Jevčák Vít

AVL stromy
Zápočtový program

2023

Obsah

Úvod

Tématem mého zápočtového je implementace AVL stromů v Pythonu. AVL stromy jsou datová struktura pro uchovávání a vyhledávání hodnot. Patří do skupiny binárních vyhledávacích stromů (BVS), pro které platí, že všechny hodnoty v levém resp. pravém podstromu jsou menší resp. větší než hodnota vrcholu. AVL stromy jsou specifické tím, že si udržují rozumnou hloubku (logaritmickou) pomocí rotací a pro každý vrchol AVL stromu platí podmínka, že se výšky jejich podstromů liší maximálně o 1.

Uživatelská dokumentace

Ovládání

Programu se zadávají příkazy přes textový soubor AVL.txt, příkazy se zadávají každý na svůj samostatný řádek, přičemž příkazy, řádky a hodnoty oddělujeme mezerou.

Insert

- Pro přidání vrcholů do stromů použijeme příkaz „insert“ s názvem stromu a hodnotami, které chceme do stromu vložit. Počet hodnot vkládaných do stromu není omezen. Pokud neexistuje strom pod zadaným názvem, tak se při vložení vytvoří nový strom. Insert upozorní na to, pokud přidáme do stromu moc vrcholů a strom nebude poté vypisovatelný.

Find

- Pro zjištění, jestli strom obsahuje vrchol s hodnotou použijeme příkaz „find“ s názvem stromu a hodnotami, které chceme ve stromu najít. Počet hodnot není omezen. Strom ve kterém chceme hledat musí existovat.

Delete

- Pro odstranění hodnot ze stromu použijeme příkaz „delete“ s názvem stromu a hodnotami, které chceme ze stromu smazat. Počet hodnot není omezen. Hodnoty, které nejsou ve stromu, který zadáme, nebudou mít žádný vliv na chod programu. Strom, ve kterém chceme odstraňovat hodnoty musí existovat. Pokud

odstraníme ze stromu poslední hodnotu, strom automaticky odstraníme.

Merge

- Pro spojení dvou stromů použijeme příkaz „merge“ s názvy 2 stromů, které chceme spojit a s názvem nového stromu, který bude obsahovat hodnoty z obou stromů. Oba stromy musí existovat a název konečného není používán na jiný strom.

Print

- Pro vytisknutí hodnot stromu použijeme příkaz „print“ s názvy stromů, které chceme vytisknout. Hodnoty stromů se vytisknou v pořadí PreOrder. Při zadání neplatného názvu vytiskne chybovou zprávu. Print je omezen na vytisknutí stromu o velikosti 256 vrcholů.

Destroy

- Pro odstranění stromů použijeme příkaz „destroy“, s názvy stromů, které chceme odstranit. Počet stromů není omezen. Neexistence stromu nemá vliv na chod programu.

