44｜跳表：为什么Redis用跳表实现而MySQL用B+树?

2023-05-24 王健伟 来自北京

《快速上手C++数据结构与算法》

你好，我是王健伟。

字符串方面知识的讲解告一段落后，这次，我们讲一讲跳表相关的知识，也讲一讲大家所关心的一个问题——为什么 Redis 用跳表实现而 MySQL 用 B+ 树实现。

在跳表中查询及复杂度分析

回顾以往学习过的数组（线性表的顺序存储）和链表（线性表的链式存储）这两种数据结构，它们各有特点。数组查找速度非常快，但插入、删除速度很慢（要挪动数据）。而链表插入、删除速度快，但查找数据慢。

假设在一个数组中的数据是有序（比如从小到大）排列的，此时若需要快速查询某个元素，那么进行折半（二分）查找是很快就可以找到该元素或者确认该元素不存在的。但如果这组有序的数据并不是用数组而是用链表保存的，那么如何快速查找数据呢?

于是在 1989 年，美国的一位计算机科学家发明了一种新的数据结构——跳跃表，简称跳表。跳表本身基于链表，是对链表的优化（改进版的链表）。跳表这种数据结构因为出现得比较晚，所以很多老项目中并没有采用。

跳表是只能在链表中元素有序的情况下使用的数据结构，即跳表中的元素必须有序。其插入、删除、搜索的时间复杂度都是 O(log_2^n)。它的最大特点是实现相对简单，效率更高，在一些流行项目比如 Redis、LevelDB 中常被用来代替平衡二叉树（AVL 树）或二分查找。

咱们说一点具体的。图 1 是一个普通的有序单链表，查询元素所需要的时间复杂度为 O(n)，插入和删除元素这两个动作如果排除寻找插入或删除位置所需要的时间，那么它的时间复杂度为 O(1)。

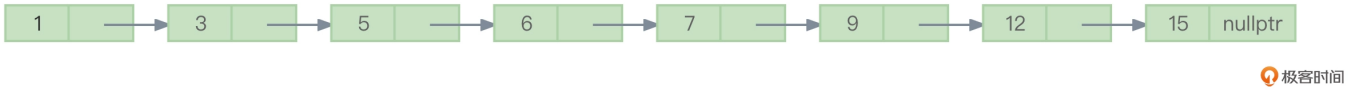

图1 普通的有序单链表

对于图 1，如何进行优化来减少它的时间复杂度呢？我们给上述链表添加一级索引，如图 2 所示：

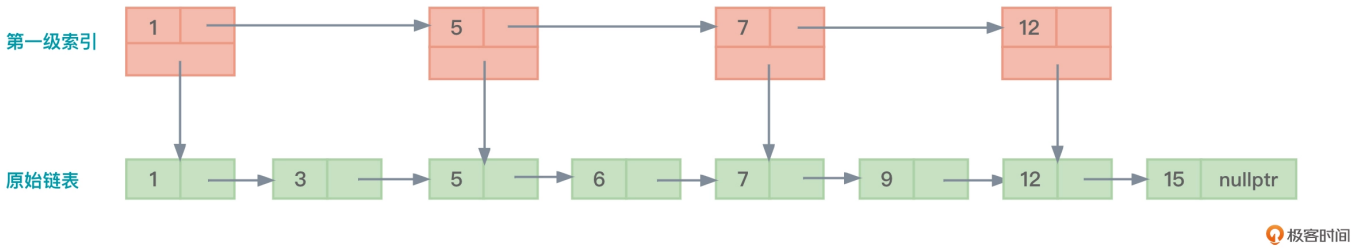

图2 增加了第一级索引的单链表

在图 2 中，最下面是原始的有序链表，在原始有序链表上面增加一层索引——第一级索引。可以看到，第一级索引中的数据当然同时也在原始链表中，但第一级索引中的数据分布要比原始链表中的数据稀疏——比如可以是原始链表中数据数量的一半。

我们把增加了索引的原始有序链表称为跳表，跳表基于原始的有序链表，是在原始的有序链表之上增加了一级或者多级索引而构成。

增加了第一级索引后，查询元素就不再是从原始链表开始查，而是从第一级索引开始查。比如查找元素 7，先找第一级索引看有没有 7 存在，找到了，说明元素 7 在原始链表中存在。再比如查找元素 9，还是先找第一级索引，看看具体步骤：

9 > 1，所以继续在第一级索引中向后找。

9 > 5，所以继续在第一级索引中向后找。

9 > 7，所以继续在第一级索引中向后找。

9 < 12，这就说明元素 9 是在 7 和 12 这两个第一级索引元素之间，那么从第一级索引向下寻找，这样就寻找到了原始链表，因为第一级索引中的元素 7 正好指向原始链表中的元素 7，所以从原始链表中的元素 7 开始向后找，最终会找到元素 9。

试想一下，如果原始链表中的元素非常多，数以千万计，即便创建的第一级索引比原始链表的数据（或节点数）少了一半，但仍旧会有不少数据。所以，我们可以在第一级索引基础上再创建第二级索引、第三级索引甚至更多级索引。

