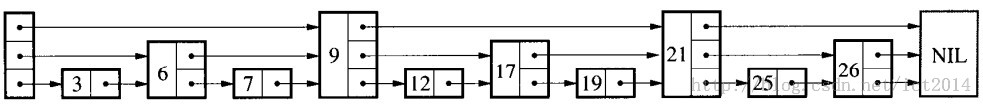
**一：基本概念**

跳跃表是一种随机化的数据结构，在查找、插入和删除这些字典操作上，其效率可比拟于平衡二叉树（如红黑树），大多数操作只需要O(log n)平均时间，但它的代码以及原理更简单。跳跃表的定义如下：

“Skip lists  are data structures  that use probabilistic  balancing rather  than  strictly  enforced balancing. As a result, the algorithms  for insertion and deletion in skip lists  are much simpler and significantly  faster  than  equivalent  algorithms  for balanced trees.”

译文：跳跃表使用概率平衡，而不是强制平衡，因此，对于插入和删除结点比传统上的平衡树算法更为简洁高效。

跳跃表基于有序单链表，在链表的基础上，每个结点不只包含一个指针，还可能包含多个指向后继结点的指针，这样就可以跳过一些不必要的结点，从而加快查找、删除等操作。如下图就是一个跳跃表：



传统的单链表是一个线性结构，向有序的链表中插入、查找一个结点需要O(n)的时间。如果使用上图的跳跃表，就可以减少查找所需的时间。

跳跃表的插入和删除操作都基于查找操作，理解了查找操作，也就理解了跳跃表的本质。查找就是给定一个key，查找这个key是否出现在跳跃表中。

结合上图，如果想查找19是否存在，从最高层开始，首先和头结点的最高层的后继结点9进行比较，19大于9，因此接着和9在该层上的后继结点21进行比较，小于21，那这个值肯定在9结点和21结点之间。

因此，下移一层，接着和9在该层上的后继结点17进行比较，19大于17，然后和21进行比较，小于21，此时肯定在17结点和21结点之间。

接着下移一层，和17在该层上的后继结点19进行比较，这样就最终找到了。

上面就是跳跃表的基本思想，跳跃表结点包含多少个指向后继元素的指针，是通过一个随机函数生成器得到的。这就是为什么论文“Skip Lists : A Probabilistic Alternative to Balanced Trees ”中有“概率”的原因了，就是通过随机生成一个结点中指向后续结点的指针数目。

**二：Redis中的跳跃表**

Redis使用跳跃表作为有序集合键的底层实现之一，若一个有序集合包含的元素数量比较多，或者有序集合中的成员是比较长的字符串时，Redis就会使用跳跃表来作为有序集合键的底层实现。

有序集合使用两种数据结构来实现，从而可以使插入和删除操作达到O(log(N))的时间复杂度。这两种数据结构是哈希表和跳跃表。向哈希表添加元素，用于将成员对象映射到分数；同时将该元素添加到跳跃表，以分数进行排序。

和链表、字典等数据结构被广泛地应用在Redis内部不同，Redis只在两个地方用到了跳跃表，一个是实现有序集合键，另一个是在集群结点中用作内部数据结构。除此之外，跳跃表在Redis里面没有其他用途。

Redis的跳跃表实现跟William Pugh在"Skip Lists: A Probabilistic Alternative to Balanced Trees"中描述的跳跃表算法类似，只是有三点不同：