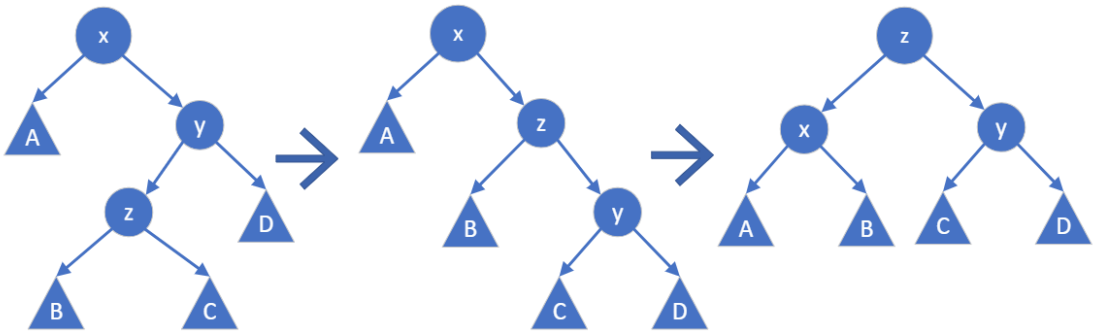
Programátorská dokumentace

Přehled tříd

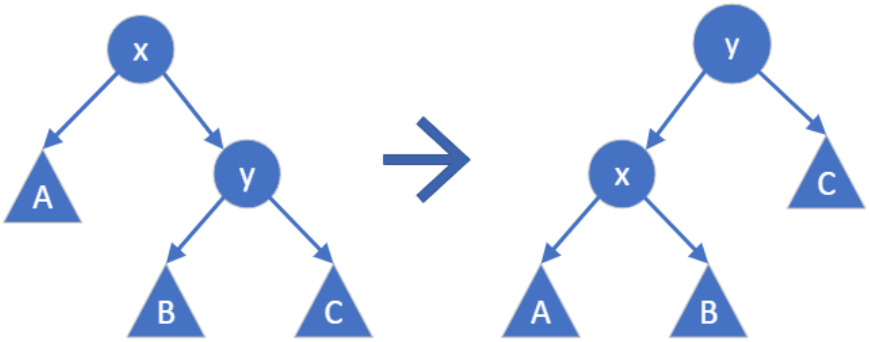
- V tomto programu mám 2 třídy *Node* a *Tree*, kdy *Node* reprezentuje vrchol stromu a má vlastnosti *key*, která uchovává hodnotu podle které hledáme, *left*, ve které si udržujeme odkaz na levého syna, *right*, kde si udržujeme odkaz na pravého, a *height*, kde si udržujeme výšku stromu s vrcholem stromu v našem *Node*. Třída *Tree* reprezentuje strom a obsahuje vlastnost *count*, která nám říká počet vrcholů ve stromě.
- Jako pomocné objekty využívám 2 slovníky *trees* a *roots*, ve kterých si udržuji odkazy na stromy a jejich kořeny dle klíče – stringu ze vstupu.

Přehled funkcí

- ***getHeight(root)***
  Funkce, která vrací výšku vrcholu, pokud vrchol neexistuje tak vrací 0.
- ***getBalance(root)***
  Funkce, která vrátí znaménko vrcholu (balance), kterou definujeme jako rozdíl výšek pravého a levého podstromu vrcholu.
- ***getMinNode(root)***
  Rekurzivní funkce, která vrátí nejmenší hodnotu z podstromu zadaného vrcholu. Rekurze se zastaví v momentě, kdy neexistuje levý syn vrcholu, což znamená, že ve stromu již není menší hodnota než nynější.
- ***insert(tree, root, key)***
  Rekurzivní funkce na přidávání hodnot do stromu, jako parametry bere strom, kořen strom a hodnotu, kterou chceme přidat. Rekurze končí tím, že buď vrchol s danou hodnotou neexistuje a my ho přidáme, či vrchol s danou hodnotou již existuje a my se pouze vrátíme. Pokud vrchol neexistuje, znamená to, že jsme se dostali na místo, kam hodnota patří. Pokud vrchol existuje, tak zkontrolujeme, jestli hodnota vrchol se nerovná přidávané hodnotě. Pokud se rovná, pak pouze vrátíme vrchol, ve kterém se nacházíme. Pokud ne, pak zkontrolujeme, jestli patří do levého resp. pravého podstromu a zavoláme insert na levého resp. pravého syna. Po vrácení se z rekurzivních volání aktualizujeme výšky vrcholů a kontrolujeme znaménko, pokud znaménko nesedí, pak rotujeme hrany viz obrázek. Na konci vracíme vrchol, ve kterém se právě nacházíme.



Případ, kdy je znaménko vrcholu x větší než 1 a přidávaná hodnota je menší než hodnota pravého syna x (obdobný případ pro znaménko menší než 1 a přidávaná hodnota větší než hodnota levého syna). Postupně provedeme pravou rotaci na pravého syna a levou rotaci na kořenu ve kterém se nacházíme. Výšky podstromů A, B, C a D jsou h a znaménko z je po rotaci 0.

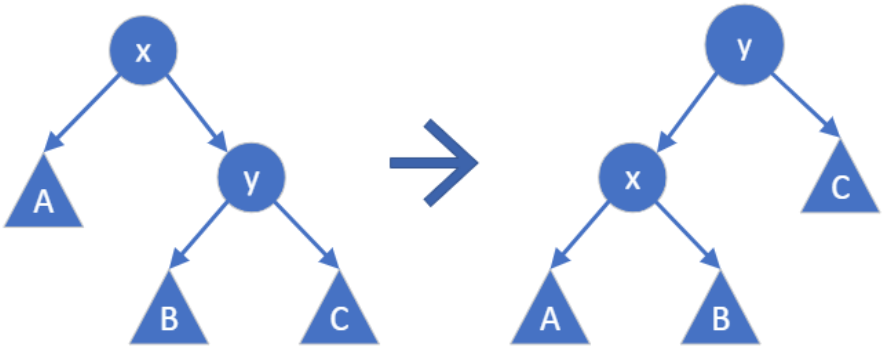


Případ, kdy je znaménko vrcholu x > 1 a přidávaná hodnota je větší než hodnota pravého syna x (obdobný případ pro x<1 a přidávaná hodnota větší než hodnota levého syna). Výška C = h+1, výšky A, B = h. Jednoduchá levá rotace. Znaménko y je na konci 0.