每一级索引当然会比上一级索引更稀疏，比如是上一级元素 / 节点数量的一半，这意味着在该级索引中查找元素需要进行的对比更少。给图 2 增加了第二级索引后如图 3。

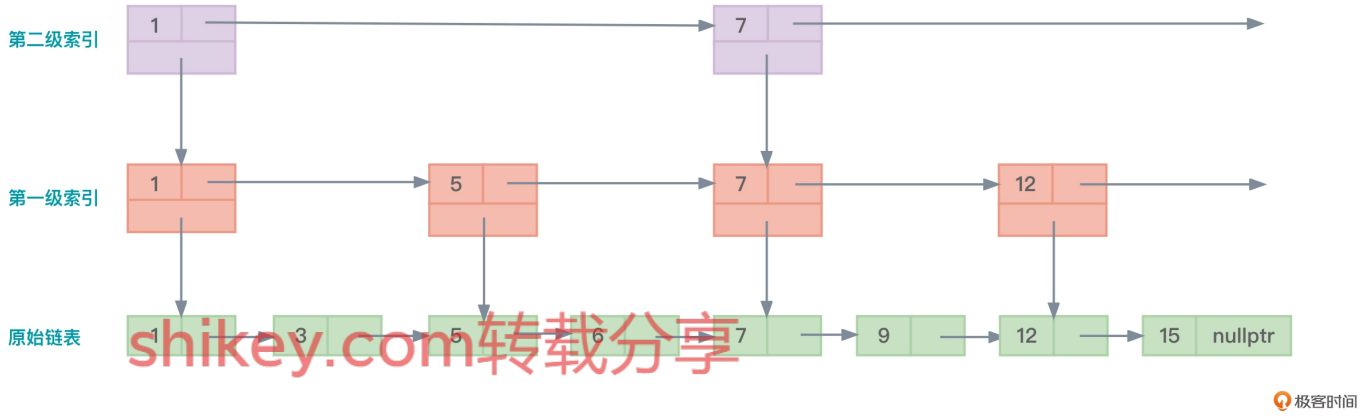

图3 增加了第二级索引的单链表

在图 3 中，查询元素当然就要从第二级（最高级）索引开始，第二级索引没有就向下辗转到第一级索引继续搜索，第一级索引没有就继续向下辗转到原始链表继续搜索。在第二级索引走

一步，相当于在第一级索引走二步，相当于在原始链表走四步。这样查询元素时移动的步伐就更大了，需要对比的元素数目就更少了。

比如，创建了第二级索引后，查找元素 9 就如下图 4 所示的路径了：

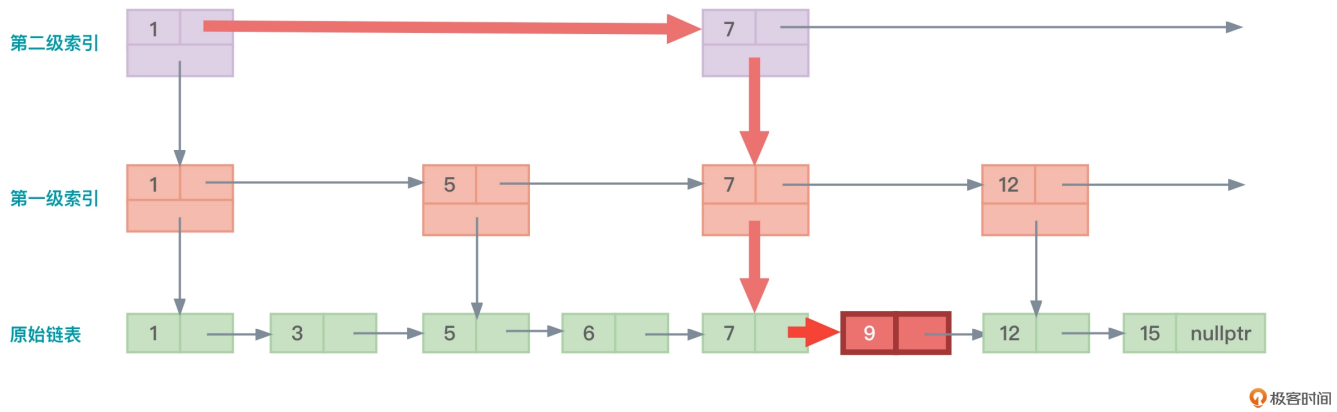

图4 增加了第二级索引后查找元素9所要经过的路径（加粗）

显然，原始链表中的元素越多，通过增加多级索引就可以大量减少查询元素的比较次数。

可以看到，如果原始链表中元素有 n 个，如果按每级索引节点数减半，也就是每两个节点变为更高级索引中的一个节点来计算，那么第一级索引的元素数量就是 n/2 个，第二级索引的元素数量就是 n/4 个，第 k 级索引的元素数量（节点数量）就是 n/(2^k)。

总结下来，如果索引一共有 h 级，而第 h 级（最高级）有 2 个节点，即 n/(2^h) = 2，那么 h =(log_2^n) -1。这表示查询的时间复杂度是 O(log_2^n)。请想一想，由从原始链表中查询数据所需要的 O(n) 时间复杂度变为 O(log_2^n)，这肯定是非常大的改进吧！

此外，因为原始链表中的数据可能会不断地增加和删除，这就会导致各级索引中的数据也要进行相应的增加和删除，所以各级索引中数据的分布也肯定会是不均匀的。这样，对于**增加和删除元素的操作，跳表的时间复杂度会变为 O(log_2^n)**。因为增加数据需要寻找插入位置以保证数据有序，删除数据需要在各个层级寻找要删除的元素。

说完时间复杂度，再看一看跳表的空间复杂度。

我们设原始链表的大小为 n，如果按每级索引节点数减半，也就是每两个节点变为更高级索引中的一个节点，最高级索引有两个节点来计算，则每级索引的节点数应该是 n/2,n/4,n/8,……,8,4,2，这其实是一个等比数列。每级索引的节点总和是 n/2+n/4+n/8+……+8+4+2=n-2。

可以看得出来，元素数为 n 的跳表中，原始链表有 n 个元素，大概还需要额外接近 n 个元素的存储空间来保存各级索引中的节点。如果按每三个节点变为更高级索引中的一个节点，最高级索引有一个节点来计算，则每级索引的节点数应该是 n/3,n/9,n/27,……,9,3,1，这也是个等比数列。每级索引的节点总和是 n/3+n/9+n/27+……+9+3+1≈n/2，相比于每两个节点变为一个节点，少了大概一半的索引节点存储空间。

无论是 n/2+n/4+n/8+……+8+4+2，还是 n/3+n/9+n/27+……+9+3+1，其实空间复杂度最终还是 O(n)，但当然还是比原始链表所占用的空间更多。所以**跳表也体现了空间换时间来提高效率的开发思想。**

不过，在实际开发中，我们一般都不用计较索引中节点所占用的空间。因为原始链表中存储的不一定是整数，可能是较大的对象，而相关的索引节点并不需要保存对象，只需要保存关键字和必须的指针，对象当然一般都会比索引中的节点大很多，所以索引中的节点所占的额外的空间更是可以忽略不计了。

在跳表中插入数据

在跳表中插入数据需要先向跳表的原始链表中插入数据（从下到上插入）。当向原始链表中插入数据时，索引也要动态更新。那么如何进行动态更新呢?