a、允许重复分数；

b、排序不止根据分数，还可能根据成员对象（当分数相同时）；

c、有一个前继指针，因此在第1层，就形成了一个双向链表，从而可以方便的从表尾向表头遍历，用于ZREVRANGE命令的实现。

Redis跳跃表的相关结构体定义在redis.h中，实现在t\_zset.c中。

1：跳跃表结点

在redis.h中定义了结构体zskiplistNode表示跳跃表结点，它的定义如下：

**typedef** struct zskiplistNode **{**

robj **\***obj**;**

double score**;**

struct zskiplistNode **\***backward**;**

struct zskiplistLevel **{**

struct zskiplistNode **\***forward**;**

unsigned int span**;**

**}** level**[];**

**}** zskiplistNode**;**

obj是该结点的成员对象指针，score是该对象的分值，是一个浮点数，跳跃表中的所有结点，都是根据score从小到大来排序的。

同一个跳跃表中，各个结点保存的成员对象必须是唯一的，但是多个结点保存的分值却可以是相同的：分值相同的结点将按照成员对象的字典顺序从小到大进行排序。

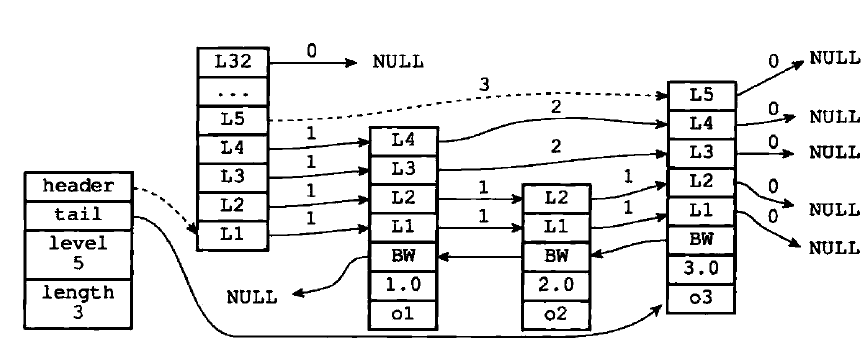
level数组是一个柔性数组成员，它可以包含多个元素，每个元素都包含一个层指针(level[i].forward)，指向该结点在本层的后继结点。该指针用于从表头向表尾方向访问结点。可以通过这些层指针来加快访问结点的速度。

每次创建一个新跳跃表结点的时候，程序都根据幂次定律（power law，越大的数出现的概率越小）随机生成一个介于1和32之间的值作为level数组的大小，这个大小就是该结点包含的层数。

Redis中的跳跃表，与普通跳跃表的区别之一，就是包含了层跨度(level[i].span)的概念。这是因为在有序集合支持的命令中，有些跟元素在集合中的排名有关，比如获取元素的排名，根据排名获取、删除元素等。通过跳跃表结点的层跨度，可以快速得到该结点在跳跃表中的排名。

计算结点的排名，就是在查找某个结点的过程中，将沿途访问过的所有结点的层跨度累加起来，得到的结果就是目标结点在跳跃表中的排名。

层跨度用于记录本层两个相邻结点之间的距离，举个例子，如下图的跳跃表：



跳跃表头结点（header指向的节点）排名为0，之后的节点排名以此类推。在上图跳跃表中查找计算分值为3.0、成员对象为o3的结点的排名。查找过程只遍历了头结点的L5层就找到了，并且头结点该层的跨度为3，因此得到该结点在跳跃表中的排名为3。

如果要查找分值为2.0、成员对象为o2的结点，查找结点的过程中，首先经过头结点的L4层，然后是o1结点的L2层，也就是经过了两个层跨度为1的结点，因此得到目标结点在跳跃表中的排名为2。

Redis中的跳跃表，与普通跳跃表的另一个区别就是，每个结点还有一个前继指针backward。可用于从表尾向表头方向访问结点。通过结点的前继指针，组成了一个普通的链表。因为每个结点只有一个前继指针，所以只能依次访问结点，而不能跳过结点。