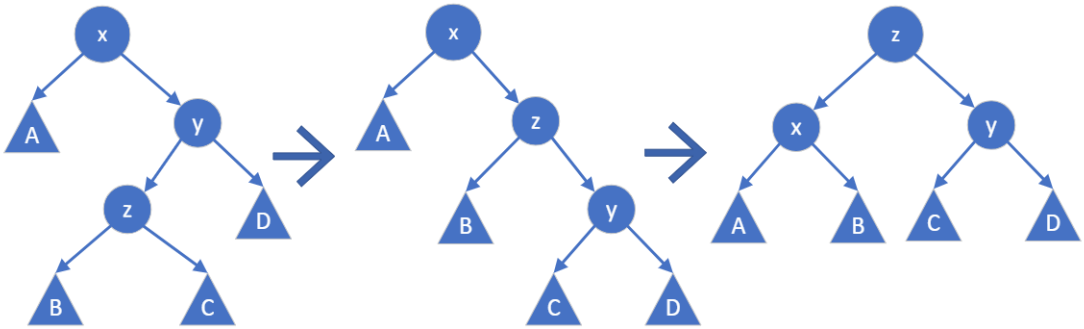
- ***delete(tree, root, key)***

  Rekurzivní funkce na mazání hodnot ze stromu. Rekurze končí tím, že buď najdeme vrchol a ten smažeme, nebo vrchol ve stromu nenajdeme a pouze se vrátíme. Pokud vrchol s hledanou hodnotou není ve stromu, pak pouze vracíme vrchol ve kterém se právě nacházíme. Pokud se hodnota hledaná nerovná hodnotě vrcholu, pak voláme *delete* na levého resp. pravého syna vrcholu. Pokud se hodnota vrcholu shoduje s hodnotou, kterou máme vymazat, pak se podíváme kolik má nynější vrchol synů. Pokud má pouze jednoho, pak napojíme místo našeho vrcholu jeho jediného syna a toho vracíme. Pokud má vrchol oba syny, pak vezmeme z jeho pravého podstromu nejmenší hodnotu a tu napojíme místo něj. Při vracení z rekurzivních funkcí aktualizujeme

výšky a znaménka, pokud znaménka nesedí pak rotujeme viz. obrázek. Na konci vracíme nynější vrchol.



Případ kdy znaménko je větší než 1 a zároveň znaménko pravého syna je 1 resp.0 . Výšky podstromů jsou: A = h, B = h resp. h+1, C = h+1. Provedeme levou rotaci. Znaménko vrcholu y po rotaci je 0. (Symetrický případ když znaménko je menší než -1 a znaménko levého syna je -1 resp.0)

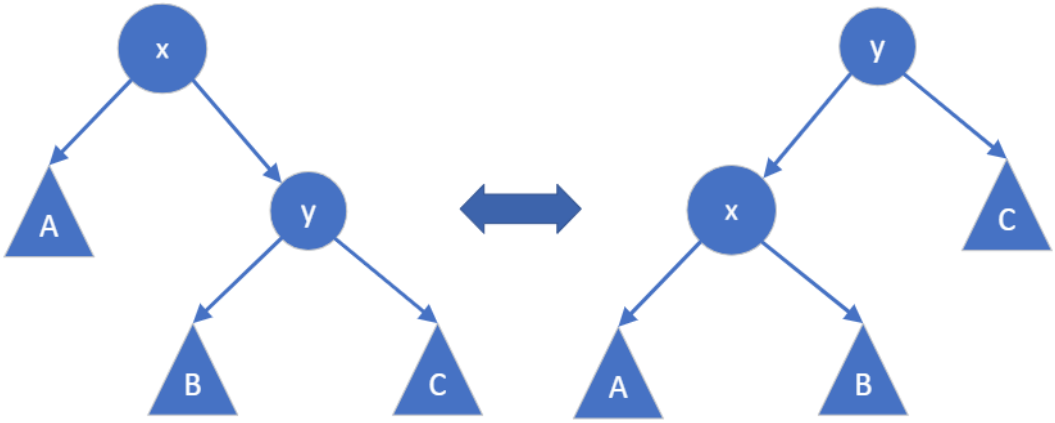


Případ kdy znaménko je větší než 1 a zároveň znaménko pravého syna je -1. Výšky podstromů jsou: A = h, B = h/h-1, C = h/h-1, D = h. Provedeme nejdříve pravou rotaci u pravého syna a poté levou rotaci u kořene. Znaménko vrcholu z po rotaci je 0.