可以采用一个概率算法——每一层节点被提取到上一层的概率是 1/2（50%），即:

原始链表节点被提取到第一级索引的概率是 1/2。

原始链表节点被提取到第二级索引的概率是 1/4。

原始链表节点被提取到第三级索引的概率是 1/8。

那么为什么要进行动态更新呢？因为原始链表中的节点多了，索引节点也应该相应地增加一些，以避免因为时间复杂度的退化导致插入、删除、查找效率变低。

OK，我们看一看具体操作。现在，针对一个开始为空的跳表，希望将这么一组乱序数据插入到跳表中：3、1、7、6、9、5、15、12。

在跳表最底层的原始链表中，准备采用带头节点的单链表。所以刚开始的跳表应该仅仅只有一个头节点，用一个被称为“头节点指针”的指针来指向该头节点。如图 5 所示：

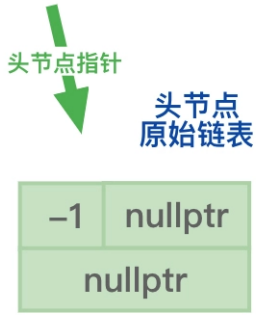

图5 只包含头结点的跳表（相当于空跳表或者可以认为目前只是个空链表）

当分别插入数据 3、1、7、6 时，跳表就会如下图 6 的子图 1 到子图 4 所示：

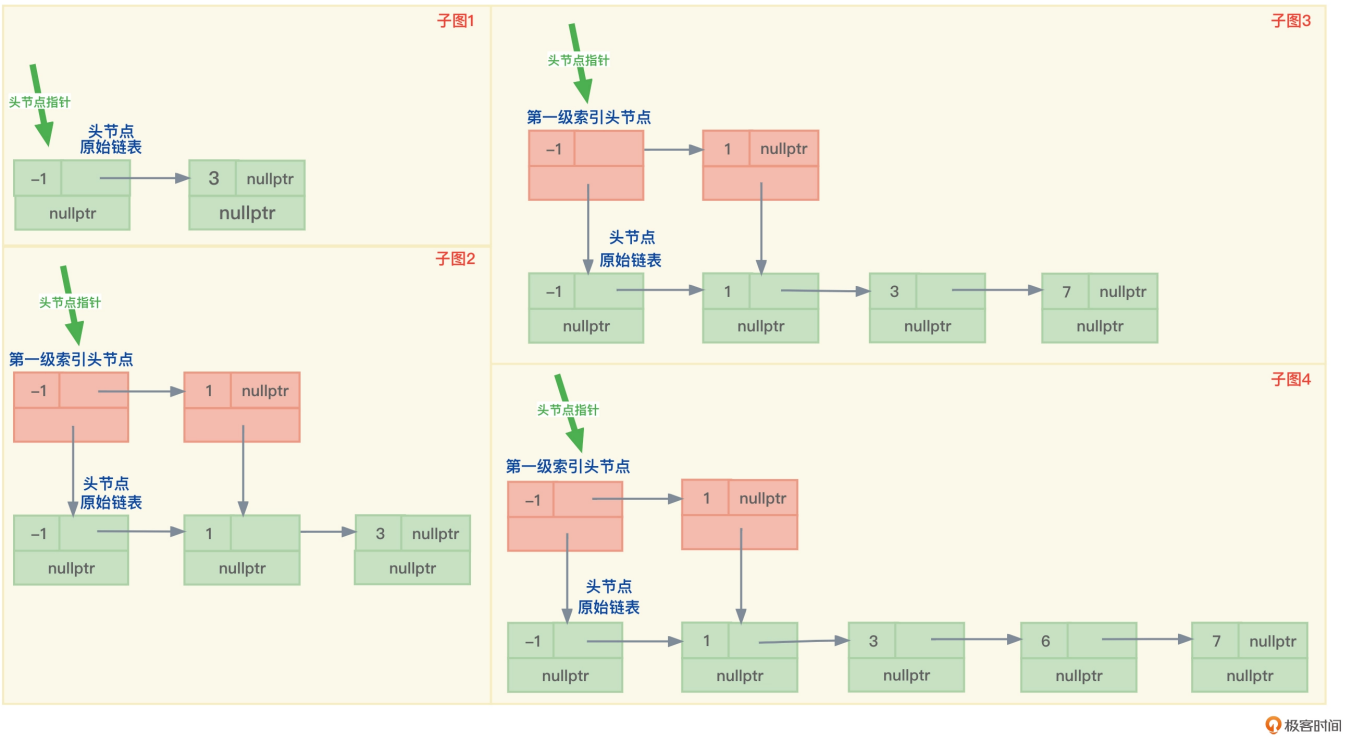

图6 插入了节点3、1、7、6后的跳表表现

图片信息量很大，我们一点一点拆解。观察图 6：

插入节点 3 时，原始链表头节点的右指针直接指向节点 3，如子图 1 所示。

继续插入节点 1，节点 1 自然排在节点 3 前面（左侧），但因为有一定概率节点被提到上一级索引，此时概率发生了，因此创建第一级索引。该级索引也需要创建个头节点，被称为“第一级索引头节点”。接着创建一个第一级索引中的节点 1，第一级索引头节点的向右指针指向这个新建立的节点 1，新建立的节点 1 的下指针指向原始链表的节点 1。第一级索引头节点的下指针指向原始链表头节点。而原来的**头节点指针**指向了新建立的第一级索引头节点，如子图 2 所示。