2：跳跃表

在redis.h中定义了结构体zskiplist表示跳跃表，它的定义如下：

**typedef** struct zskiplist **{**

struct zskiplistNode **\***header**,** **\***tail**;**

unsigned long length**;**

int level**;**

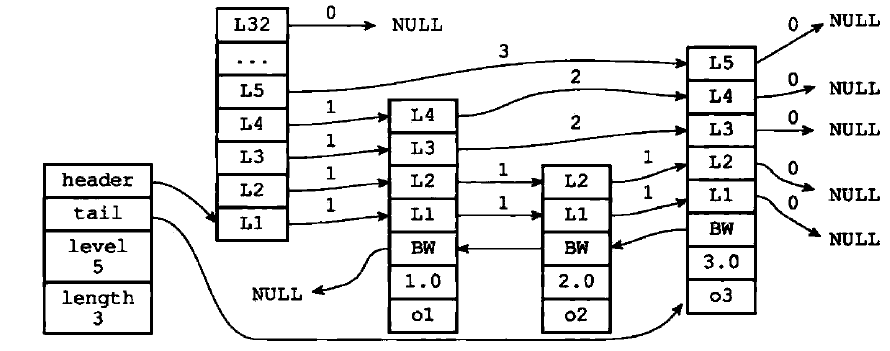
**}** zskiplist**;**

header和tail指针分别指向跳跃表的表头结点和表尾结点，通过这两个指针，定位表头结点和表尾结点的复杂度为O(1)。表尾结点是表中最后一个结点。而表头结点实际上是一个伪结点，该结点的成员对象为NULL，分值为0，它的层数固定为32（层的最大值）。

length属性记录结点的数最，程序可以在O(1)的时间复杂度内返回跳跃表的长度。

level属性记录跳跃表的层数，也就是表中层高最大的那个结点的层数，注意，表头结点的层高并不计算在内。

下面就是一个zskiplist表示的跳跃表：



**三：实现**

1：随机算法

一个具有k个后继指针的结点称为k层结点。假设k层结点的数量是k+1层结点的P倍，那么其实这个跳跃表可以看成是一棵平衡的P叉树。跳跃表结点的层数，采用随机化算法得到，实现如下：

int zslRandomLevel**(**void**)** **{**

int level **=** 1**;**

**while** **((**random**()&**0xFFFF**)** **<** **(**ZSKIPLIST\_P **\*** 0xFFFF**))**

level **+=** 1**;**

**return** **(**level**<**ZSKIPLIST\_MAXLEVEL**)** **?** level **:** ZSKIPLIST\_MAXLEVEL**;**

**}**

这里的ZSKIPLIST\_P是0.25。上述代码中，level初始化为1，然后，如果持续满足条件：(random()&0xFFFF) < (ZSKIPLIST\_P \* 0xFFFF)的话，则level+=1。最终调整level的值，使其小于ZSKIPLIST\_MAXLEVEL。

理解该算法的核心，就是要理解满足条件：(random()&0xFFFF) < (ZSKIPLIST\_P \* 0xFFFF)的概率是多少？

random()&0xFFFF形成的数，均匀分布在区间[0, 0xFFFF]上，那么这个数小于(ZSKIPLIST\_P \* 0xFFFF)的概率是多少呢？自然就是ZSKIPLIST\_P，也就是0.25了。

因此，最终返回level为1的概率是1-0.25=0.75，返回level为2的概率为0.25\*0.75，返回level为3的概率为0.25\*0.25\*0.75......因此，如果返回level为k的概率为x，则返回level为k+1的概率为0.25\*x，换句话说，如果k层的结点数是x，那么k+1层就是0.25\*x了。这就是所谓的幂次定律（power law），越大的数出现的概率越小。

测试代码如下：

void test\_zslRandomLevel**(){**

unsigned int trytimes **=** 0xffffff**;**

unsigned int i **=** 0**;**

int resset**[**33**]** **=** **{**trytimes**,};**

double percent **=** 0.0**;**

**for(**i **=** 0**;** i **<** trytimes**;** i**++){**

resset**[**zslRandomLevel**()]++;**

**}**

**for(**i **=** 1**;** i **<=** 32**;** i**++){**

**if(**resset**[**i**-**1**]** **==** 0**){**

percent **=** 0.0**;**

**}**

**else{**

percent **=** **(**double**)**resset**[**i**]/**resset**[**i**-**1**];**

**}**

printf**(**"resset[%u] is %d, percentage of resset[%u] is %f%%\n"**,**

i**,** resset**[**i**],** i**-**1**,** percent**);**

**}**

**}**

结果如下：

resset**[**1**]** is 12583714**,** percentage of resset**[**0**]** is 0.750048**%**

