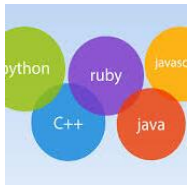
二叉樹、平衡樹、紅黑樹，放馬來吧

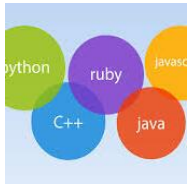
2019.11.05 程式語言

近期文章



Vue中容易被忽視的知識點

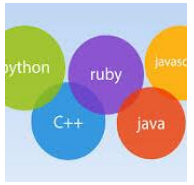
2019.12.09



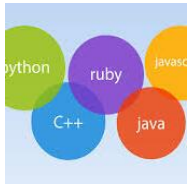
if我是前端Leader，談談前端框架體系建設

2019.12.09

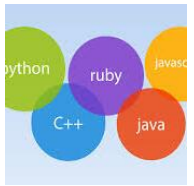
扁擔寬　板凳長
扁擔想綁在板凳上
板凳不讓扁擔綁在板凳上
扁擔偏要綁在板凳上
板凳偏偏不讓扁擔綁在那板凳上
到底扁擔寬　還是板凳長
……
BST矮 BST高
BST有AVL 也有RBTree
各種形狀的小樹，各種性格的大樹
到底小樹好 還是大樹好
……



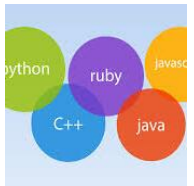
Spark入門（一）用SparkShell初嘗Spark滋味
2019.12.08



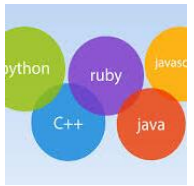
Spark入門（二）如何用Idea運行我們的Spark項目
2019.12.08



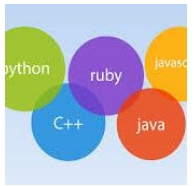
Spark入門（三）Spark經典的單詞統計
2019.12.08



Spark入門（四）Spark的map、flatMap、mapToPair
2019.12.08



Spark入門（五）Spark的reduce和reduceByKey
2019.12.08



Spark入門（六）Spark的combineByKey、sortBykey
2019.12.08

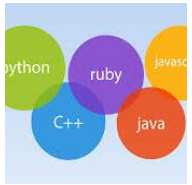
Spark入門（七）Spark的intersection、subtract、union和distinct

暫時編到這裡.....

不知道為什麼，各種樹在大腦裡各種浮現，想起了上面的歌詞。

世間物種億萬萬，人都個性習慣差異萬萬千，何況DataStructure世界裡的這些樹呢？

2019.12.08



Spark實戰尋找5億次訪問中，訪問次數最多的人

2019.12.08

目錄 關閉

寫在前面

> 道生一，一生二，二生三，三生萬物。

任何事物都不是憑空產生，也不會兀自消失。
它們的出現都有各自的道理，生存也遵循著一定的法則，消失也有各自的緣由。
天空之浩瀚，江海之遼闊，處於其中的世間萬物都在用自己的方式達到各自的平衡。

DataStructure裡的各種樹看得有點玄之又玄，如果只是記一些算法覺得並不能完全代表這些樹結構存在的意義。
想寫一些對這些樹結構的理解和感受，而不是簡單地重複概念，先不涉及算法代碼相關，只是偶爾有了一些體會，想到哪寫到哪吧。

如能讓你產生一點共鳴，甚好；如不能，就當作者在自言自語。

關於樹

大自然的樹種有萬千種，各有各的模樣和特性，為了生存，各顯神通。二進制世界裡的“樹”也是對大自然的一種映射吧。

事物並不是非黑即白。
樹形結構的出現，是由鏈表進化而來，是從線性到非線性結構的發展。
而鏈表，又何嘗不是一種特殊形狀的樹呢？

看過了好多篇關於樹的技術文章，特別是紅黑樹的，一上來甩出來五條性質，balabala

> 性質：
> 1、節點是紅色或黑色。
> 2、根節點是黑色。
> 3、所有葉子都是黑色（葉子是NIL節點）。
> 4、每個紅色節點必須有兩個黑色的子節點。（從每個葉子到根的所有路徑上不能有兩個連續的紅色節
> 5、從任一節點到其每個葉子的所有簡單路徑都包含相同數目的黑色節點。