接着插入节点 7，自然排在原始链表节点 3 后面，由于概率没发生，因此该节点没有被提到上一级，如子图 3 所示。

接着插入节点 6，自然排在原始链表节点 3 后面，节点 7 前面，由于概率没发生，因此该节点没有被提到上一级，如子图 4 所示。

接着看图 7：

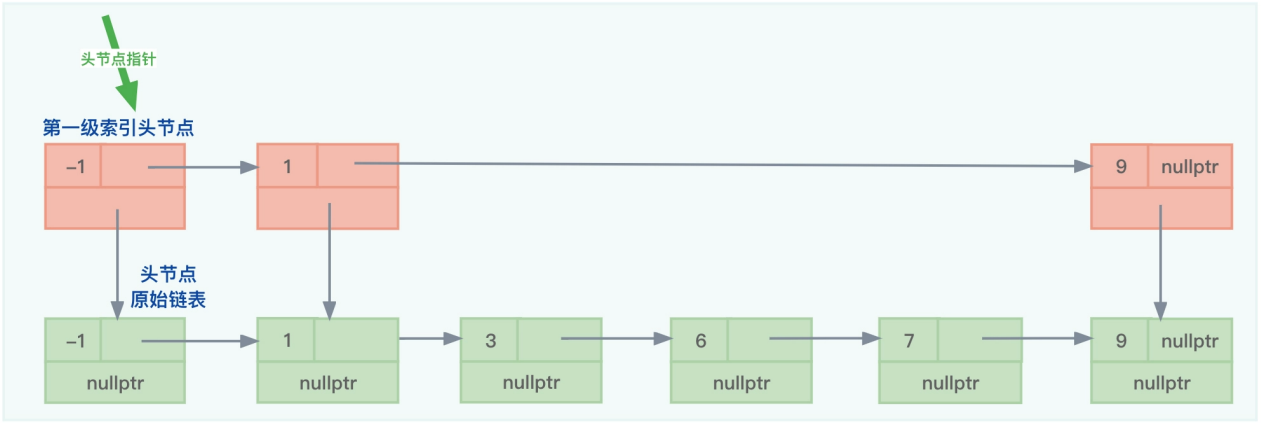

图7 继续插入节点9后的跳表表现

观察这张图片，接着插入节点 9，自然排在原始链表节点 7 后面（右侧）。此时概率发生了，节点 9 被提取到第一级索引中。因此在第一级索引中创建节点 9，新建立的第一级索引中的节点 9 的下指针指向原始链表中的节点 9，第一级索引中的节点 1 的右指针指向第一级索引中的节点 9。

接着看下面的图 8：

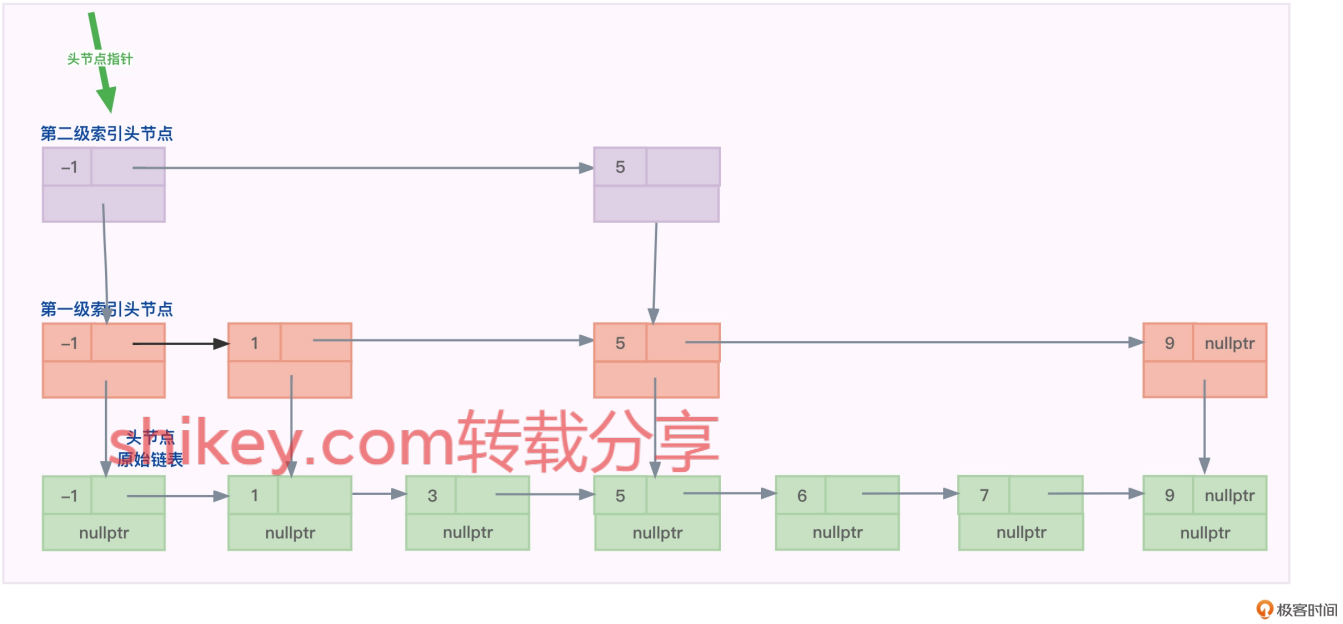

图8 继续插入节点5后的跳表表现

如上图。接着插入节点 5，自然排在原始链表节点 3 之后，节点 6 之前。

此时概率发生了，节点 5 被提取到第一级索引中。因此在第一级索引中创建节点 5，新建立的第一级索引中的节点 5 的向右指针指向同级的节点 9，下指针指向原始链表的节点 5，第一级索引中的节点 1 的向右指针指向新建立的同级节点 5。