resset**[**2**]** is 3146005**,** percentage of resset**[**1**]** is 0.250006**%**

resset**[**3**]** is 785421**,** percentage of resset**[**2**]** is 0.249657**%**

resset**[**4**]** is 196516**,** percentage of resset**[**3**]** is 0.250205**%**

resset**[**5**]** is 49350**,** percentage of resset**[**4**]** is 0.251125**%**

resset**[**6**]** is 12163**,** percentage of resset**[**5**]** is 0.246464**%**

resset**[**7**]** is 3024**,** percentage of resset**[**6**]** is 0.248623**%**

resset**[**8**]** is 748**,** percentage of resset**[**7**]** is 0.247354**%**

resset**[**9**]** is 216**,** percentage of resset**[**8**]** is 0.288770**%**

resset**[**10**]** is 46**,** percentage of resset**[**9**]** is 0.212963**%**

resset**[**11**]** is 12**,** percentage of resset**[**10**]** is 0.260870**%**

resset**[**12**]** is 0**,** percentage of resset**[**11**]** is 0.000000**%**

resset**[**13**]** is 0**,** percentage of resset**[**12**]** is 0.000000**%**

resset**[**14**]** is 0**,** percentage of resset**[**13**]** is 0.000000**%**

resset**[**15**]** is 0**,** percentage of resset**[**14**]** is 0.000000**%**

resset**[**16**]** is 0**,** percentage of resset**[**15**]** is 0.000000**%**

resset**[**17**]** is 0**,** percentage of resset**[**16**]** is 0.000000**%**

resset**[**18**]** is 0**,** percentage of resset**[**17**]** is 0.000000**%**

resset**[**19**]** is 0**,** percentage of resset**[**18**]** is 0.000000**%**

resset**[**20**]** is 0**,** percentage of resset**[**19**]** is 0.000000**%**

resset**[**21**]** is 0**,** percentage of resset**[**20**]** is 0.000000**%**

resset**[**22**]** is 0**,** percentage of resset**[**21**]** is 0.000000**%**

resset**[**23**]** is 0**,** percentage of resset**[**22**]** is 0.000000**%**

resset**[**24**]** is 0**,** percentage of resset**[**23**]** is 0.000000**%**

resset**[**25**]** is 0**,** percentage of resset**[**24**]** is 0.000000**%**

resset**[**26**]** is 0**,** percentage of resset**[**25**]** is 0.000000**%**

resset**[**27**]** is 0**,** percentage of resset**[**26**]** is 0.000000**%**

resset**[**28**]** is 0**,** percentage of resset**[**27**]** is 0.000000**%**

resset**[**29**]** is 0**,** percentage of resset**[**28**]** is 0.000000**%**

resset**[**30**]** is 0**,** percentage of resset**[**29**]** is 0.000000**%**

resset**[**31**]** is 0**,** percentage of resset**[**30**]** is 0.000000**%**

resset**[**32**]** is 0**,** percentage of resset**[**31**]** is 0.000000**%**

可见，层数分布基本上是符合预期的。

2：插入

向跳跃表插入新的结点，代码如下：

zskiplistNode **\***zslInsert**(**zskiplist **\***zsl**,** double score**,** robj **\***obj**)** **{**

zskiplistNode **\***update**[**ZSKIPLIST\_MAXLEVEL**],** **\***x**;**

unsigned int rank**[**ZSKIPLIST\_MAXLEVEL**];**

int i**,** level**;**

redisAssert**(!**isnan**(**score**));**

x **=** zsl**->**header**;**

**for** **(**i **=** zsl**->**level**-**1**;** i **>=** 0**;** i**--)** **{**

*/\* store rank that is crossed to reach the insert position \*/*

rank**[**i**]** **=** i **==** **(**zsl**->**level**-**1**)** **?** 0 **:** rank**[**i**+**1**];**

