jako nulový prvek.
* **Vyhledávání:**
  + Počáteční krok
    - Uzel, který je v daném okamžiku vyhledávání aktuální, budeme označovat u. Na začátku jím bude uzel, na který je levý odkaz v hlavě stromu. Hledaný prvek nechť je x. Aktuální index bitu větvení označíme d. Jeho hodnotu na začátku nastavíme na -1. Index bitu uložený v uzlu u udávající, podle kterého bitu je v uzlu u větvení, budeme označovat u.bit. Pro zjištění hodnoty bitu prvku x s indexem d budeme používat funkci Bit(x,d).
  + Průběžný krok
    - Je-li index bitu u.bit uložený v aktuálním uzlu u menší nebo roven indexu d, je to příznak, že jsme se dostali na konec vyhledávání. Srovnáme, zda prvek uložený v aktuálním uzlu u je roven hledanému prvku x. Pokud ano, prvek byl nalezen. Jinak hledání končí neúspěšně.
    - Jestliže nejsme na konci vyhledávání, do indexu d uložíme hodnotu bitu větvení aktuálního uzlu u.bit. Zjistíme hodnotu Bit(x,d). Je-li tato hodnota 0, učiníme aktuálním uzlem levého následníka současného uzlu u, jinak jím bude jeho pravý následník. A vyhledávání opět pokračuje krokem 2.
* **Přidávání:**
  + Přidávaný prvek x nejprve vyhledáme. Nechť vyhledání skončilo v uzlu, ve kterém je prvek y.
    - Je-li prvek y obsažený v uzlu, ve kterém vyhledávání skončilo, roven hledanému prvku x, přidávání tím končí.
    - Pokud nebyl prvek x ve stromu nalezen, začneme postupně srovnávat od začátku (zleva) jednotlivé bity prvku x s odpovídajícími bity prvku y, až najdeme první bit, ve kterém se tyto prvky liší. Označme index tohoto bitu d. Následně začneme opět procházet strom od začátku po stejné cestě, po které jsme vyhledávali přidávaný prvek x, tak dlouho, dokud
      * nenajdeme uzel u, jehož index bitu u.bit je větší než index d. Pak před tento uzel vložíme do stromu nový uzel v, do kterého dáme přidávaný prvek x a index bitu v.bit tohoto uzlu nastavíme na d.
      * anebo se dostaneme do uzlu u, ve kterém vyhledávání skončilo. Pak nový uzel v s prvkem x vložíme pod uzel w, z kterého jsme k uzlu u dostali (vložíme ho do hrany vedoucí z uzlu w nahoru do uzlu u) a údaj indexu bitu v.bit nového uzlu nastavíme na d.
    - Při vkládání nového uzlu v bude jeho levý odkaz na tento uzel (na sebe), jestliže Bit(x,d) má hodnotu 0. Je-li hodnota tohoto bitu rovna 1, je naopak jeho pravý odkaz na tento uzel (na sebe).