因为节点 5 被提取到了当前已经存在的最高一级索引中，因此有概率创建一级新索引。此时概率果然发生了，结果因此创建第二级索引（该级索引也和第一级索引一样需要创建个头节点，被称为“第二级索引头节点”），接着创建一个第二级索引中的节点 5，第二级索引头节点的向右指针指向这个新建立的节点 5，新建立的节点 5 的下指针指向第一级索引的节点 5。第二级索引头节点的下指针指向第一级索引头节点。而原来的头节点指针指向了新建立的第二级索引头节点。

重复上述步骤，就可以向跳表中继续插入数据，你可以尝试完善这次的操作，这里我就不再赘述了。

从跳表中删除数据

另一方面，从跳表中删除节点并不复杂，只需要在各个层级，包括原始链表，也包括各级索引都将该节点删除即可。我们把要删除的节点称为节点 A，如果用通用一点的语言可以像下面这样描述删除节点。

从上到下在每级索引中找节点 A，找到则删除，找不到就找比 A 小但与 A 最接近的节点。这里不要忘记跳表是排好序的，这个节点其实是 A 的前趋节点。

沿着比 A 小的这个节点向下找到下一级索引中数值相同的节点，在这级中继续找节点 A，找到则删除，找不到则继续找比 A 小但与 A 最接近的节点。

沿着比 A 小的这个节点继续向下，如此反复……
这里以前面的图 8 为例，假如要删除节点 5，该怎么做呢?

从最上面的索引即第二级索引头节点开始向右查找，发现节点 5 在该级索引中，因此删掉该节点并设置相应的指针指向。删除该节点之前，一定要注意保存该节点左侧的节点，这样才能方便向下遍历和设置删除节点后的指针指向。注意，节点 5 左侧的节点其实就是第二级索引头节点。虽然此时第二级索引只剩“第二级索引头节点”，但并没有关系，不用做额外处理。

沿着第二级索引头节点向下找，找到第一级索引头节点，沿着该头节点向右找，找到了该级索引中的节点 5，删除掉该节点并设置相应的指针指向。这里注意，删除该节点之前要保存该节点左侧的节点以方便向下遍历和设置删除节点后的指针指向，而节点 5 左侧的节点其实就是节点 1。

继续沿着第一级索引中的节点 1，这个我们之前保存了。找到了原始链表中的节点 1，沿着该节点向右找，找到了其中的节点 5，删除掉该节点并设置相应的指针指向。

删除节点 5 后的结果如图 9 所示：

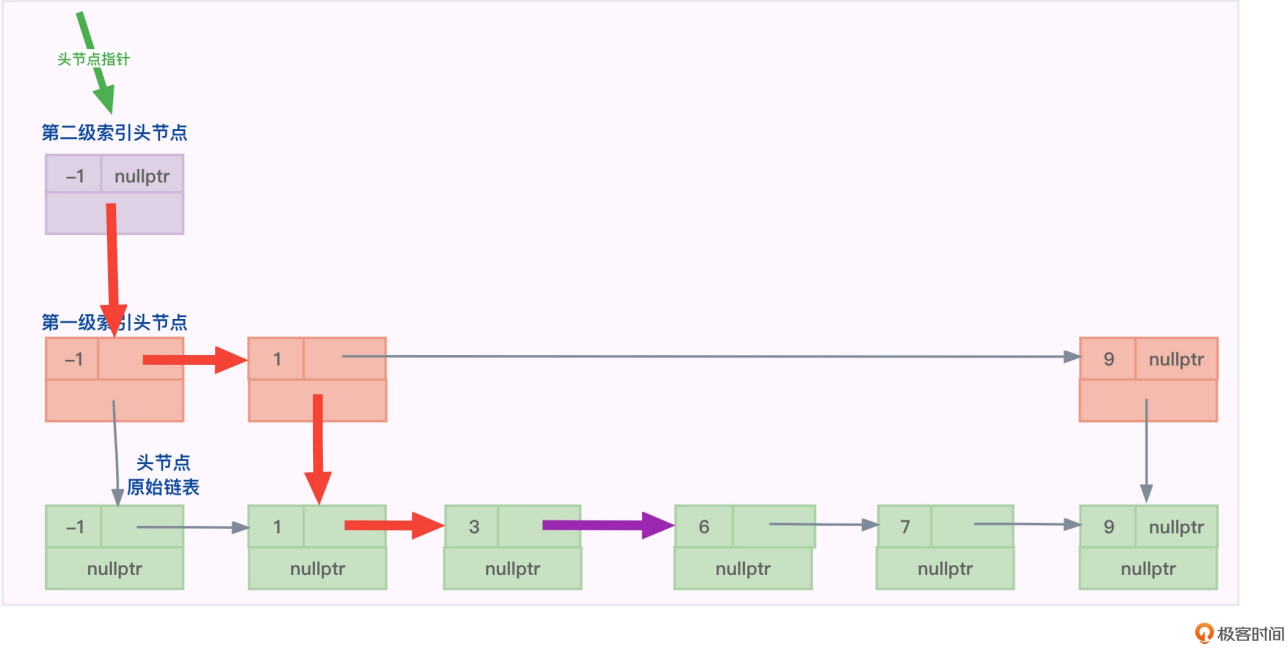

图9 删除节点5后的跳表外观（注意粗线代表寻找节点的路线）

不过，一个例子可能不太好理解，我们这次还是以前面图 8 为例，假如要删除的不是节点 5，而是节点 7：

从最上面的索引即第二级索引头节点开始向右查找，因为 5<7 而节点 5 之后再没节点了，所以找到了节点 5。

沿着第二级索引中的节点 5 向下找，找到了第一级索引中的节点 5。沿着第一级索引中的节点 5 向右找，发现是节点 9，因为 7<9，这说明第一级索引中没有节点 7。