**while** **(**x**->**level**[**i**].**forward **&&**

**(**x**->**level**[**i**].**forward**->**score **<** score **||**

**(**x**->**level**[**i**].**forward**->**score **==** score **&&**

compareStringObjects**(**x**->**level**[**i**].**forward**->**obj**,**obj**)** **<** 0**)))** **{**

rank**[**i**]** **+=** x**->**level**[**i**].**span**;**

x **=** x**->**level**[**i**].**forward**;**

**}**

update**[**i**]** **=** x**;**

**}**

*/\* we assume the key is not already inside, since we allow duplicated*

*\* scores, and the re-insertion of score and redis object should never*

*\* happen since the caller of zslInsert() should test in the hash table*

*\* if the element is already inside or not. \*/*

level **=** zslRandomLevel**();**

**if** **(**level **>** zsl**->**level**)** **{**

**for** **(**i **=** zsl**->**level**;** i **<** level**;** i**++)** **{**

rank**[**i**]** **=** 0**;**

update**[**i**]** **=** zsl**->**header**;**

update**[**i**]->**level**[**i**].**span **=** zsl**->**length**;**

**}**

zsl**->**level **=** level**;**

**}**

x **=** zslCreateNode**(**level**,**score**,**obj**);**

**for** **(**i **=** 0**;** i **<** level**;** i**++)** **{**

x**->**level**[**i**].**forward **=** update**[**i**]->**level**[**i**].**forward**;**

update**[**i**]->**level**[**i**].**forward **=** x**;**

*/\* update span covered by update[i] as x is inserted here \*/*

x**->**level**[**i**].**span **=** update**[**i**]->**level**[**i**].**span **-** **(**rank**[**0**]** **-** rank**[**i**]);**

update**[**i**]->**level**[**i**].**span **=** **(**rank**[**0**]** **-** rank**[**i**])** **+** 1**;**

**}**

*/\* increment span for untouched levels \*/*

**for** **(**i **=** level**;** i **<** zsl**->**level**;** i**++)** **{**

update**[**i**]->**level**[**i**].**span**++;**

**}**

x**->**backward **=** **(**update**[**0**]** **==** zsl**->**header**)** **?** **NULL** **:** update**[**0**];**

**if** **(**x**->**level**[**0**].**forward**)**

x**->**level**[**0**].**forward**->**backward **=** x**;**

**else**

zsl**->**tail **=** x**;**

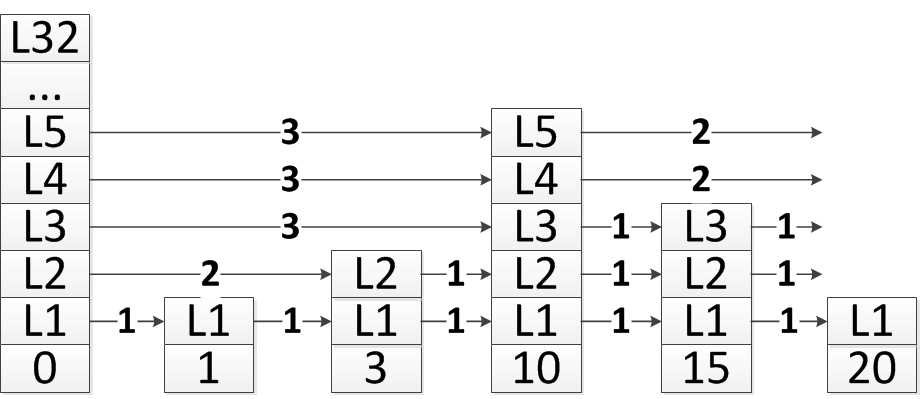
zsl**->**length**++;**

**return** x**;**

**}**

因为Redis中的跳跃表加入了层跨度的概念，因此比常规的跳跃表插入稍微复杂一些。这里主要使用了update和rank辅助数组（常规跳跃表的插入只需要update数组）。其中，update数组记录插入结点在每层上的前驱结点，而rank数组则记录该结点在跳跃表中的排名，这里表头（伪）结点排名为0，以此类推。结点的排名，等于查找该结点时，之前所遍历过的结点的层跨度之和。