@_@

各種文字的概念，各種形狀的樹

如何出現，如何變色，又如何旋轉，讓人看得也天旋地轉，看完還一頭霧水啊~~~~~

忽然覺得，紅黑樹，是道法自然的一種體現。

紅與黑，不在意顏色，亦可看成黑白兩道。

它的定義：一種自平衡二叉查找樹。
自平衡，如何自平衡？平衡自在內心。

事物的出現自有它的道理，那麼紅黑樹是怎樣出現的？
它出現過程大致是：

```
二叉樹 -> 二叉查找樹(BST) -> AVL樹
....... ....... ...... .... .... ..... |
```

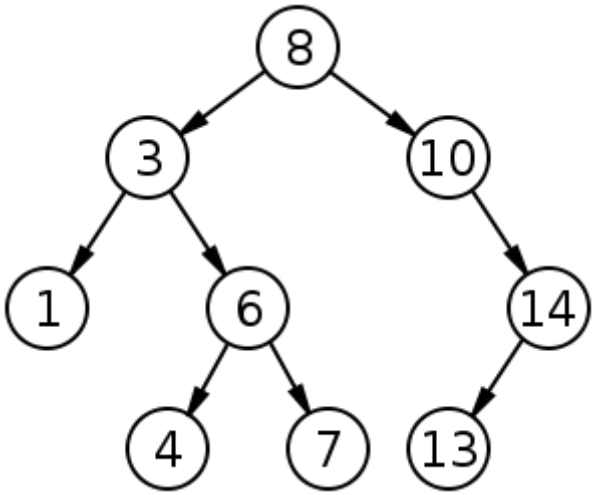
```
（B樹 -> 2-3樹）-> 紅黑樹(RB Tree)
```

BST

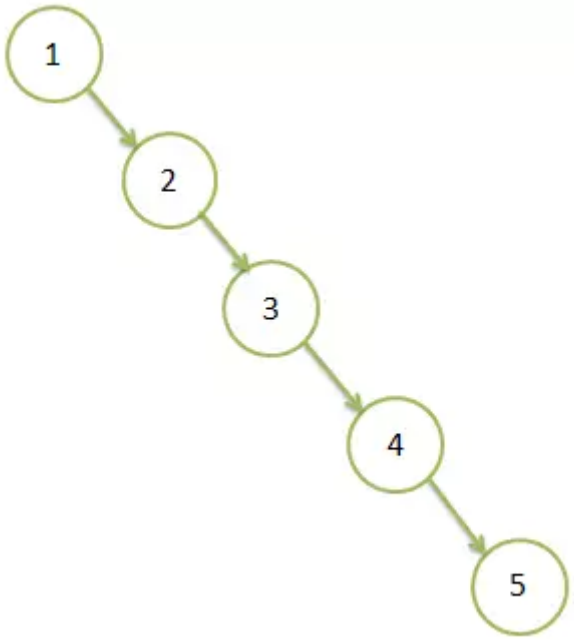
Binary Search Tree

BST先不說來龍去脈了，看名字，二叉查找樹的出現自然是為了查找方便了。

如下，就是一棵BST了。



但是BST在增刪改查的過程中，也可能變成下面這樣：



不想要這樣的情況是不是？這是最壞的運行情況O(N)了，沒有比這樣更壞了。

但它也符合BST的性質，只是看起來就極不平衡，一碰就倒啦，極易失衡呢。

AVL

> **Self**-Balancing Binary Search Tree / Height-Balanced Binary Search Tree

關於名字

名字不好懂？本來就不好懂~

“AVL”名字來源：其發明者前蘇聯學者 G.M. Adelson-Velsky 和 E.M. Landis 名字而命名。

英文定義如下：

```
Definition:
A tree is perfectly balanced if it is empty or the number of nodes in each sul
```

好一個“perfectly”~ But who is perfect?

好了好了，好好說~

> AVL樹是最先發明的自平衡二叉搜索樹（BBST）

從定義名字可以看出來，AVL是為了平衡而來。

它是一種高度平衡樹，也是一種二叉搜索樹，它的出現是為了解決二叉搜索樹退化成鏈表的問題。

性質有：

- 每棵子樹中的左子樹和右子樹的深度差不能超過 1；
- 二叉樹中每棵子樹都要求是平衡二叉樹。

來個圖看看：

> 追求極致的平衡，而極致，也會帶來弊端。
> 要追求什麼，必然要通過另一些什麼來換取。
> 而時刻要保持的“極致”，必然時刻要不鬆懈地維護。這其實挺累的。
> 它只是想要追求平衡的完美吧。
> 像紅樓夢裡的寶釵，賈府上上下下都對她讚不絕口，上至賈母王夫人，下至丫頭，都喜歡她，為人周到
> 事事無絕對。
> 追求極致，就可能出現弊端。或者說，極致點，也是一個弱點。