沿着第一级索引中的节点 5 向下找，找到了原始链表中的节点 5，沿着该节点向右找，找到了其中的节点 7，删除掉该节点并设置相应的指针指向。

删除节点 7 后的结果如图 10 所示：

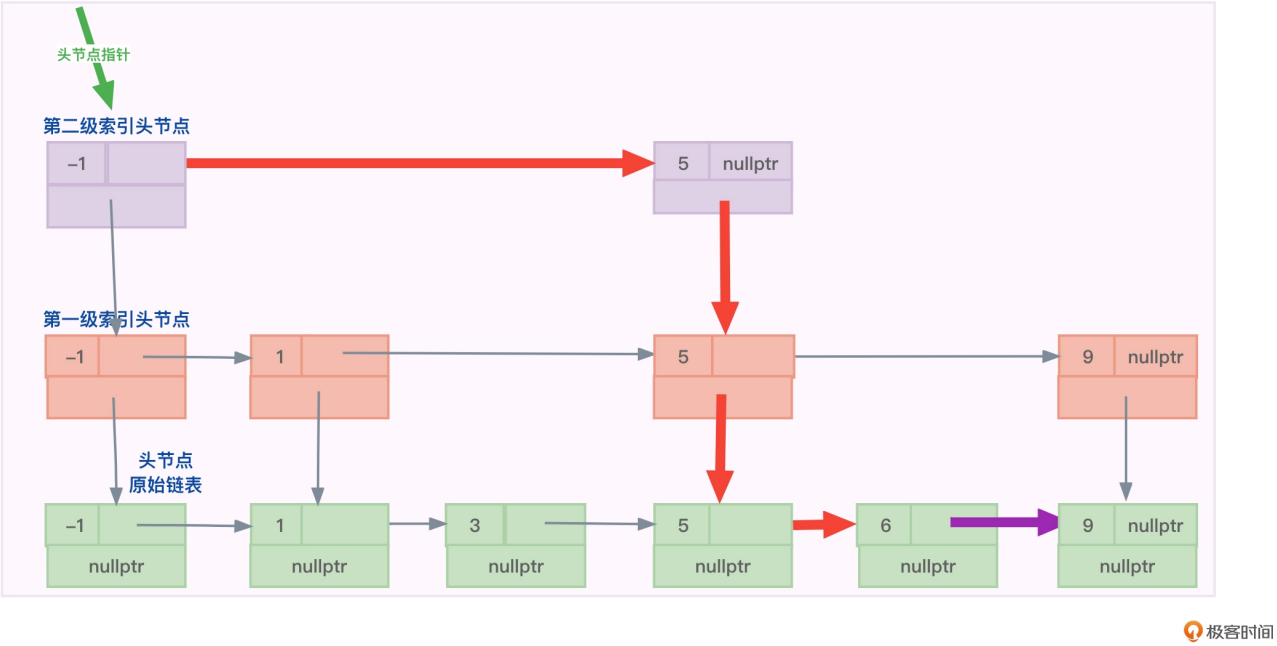

图10 删除节点7后的跳表外观（注意粗线代表寻找节点的路线）

当然，你也可以尝试自己选择一个删除的节点，想一想应该经历哪些步骤。如果有哪里不理解，也欢迎你随时和我交流。

实现代码

所有的思路理解透彻之后，你就可以尝试理解一下跳表的实现源码了。

复制代码

```
#include "MersenneTwister.h" //从外面找的一个文件，用来产生更随机的数字

//每个节点的定义
template <typename T>
struct Node
{
  T  data;
  Node* right;  //向右指向的指针
  Node* down;   //向下指向的指针
};

#define MAX_LEVEL 100
MTRand g_ac; //全局量以产生更加随机的随机数，#include "MersenneTwister.h"