下图是一个简化的跳跃表，每个结点只保留了分数、层指针和层跨度。所以，下图中表头结点排名为0，分数为1、3、10、15、20的结点，排名分别为1、2、3、4、5。



首先看插入代码的第一部分，也就是寻找插入结点在每层上的前驱结点的代码：

x **=** zsl**->**header**;**

**for** **(**i **=** zsl**->**level**-**1**;** i **>=** 0**;** i**--)** **{**

*/\* store rank that is crossed to reach the insert position \*/*

rank**[**i**]** **=** i **==** **(**zsl**->**level**-**1**)** **?** 0 **:** rank**[**i**+**1**];**

**while** **(**x**->**level**[**i**].**forward **&&**

**(**x**->**level**[**i**].**forward**->**score **<** score **||**

**(**x**->**level**[**i**].**forward**->**score **==** score **&&**

compareStringObjects**(**x**->**level**[**i**].**forward**->**obj**,**obj**)** **<** 0**)))** **{**

rank**[**i**]** **+=** x**->**level**[**i**].**span**;**

x **=** x**->**level**[**i**].**forward**;**

**}**

update**[**i**]** **=** x**;**

**}**

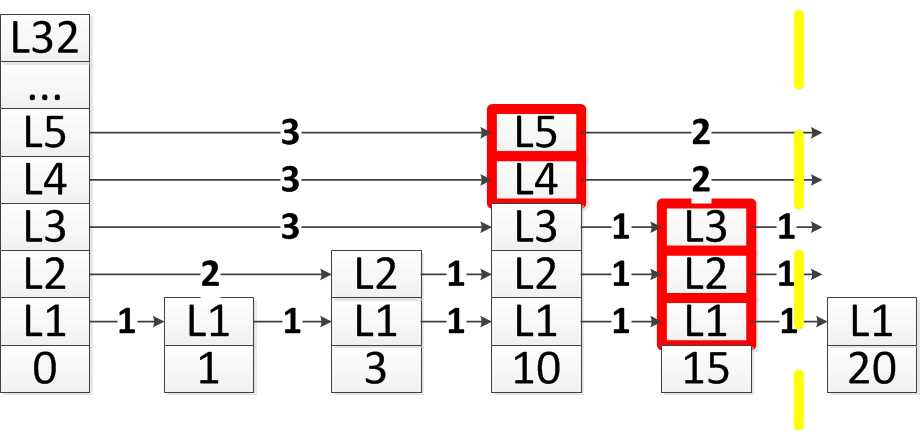
从表头结点的最高层开始查找，首先在该层中寻找插入结点的前驱结点。只要插入结点比当前结点x在该层的后继结点x->level[i].forward要大，则首先记录x后继结点的排名：rank[i] += x->level[i].span; 接着开始比较x的后继结点：x = x->level[i].forward。

注意，因为Redis中的跳跃表中，允许分数重复而不允许成员对象重复。所以，这里的判断条件中，一旦分数相同，则要比较成员对象的字典顺序。

一旦当前结点x的后继结点为空，或者后继结点比插入结点要大，说明找到了插入结点在该层的前驱结点，记录到update数组中：update[i] = x，此时，rank[i]就是结点x的排名。

然后，开始遍历下一层，从x结点开始比较。

在上图的跳跃表中，假设现在要插入的结点分数为17，黄色虚线所标注的，就是插入新结点的位置。下面标注红色的，就是在每层找到的插入结点的前驱结点，记录在update[i]中，而rank[i]记录了update[i]在跳跃表中的排名，因此，rank[4] = 3, rank[3] = 3, rank[2] = 4, rank[1] = 4, rank[0] = 4。