- ***find(tree, root, key)***
  Rekurzivní funkce na zjištění zda se hodnota nachází ve stromu. Rekurze končí tím, že buď se dostaneme do vrcholu, který neexistuje (místo, kde by měla být hodnota), tudíž hodnota se ve stromu nenachází a vracíme False, nebo se hodnota shoduje s hodnotou vrcholu ve kterém se nacházíme a vracíme True. Jinak voláme funkci na levý resp. pravý podstrom vrcholu, podle toho jestli je hodnota vrcholu menší resp. větší než hledaná hodnota.
- ***preOrder(root)***
  Rekurzivní funkce která vytiskne strom v pořadí preorder. Pokud se dostaneme do vrcholu, který neexistuje tak se vracíme, jinak

vytiskneme hodnotu vrcholu a zavoláme *preOrder* na levý podstrom, poté na pravý.

- ***inOrder(root)***
  Rekurzivní funkce, která nám vrátí vrcholy seřazené v listu. Pokud se dostaneme, někam kde vrchol není, pak vrací pomocný list, jinak nejdříve zavoláme *inOrder* na levý podstrom, poté do listu přidáme hodnotu nynějšího vrcholu a pak zavoláme funkci na pravý podstrom.
- ***RotationL(root)***
  Funkce která otočí pravou hranu vrcholu doleva. Funkce funguje viz obrázek kdy původní stav je vlevo a výstupní stav napravo.



Levá resp. pravá rotace

- ***RotationR(root)***
  Funkce, která otočí levou hranu vrcholu doprava. Viz obrázek, kdy původní stav je na pravé straně a výstupní na levé straně.
- ***merge(array1, array2)***
  Funkce, která sleje dva seřazené listy do jednoho listu, který je také seřazený. Funkce má pro každý list svůj „pointer“ a vždy kontroluje, který list má na daném místě menší hodnotu, daná hodnota se pak přidá do slitého listu a pointer se zvýší. Pokud se v listech nachází stejné hodnoty pak se zkopíruje do slitého listu pouze jednou a oba pointery se zvýší (Funkce předpokládá, že dané listy obsahují v sobě vždy různé hodnoty, což je pro nás dostačující, jelikož do stromu nevložíme nikdy 2 stejné hodnoty).
- ***constructTree()***
  Rekurzivní funkce, která ze seřazeného listu postaví strom v čase O(m), kde m je počet prvků v listu. Funkce se zastaví, pokud nemá prvky které vložit do podstromů. Funkce vezme prostřední hodnotu listu a postaví jí jako kořen nového stromu, poté postaví levý podstrom zavoláním sebe sama na levého syna s polovičním listem a pak pravý podstrom obdobně.

- ***deleteNodes()***
  Rekurzivní funkce, která odstraní vrchol a všechny vrcholy, které jsou na něj napojeny. Rekurze končí, když narazí na vrchol který neexistuje. Jinak volá sebe sama nejdříve na levého syna a poté na pravého, nakonec vrchol ve kterém se nacházíme označíme za *None* a vrátíme.
- ***deleteTree()***
  Funkce, která zajistí odstranění stromu. Nejdříve odstraní strom pomocí funkce *deleteNodes* a poté ze slovníků *roots* a *trees* odstraní odkazy na tyto stromy, vrcholy, strom ve *trees* předtím ještě označí za *None*.

Složitosti operací

insert, delete, find
Složitost insertu do BVS je O(h), kde h je hloubka stromu. Díky vlastnostem AVL stromu je hloubka O(log n), kde n je počet vrcholů ve stromu. Operace rotace se provede maximálně h-krát (neplatí pro find), tudíž celková složitost insertu, deletu a findu je O(log n)
merge
Složitost operace závisí na jejích podoperacích. Nejdříve provedeme inOrder pro stromy o počtu vrcholů m resp. n. Složitost inOrderu je O(p), kde p je počet vrcholů ve stromu. Dále listy z inOrderu slijeme do jednoho funkcí merge, která má složitost O(m+n), kde m, n jsou počty hodnot v listech. Dále provedeme operaci constructTree, která má složitost O(k), kde k je počet hodnot v listu. Tudíž máme složitosti operací postupně O(m), O(n), O(m+n), a O(m+n) a z toho celková složitost je O(m+n).
print
Operace print má složitost O(n), kde n je počet vrcholů ve stromu, který tiskneme.