hw5 RBtree VS SkipList VS AVL

1. 测试说明

1.1数据说明

输入数据为0—`size-1`。Size取50,100,2000,5000,10000。分为顺序和乱序两种。

在"./data" 文件夹下，一个"ran_numx.txt"中分为`insert`和`search`两部分。

具体顺序为：

```
Size  #输入大小
0
1
...
Size-1
Size/16 #一个搜索集合大小
...
```

`search` **的数据集**有五组：Size/16 ; Size/8 ; Size/4 ; Size/2 ; Size

顺序的组从0开始，乱序的组用`rand()%Size`生成。

查找的数据皆为已插入的。

1.2 衡量标准

使用了`lab1`提供的函数作为**耗时的度量**，单位为cycles

```
static inline uint64_t rdtsc()
{
    uint32_t low, high;
    asm volatile ("rdtsc" : "=a" (low), "=d" (high));
    return ((uint64_t)high << 32) | low;
}
```

旋转次数测试中，认为左旋、右旋为一次旋转。

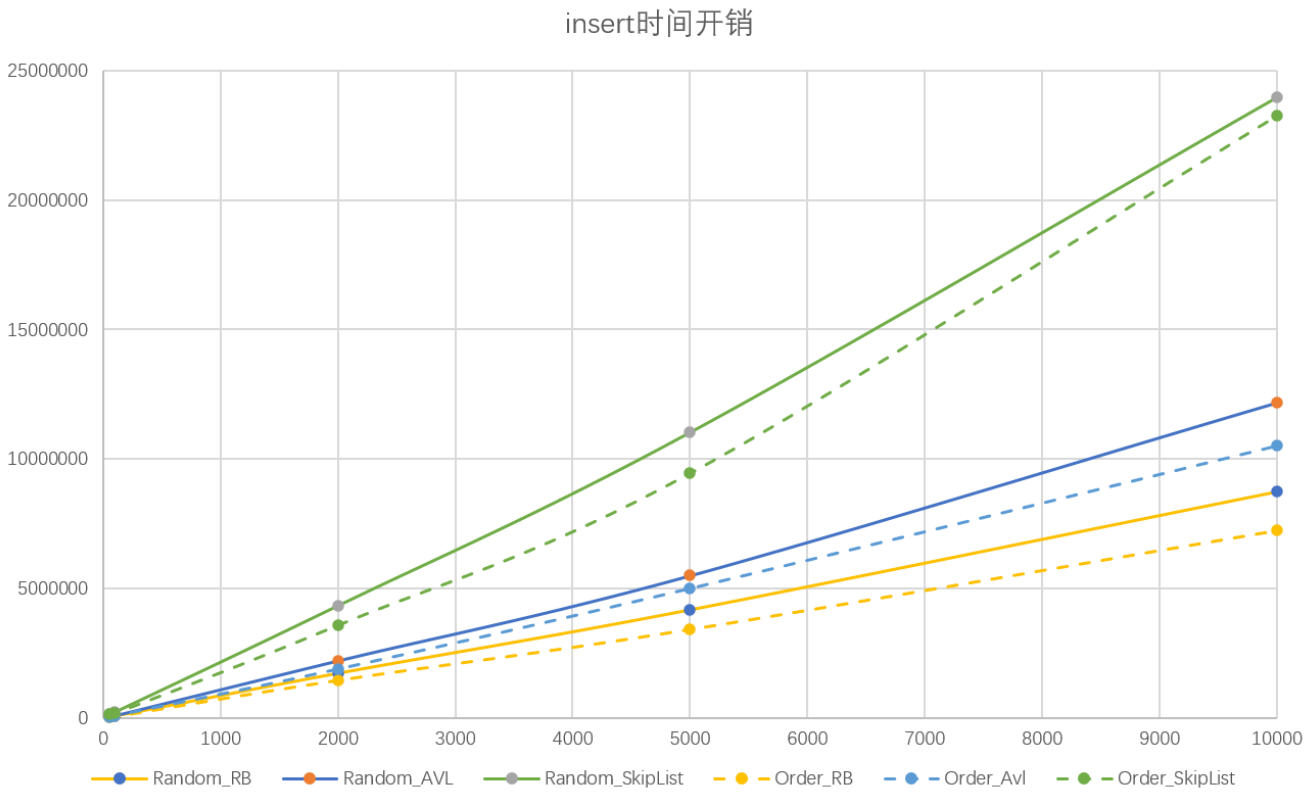
2. 实验数据

2.1 insert

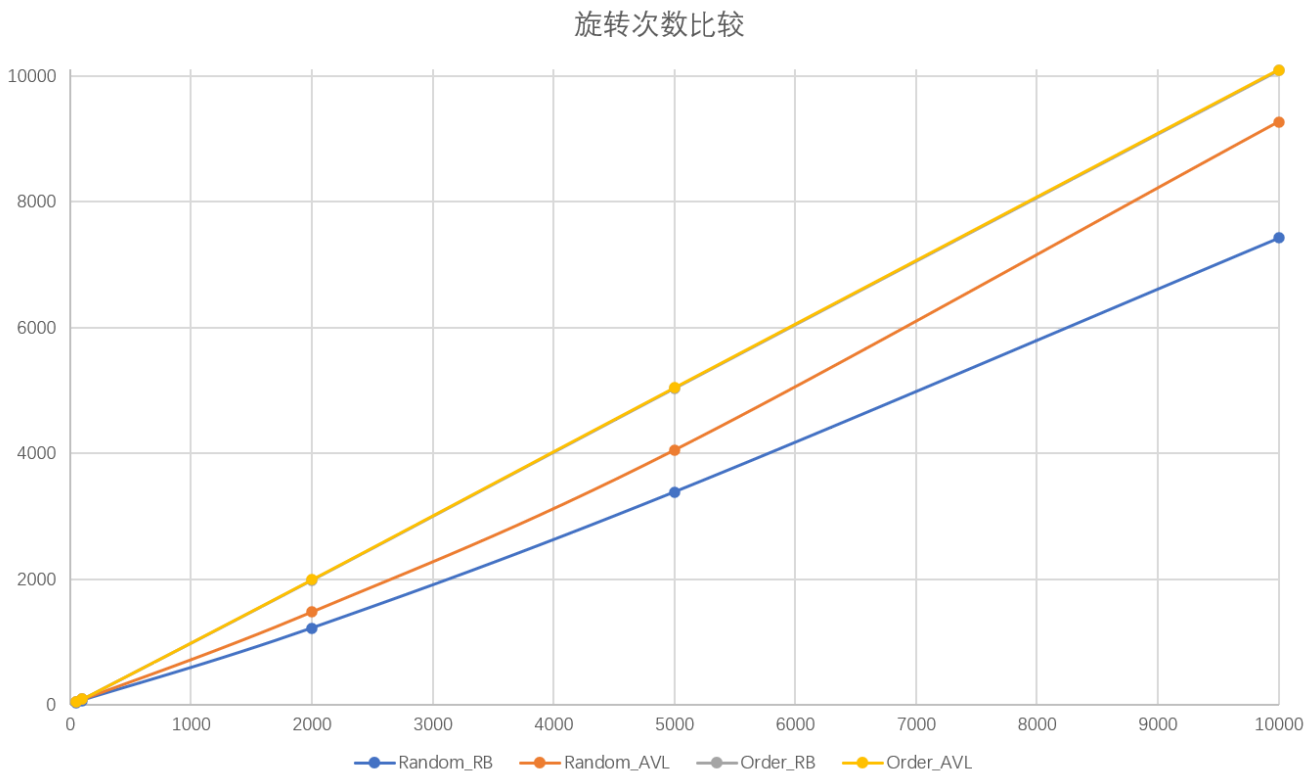
表1 insert时间开销和旋转次数

	Size	Random(Time)			Order(Time)			Random(Rotate)		Order(Rotate)	
		Rbtree	AVL	SkipList	Rbtree	AVL	SkipList	Rbtree	AVL	Rbtree	AVL
insert	50	79474	83104	139848	30260	44124	154414	32	42	41	44
	100	78290	96124	229462	63398	78656	176080	70	88	90	93
	2000	1727938	2214192	4337792	1466888	1883380	3580664	1219	1479	1984	1991
	5000	4177164	5497780	11028564	3431674	5002336	9443590	3383	4050	5031	5040
	10000	8733218	12177082	23966888	7246140	10520822	23249310	7419	9264	10081	10090