優缺點

- 優點:
  查找元素時，速度很快。
- 缺點:
  在添加或刪除元素時，由於“追求極致的平衡”，通過不斷地循環來達到自平衡，因而增加了時間複雜度。

AVL旋轉

AVL左旋

2-3樹

在說紅黑樹之前，應該看一下2-3樹，其實可以更好地認識紅黑樹。
（留著空）

紅黑樹（RB Tree）

> 既生瑜，何生亮？

還是來看看英文的定義~ 有時候英文的更好理解。
不要怕，很好懂~

> Definition:
> Red-Black Trees are binary search trees that are named after the way the nodes
> 紅黑樹是二叉搜索樹，它的命名來自於它結點被標記顏色的方式。
> Red-Black Tree **is** a self-balancing Binary Search Tree (BST) where every node
> 它是一棵自平衡二叉搜索樹，遵循以下性質。
> RedBlackTree
> **Every node has a color either red **or** black.
> **Root **of** tree **is** always black.
> **There are no two adjacent red nodes (A red node cannot have a red parent **or**
> **Every path **from** a node (including root) to any **of** its descendant NULL node

看這個結構圖和幾條balabala的性質，是不是覺得很暈？很暈就對了~

先來了解個大概~

RB Tree也是平衡二叉搜索樹，和AVL一樣，都是Self-Balanced。
那它們都什麼區別？有了AVL，為什麼還要有RB Tree?

前面說到過，AVL追求極致的平衡，而紅黑樹的出現，應該是為了彌補追求極致所造成的失衡了，RB Tree只做到部分的平衡，是一種折中的表現。
它放棄的部分平衡，是為了換取增刪操作時的效率。

犧牲了高度的絕對平衡，換來時間的優化。這比交換值不值？
我說值得你同意嗎？

> 能力守恆，價值也是。
> 不同人的價值觀不一樣，但是你自己心裡的天平會平衡就可以。旁人多說無益。

AVL樹，為了平衡會要經過好多次旋轉，旋轉旋轉.....

> 白雪，夏夜，我不停歇.....

而RB Tree的結構設計，使得任何不平衡都會在三次旋轉之內解決。（很厲害！）

有得有失~ 有失有得~
得失，也是一種自平衡。

（它們如何旋轉變化待補充~）

RB Tree的旋轉

左旋

（圖片來源於資料文章）

- 將E的右子樹S繞E逆時針旋轉，使得E的右子樹S成為E的父親，同時修改相關節點的引用。

- 旋轉之後，二叉查找樹的屬性仍然滿足。

```
/** From CLR */
private void rotateLeft(TreeMapEntry<K,V> p) {
if (p != null) {
TreeMapEntry<K,V> r = p.right;
p.right = r.left;
if (r.left != null)
r.left.parent = p;
r.parent = p.parent;
if (p.parent == null)
root = r;
else if (p.parent.left == p)
p.parent.left = r;
else
p.parent.right = r;
r.left = p;
p.parent = r;
}
}
```

右旋

將S的左子樹繞S順時針旋轉，使得S的左子樹成為S的父親，同時修改相關節點的引用。
旋轉之後，二叉查找樹的屬性仍然滿足。

（圖片來源於資料文章）

```
/** From CLR */
private void rotateRight(TreeMapEntry<K,V> p) {
if (p != null) {
TreeMapEntry<K,V> l = p.left;
p.left = l.right;
if (l.right != null) l.right.parent = p;
l.parent = p.parent;
if (p.parent == null)
root = l;
else if (p.parent.right == p)
p.parent.right = l;
else p.parent.left = l;
l.right = p;
p.parent = l;
}
}
```

AVL VS RB Tree

就AVL和RB Tree兩種結構相比較，

- AVL像一位特長生，顯著的特長就是它的“絕對平衡”，鋒芒畢露。
- RB Tree像一位比較全面發展的學生，沒有絕對的特長，但各方面表現都比較好，深諳折中之道。

不能說誰一定優於誰，“絕對”與“非絕對”也是相對（太繞），它們各自有自己的平衡，並且有自己的擅長點。

寫在後面

內容尚需補充與完善，偶有寫著寫著內容飄向別處，隨便看看就好~

參考

- Data Structures and Algorithms
- Red-Black Tree | Set 1 (Introduction)
- 紅黑樹的原理和TreeMap實現

相關文章

來自JVM的靈魂拷問：你是什麼垃圾？

Java虛擬機裡的magicworld

Java那麼多鎖，能鎖住滅霸嗎？

Java線程池你知它有多深

Back to Top