剩下的代码就是将结点插入到跳跃表中，首先是：

level **=** zslRandomLevel**();**

**if** **(**level **>** zsl**->**level**)** **{**

**for** **(**i **=** zsl**->**level**;** i **<** level**;** i**++)** **{**

rank**[**i**]** **=** 0**;**

update**[**i**]** **=** zsl**->**header**;**

update**[**i**]->**level**[**i**].**span **=** zsl**->**length**;**

**}**

zsl**->**level **=** level**;**

**}**

首先利用zslRandomLevel，生成一个随机的层数level。如果该level大于当前跳跃表的最大level的话，则需要初始化插入结点在超出层上，也就是在层数[zsl->level, level)上的前驱结点及其排名。这里直接初始化前驱结点为头结点，排名为0，并且初始化前驱结点在相应层上的层跨度为zsl->length，也就是头结点和尾节点之间的距离。

然后更新zsl->level的值。需要注意的是，因Redis中，使用哈希表和跳跃表两种结构表示有序集合，因此，在跳跃表的插入操作中，无需判断插入结点是否与表中结点重复，这是因为在调用zslInsert之前，调用者应该已经使用哈希表进行过检测了。

接下来：

x **=** zslCreateNode**(**level**,**score**,**obj**);**

**for** **(**i **=** 0**;** i **<** level**;** i**++)** **{**

x**->**level**[**i**].**forward **=** update**[**i**]->**level**[**i**].**forward**;**

update**[**i**]->**level**[**i**].**forward **=** x**;**

*/\* update span covered by update[i] as x is inserted here \*/*

x**->**level**[**i**].**span **=** update**[**i**]->**level**[**i**].**span **-** **(**rank**[**0**]** **-** rank**[**i**]);**

update**[**i**]->**level**[**i**].**span **=** **(**rank**[**0**]** **-** rank**[**i**])** **+** 1**;**

**}**

*/\* increment span for untouched levels \*/*

**for** **(**i **=** level**;** i **<** zsl**->**level**;** i**++)** **{**

update**[**i**]->**level**[**i**].**span**++;**

**}**

首先调用zslCreateNode创建一个跳跃表节点。然后在层数[0, level)中，根据update[i]记录的每层上的前驱结点，将新结点插入到每层中。

接着更新每层上，新结点及其前驱结点的层跨度。节点的层跨度，等于该节点在第i层上的后继节点的排名，减去该节点的排名。

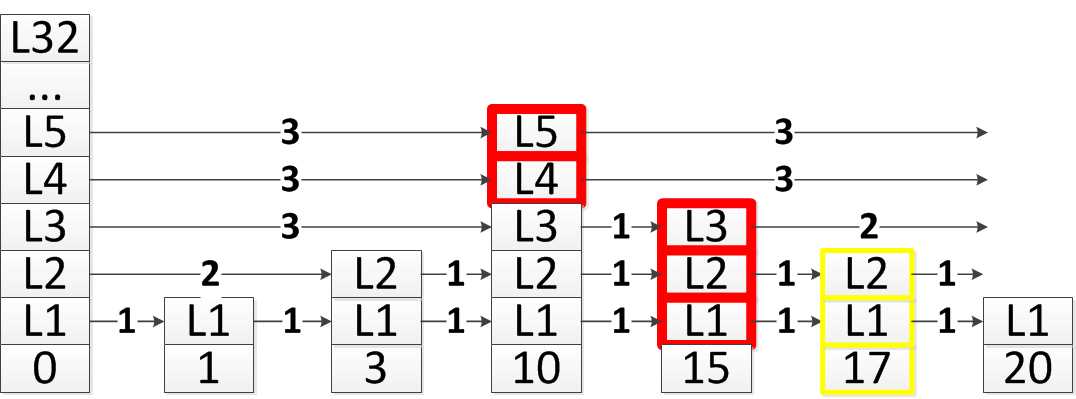
新结点在第i层的后继节点，也就是之前update[i]的后继节点，它的排名是update[i]->level[i].span + rank[i]，插入新结点之后，它的排名加1，也就是update[i]->level[i].span + rank[i] + 1。新结点的排名，就是rank[0] + 1，因此，新结点在第i层的层跨度就是(update[i]->level[i].span + rank[i] + 1) – (rank[0] + 1)，也就是update[i]->level[i].span - (rank[0] - rank[i])。