对其中的时间开销、旋转次数分别绘制了以下两张图。



- 上图展示了不同Size下，各种数据结构 `insert` 操作的时间开销。
- 其中虚线为顺序插入，实线为乱序，同一种颜色为一种数据结构
- 理论上来说：顺序插入会让AVL、RB树的退化成单链表，需要更多次旋转。
- 实验结果似乎两者耗时区别不大，甚至顺序插入更好，可能的原因有：
  - 数据集较小，使得乱序顺序效率差别不大。
  - 使用 `random_shuffle` 打扰数据，可能打乱的效果不好
  - 但根据下图，旋转次数确实是乱序时候少，那么可能是编译器对顺序的数组进行了未知的优化



- 上图展示了不同Size下，`AVL`和`RBtree`在`insert`操作的旋转次数。
- 上方两条为顺序插入结果（由于太接近，重合在一起），下方为乱序
- 理论上来说：顺序插入会让AVL、RB树的退化成单链表，需要更多次旋转。而RBtree的旋转次数比Avltree更好
- 实验结果与理论吻合较好：
  - 顺序插入旋转次数明显多余乱序
  - 在乱序中，`RBtree`明显比`AVL`旋转次数少。在顺序中由于都退化成单链表，旋转次数一样

2.2 search

表2 search时间消耗对比

total_size	search_size	Random						Order					
		Rbtree		SkipList		AVL		Rbtree		SkipList		AVL	
		time	average	time	average	time	average	time	average	time	average	time	average
50	3	2142	714	3074	1024	2814	938	1628	542	3210	1070	4042	1347
	6	2570	428	3320	553	4356	726	2266	377	4376	729	4672	778
	12	5398	449	6338	528	6430	535	5204	433	8012	667	6838	569
	25	9636	385	10478	419	12662	506	12522	500	12970	518	13002	520
	50	17074	341	18110	362	24286	485	24042	480	18794	375	23252	465
100	6	3216	536	3414	569	4960	826	2746	457	3184	530	3608	601
	12	5090	424	6462	538	11006	917	5162	430	6580	548	19280	1606
	25	10900	436	21190	847	15996	639	9368	374	9732	389	11942	477
	50	20820	416	24420	488	31718	634	20414	408	19652	393	25558	511
	100	38132	381	37116	371	53372	533	38654	386	57636	576	45278	452
2000	125	76830	614	93800	750	154476	1235	47206	377	49588	396	67122	536
	250	186172	744	205786	823	289036	1156	177598	710	128090	512	159134	636
	500	238754	477	280772	561	397590	795	209334	418	200596	401	293676	587
	1000	589880	589	596466	596	873616	873	428174	428	437858	437	615096	615
	2000	1069716	534	1120728	560	1612026	806	882198	441	874874	437	1307338	653
5000	312	191274	613	198194	635	302528	969	138094	442	126890	406	199570	639
	625	433344	693	409246	654	690052	1104	290050	464	290598	464	407884	652
	1250	747304	597	734824	587	1249008	999	529910	423	546360	437	773114	618
	2500	1424194	569	1422362	568	2431572	972	1107674	443	1143020	457	1704592	681
	5000	2973164	594	2815770	563	4838632	967	2213708	442	2279448	455	3644108	728
10000	625	445344	712	654086	1046	1101196	1761	358488	573	310182	496	436448	698
	1250	900528	720	963658	770	1647766	1318	598568	478	565838	452	822566	658
	2500	1747428	698	1717992	687	3093840	1237	1138634	455	1122086	448	1729276	691
	5000	3224596	644	3336782	667	6303340	1260	2255244	451	2234724	446	3544096	708
	10000	6723202	672	6291708	629	12171382	1217	4891354	489	4756294	475	6932664	693

对于每个大小为`total_size`的数据集，分别进行了：Size/16 ; Size/8 ; Size/4 ; Size/2 ; Size大小的五次查找。并求取了平均查找时长`average`

- 可以看到，在数据比较大的时候，红黑树和跳表都比AVL树有明显优势。红黑树利用染色来优化旋转，跳表用概率维持平衡，都比AVL单纯的旋转来得更高效。