```

```cpp
//跳表定义
template <typename T>
class SkipList
{
public:
  SkipList()  //构造函数
  {
    pathList.reserve(MAX_LEVEL); //保留空间提升性能
    m_head = new Node<T>; //先创建一个头结点
    m_head->right = nullptr;
    m_head->down = nullptr;
    m_head->data = -1; //可以随便给个值，方便调试时观察和区分
  }
  ~SkipList() //析构函数
  {
    while (m_head != nullptr)
    {
      Node<T>* pnode = m_head->right;
      Node<T>* ptmp;
      while (pnode != nullptr) //该循环负责释放数据节点
      {
        ptmp = pnode;
        pnode = pnode->right;
        delete ptmp;
      }

      Node<T>* tmphead = m_head;
      m_head = m_head->down;
      delete tmphead;
      tmphead = nullptr;
    }
    return;
  }

  //插入数据
  template <typename T>
  void Insert(const T& e)
  {
    pathList.clear(); //记得每次清理，以免使用上次遗留的旧数据
    Node<T>* p = m_head;

    //先往右找再往下找(跨过多级索引，找到原始链表)，目的就是找到原始链表中的插入位置
    while (p)
    {
      //向右找合适的节点位置
      while (p->right != nullptr && p->right->data < e)
      {
        p = p->right; //先往右走
      }
```

```
64
65        pathList.push_back(p);  //尾部插入元素，最终原始链表在最尾部
66        p = p->down; //再往下走
67      } //end while
68
69      bool insertupsign = true; //向上层插入节点的概率，刚开始是原始链表，肯定是要将数据插入其
70      Node<T>* downNode = nullptr; //创建一个新节点时，新节点的down指针指向的就是这个downN
71
72      //下面这个while能够从最底下往上回溯处理
73      while (insertupsign == true && pathList.size() > 0)
74      {
75        //insert指向插入位置的节点
76        Node<T>* insert = pathList.back(); //返回最底部的数据—第一次返回的是原始链表
77        pathList.pop_back(); //干掉最底部的数据
78
79        //创建新节点
80        Node<T>* newnode = new Node<T>();
81        newnode->data = e;
82        newnode->right = insert->right; //原来insert节点指向的元素现在让newnode指向
83        newnode->down = downNode;
84
85        insert->right = newnode; //insert指向这个新节点
86        downNode = newnode;
87
88        //每一层节点被提取到上一层的概率是1/2 (50%)
89        unsigned int tmpvalue = g_ac.randInt(); //产生随机数
90        if (tmpvalue % 2 == 0)
91        {
92          insertupsign = true;
93        }
94        else
95        {
96          insertupsign = false;
97        }
98      } //end while
99
100     //标志要新加一级索引（新加一级索引意味着肯定要在这个索引中加入一个节点）
101     if (pathList.size() <= 0 && g_ac.randInt() % 2 == 0)  //后一个条件表示是否增加新一
102     {
103       //表示新节点被增加到了最上层索引并且增加新一级索引
104       Node<T>* newnode = new Node<T>();
105       newnode->right = nullptr;
106       newnode->down = downNode;
107       newnode->data = e;
108
109       //既然是新加一级索引，那么索引也要个头节点，所以如下是增加头节点
110       Node<T>* tmpheadnode = new Node<T>();
111       tmpheadnode->right = newnode;
112       tmpheadnode->down = m_head;
```

```
113       tmpheadnode->data = -1;
114       m_head = tmpheadnode; //m_head指向最上级索引的头节点
115     }
116     return;
117   }
118
119   //删除数据，返回true表示找到并删除了数据，返回false表示没找到数据
120   template <typename T>
121   bool Delete(const T& e)
122   {
123     Node<T>* p = m_head;
124     bool iffind = false; //是否找到
125     while (p)
126     {
127       //向右找合适的节点位置
128       while (p->right != nullptr && p->right->data < e)
129       {
130         p = p->right; //先往右走
131       }
132
133       //右边没节点了，或者右边的节点过大，则需要向下找
134       if (p->right == nullptr || p->right->data > e)
135       {
136         p = p->down; //向下
137       }
138       else
139       {
140         //找到了要删除的节点
141         Node<T>* pdelnode = p->right; //pdelnode是要删除的节点，而p是要删除的节点左侧的
142         iffind = true;
143         p->right = p->right->right; //要删除节点左侧的节点指向要删除节点右侧的节点
144         delete pdelnode; //释放要删除的节点的内存，注意删除位置放这里比较合适
145         p = p->down;
146       }
147     } //end while
148
149     //如果某级索引只有一个节点，删除该节点后，可能会导致该该级索引一个节点也不存在，只有索引头节
150     return iffind;
151   }
152
153   //查找数据
154   template <typename T>
155   bool Find(const T& e)
156   {
157     Node<T>* p = m_head;
158     int jumptimes = 0; //寻找进行的指针跳跃次数统计
159
160     while (p)
161     {
```

```
162       //向右找合适的节点位置
163       while (p->right != nullptr && p->right->data < e)
164       {
165         jumptimes++;
166         p = p->right; //先往右走
167       }
168
169       //右边没节点了，或者右边的节点过大，则需要向下找
170       if (p->right == nullptr || p->right->data > e)
171       {
172         jumptimes++;
173         p = p->down; //向下
174       }
175       else
176       {
177         //找到了
178         cout <<"Find成功找到元素:"<< e <<"共进行:"<< jumptimes <<"次跳跃。"<< endl;
179         return true;
180       }
181     } //end while
182     cout <<"Find寻找元素:"<< e <<"失败!"<< endl;
183     return false;
184   }
185
186   //输出跳表中的所有元素，从上到下，从左到右输出
187   void DispSkipList()
188   {
189     if (m_head->right == nullptr && m_head->down == nullptr)
190       return; //空表没啥可输出的
191
192     cout <<"--------------begin----------------"<< endl;
193     Node<T>* tmphead = m_head;
194     int level = 0; //统计下有多少级索引
195     while (true)
196     {
197       Node<T>* currp = tmphead->right;
198       while (currp != nullptr)
199       {
200         cout << currp->data <<" ";
201         currp = currp->right;
202       } //end while
203       cout << endl;
204       level++;
205       tmphead = tmphead->down; //下走
206       if (tmphead == nullptr)
207         break;
208     } //end while
209     cout <<"--------------end----------------"<< endl;
210     cout <<"共产生了:"<< level-1 <<"级索引。"<< endl;
```

```
    return;
  }
  //获取跳表的高度（有多少级索引），不计算原始链表高度
  int getlevel()
  {
    if (m_head->down == nullptr)
      return 0;

    Node<T>* p = m_head;
    int level = 0;
    while (p)
    {
      level++;
      p = p->down; //下走
    }
    return level - 1;
  }
private:
  Node<T>* m_head; //头指针，指向最上级索引的头节点
  vector<Node<T> *> pathList; //#include <vector>，记录这从索引到最底下的原始链表的向下
};
```

在 main 主函数中，加入如下测试代码来测试插入、删除、查找操作：

复制代码

```
SkipList<int> myskiplist;
myskiplist.Insert(3);
myskiplist.Insert(1);
myskiplist.Insert(7);
myskiplist.Insert(6);
myskiplist.Insert(9);
myskiplist.Insert(5);
myskiplist.Insert(15);
myskiplist.Insert(2);
myskiplist.DispSkipList();
myskiplist.Delete(1);
myskiplist.DispSkipList();
for (int i = 0; i <21000000; ++i) //搞大量数据进行插入和查找测试
{
  myskiplist.Insert(i);
}
myskiplist.Find(1);
cout <<"当前跳表索引高度："<< myskiplist.getlevel() << endl;
```