前驱结点update[i]的层跨度，等于新结点的排名rank[0] + 1，减去update[i]的排名rank[i]，也就是(rank[0] + 1) - rank[i]，也就是(rank[0] - rank[i]) + 1。

针对新增的层数，也就是在[原zsl->level, level)的层中，新结点在层i中的后继结点，就相当于尾结点，尾结点的排名，在插入新结点后，就是zsl-> length + 1。所以，这些层中，新结点的层跨度为(zsl->length + 1) – (rank[0] + 1)，因这些层中的前驱结点update[i]的层跨度初始化为zsl->length，排名rank[i]为0，因此新结点的层跨度(zsl->length + 1) – (rank[0] + 1)等于update[i]->level[i].span - (rank[0] - rank[i])。而且，前驱结点update[i]的层跨度，也就等于(rank[0] - rank[i]) + 1。这也就是为什么在超出层中，初始化rank[i]为0，update[i]->level[i].span为zsl->length的原因了。

最后，如果新结点层数level小于zsl->level，则在[level, zsl->level)中，所有找到的前驱结点的层跨度要加1.

因此，插入新结点17后，效果如下：



最后，就是更新新结点x，及其后继节点的前驱指针。并更新跳跃表的长度。

x**->**backward **=** **(**update**[**0**]** **==** zsl**->**header**)** **?** **NULL** **:** update**[**0**];**

**if** **(**x**->**level**[**0**].**forward**)**

x**->**level**[**0**].**forward**->**backward **=** x**;**

**else**

zsl**->**tail **=** x**;**

zsl**->**length**++;**

3：获取节点的排名

unsigned long zslGetRank**(**zskiplist **\***zsl**,** double score**,** robj **\***o**)** **{**

zskiplistNode **\***x**;**

unsigned long rank **=** 0**;**

int i**;**

x **=** zsl**->**header**;**

**for** **(**i **=** zsl**->**level**-**1**;** i **>=** 0**;** i**--)** **{**

**while** **(**x**->**level**[**i**].**forward **&&**

**(**x**->**level**[**i**].**forward**->**score **<** score **||**

**(**x**->**level**[**i**].**forward**->**score **==** score **&&**

compareStringObjects**(**x**->**level**[**i**].**forward**->**obj**,**o**)** **<=** 0**)))** **{**

rank **+=** x**->**level**[**i**].**span**;**

x **=** x**->**level**[**i**].**forward**;**

**}**

*/\* x might be equal to zsl->header, so test if obj is non-NULL \*/*

**if** **(**x**->**obj **&&** equalStringObjects**(**x**->**obj**,**o**))** **{**

**return** rank**;**

**}**

**}**

**return** 0**;**

**}**

从跳跃表zsl中，得到分数为score，成员为o的结点的排名。若找到了该节点，则返回该结点的排名；没找到返回0。

该函数从头结点的最高层开始，寻找每层上最后一个小于等于寻找结点的结点，找到之后，判断该结点是否就是要寻找的结点。若是则返回其排名，不是则接着从下一层开始寻找，直到level[0]。

有关跳跃表的其他代码解析，可以参阅：

https://github.com/gqtc/redis-3.0.5/blob/master/redis-3.0.5/src/t\_zset.c

参考：

http://dsqiu.iteye.com/blog/1705530

<http://blog.csdn.net/ict2014/article/details/17394259>

http://blog.csdn.net/kisimple/article/details/38706729