代码执行了大概 110 秒（如果在 Visual Studio 中选择编译和执行 Release 版则执行的速度会快很多），因为循环次数较大。代码每次的执行结果可能不同，某一次代码的执行结果如下：

```
----------begin----------
2
1 2 6
1 2 3 6
1 2 3 5 6 7 9 15
----------end----------
共产生了:3级索引。
----------begin----------
2
2 6
2 3 6
2 3 5 6 7 9 15
----------end----------
共产生了:3级索引。
Find成功找到元素:1共进行:22次跳跃。
当前跳表索引高度：24
```

当然，跳表的实现代码可以非常灵活。跳表中的原始链表也不一定非是普通链表，用双向链表实现也完全可以，只要能更好地实现需求，就是好代码。

在跳表中插入数据时，我这里的代码采用的是概率算法——每一层节点被提取到上一层的概率是 1/2（50%），来决定插入的数据是否向上一层索引提取。当然也可以采用其他算法，比如根据一定的条件产生一个随机数 k，把新插入的节点添加到第 1 到第 k 级索引中，这方面你有兴趣可以发挥自己的想象力来进一步丰富代码。

跳表、红黑树、B+ 树

面试中经常有这样的问题：MySQL 为什么用 B+ 树实现索引而不用跳表？Redis 为什么用跳表实现有序集合而不是用红黑树实现？

MySQL 为什么用 B+ 树实现索引而不用跳表

前面的课程，我们讲解过 MySQL 中的索引是采用 B+ 树实现的，实现方法和这里所讲述的跳表其实很类似——都是增加多级索引来实现。

我们可以客观分析一下 B+ 树和跳表在数据查询和写操作上的优劣。

B+ 树

B+ 树一般都是叶子满了且索引节点也满了才会增加 B+ 树的高度，MySQL 中的 B+ 树一般三层高就能够保存 2100 多万条记录。

对于数据查询，即便这些要查询的数据保存在磁盘中，那么也只需要进行 3 次磁盘 IO 操作就可以拿到数据。

对于写操作，也就是 B+ 树的插入操作，涉及 B+ 树节点的拆分、合并，相对比较复杂。

跳表

向跳表中增加数据时，跳表的高度是否增加是靠随机值——一个新数据插入到原始链表后，有 50% 的概率在第二层索引中加入该数据，有 25% 的概率在第三层索引中加入该数据，以此类推，直到最顶层索引。

对于数据查询，如果要在原始链表中存储 2100 多万条记录（2^{24}），跳表的高度大概要达到 24 层，这意味着查询一次数据，如果运气比较坏可能要进行 24 次磁盘 IO 操作才可以拿到数据。和 B+ 树比，跳表的查询效率显然慢很多。

对于写操作，跳表的插入操作和 B+ 树相比要简单很多，所以跳表的写入方面性能要优于 B+ 树。

MySQL 中最常用的 InnoDB 和 MyISAM 存储引擎的底层索引用的都是 B+ 树。其实，MySQL 的存储引擎是可以更换的。如果你有兴趣，完全可以自己设计一个存储引擎底层索引采用跳表来实现。估计写入数据的性能会不错，但查询数据的性能应该会比 InnoDB 和

MyISAM 引擎差不少。此外，跳表这种数据结构出现得比较晚也是在很多应用场景中没有采用跳表的主要原因之一。

Redis 为什么用跳表实现有序集合而不是用红黑树 /B+ 树 / 二叉树实现?

Redis 是内存数据库，因为不存在操作磁盘的问题，所以层的高度比较高并不是大问题。

Redis 有序集合支持的核心操作一般也就是数据的插入、删除、查询，当然这些动作用红黑树、B+ 都可以完成。

对于数据的插入、删除、查询等操作：

二叉树（包括 AVL 树、红黑树）需要对节点调整平衡（旋转、变色操作）。

B+ 树存在一系列节点的拆分、合并操作。

跳表插入和删除数据时不需要考虑平衡性，也不需要拆分合并等。

另外，对于按照某个区间查找数据这个操作，AVL 树、红黑树的效率没有跳表高。因为跳表能以 O(log_2^n) 的时间复杂度定位到区间的起点，然后在原始链表中顺序向后遍历即可。

此外，跳表的实现代码相对更简单，通过改变创建索引的算法，可以很有效地平衡代码的执行效率和内存的消耗问题。

所以，Redis 选择了跳表这种数据结构，因为实现简单，代码易读易懂，开销小（效率高）。

小结

本节我带你学习了跳表这种数据结构，跳表本身基于链表，是对链表的优化。在原始的有序链表之上增加一级或者多级索引而构成。

跳表中的元素必须有序。跳表实现相对简单，效率更高，在一些流行项目比如 Redis、LevelDB 中常被用来代替平衡二叉树（AVL 树）或二分查找。其插入、删除、搜索的时间复杂度都是 O(log_2^n)。

文中我详细向你阐述了通过跳表查找、插入、删除数据元素所要经过的步骤并给出了详尽的实现代码。

最后，我向你详细解释了两个面试的经典问题。MySQL 为什么用 B+ 树实现索引而不用跳表？

一句话总结，就是为了有效减少磁盘操作次数，从而提升数据查询效率。另一方面，Redis 为什么使用跳表来实现有序集合呢？是因为实现简单，代码易读易懂，效率高。

思考题

给定一个有序链表，尝试采用画图的方式将其转换为一个跳表。

欢迎你在留言区和我互动。如果觉得有所收获，也可以把课程分享给更多的朋友一起学习。我们下节课见！

精选留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。