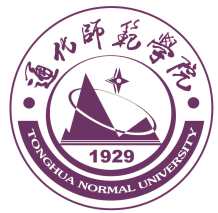
算法分析与设计

第四讲跳跃表

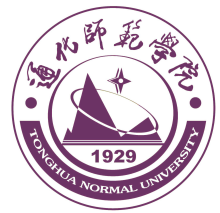
纪洪波

通化师范学院　计算机学院

2016 年 9 月 4 日



目录

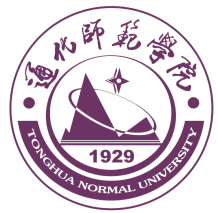


顺序查找

5	4	7	2	8	1	3	6

对于上面的情况无论是查找 3 还是查找 8，只有一种办法，那就是遍历。时间复杂度是 O(N)

但是举一个生活中的例子，我们在查新华字典的时候，没有人会一页一页的翻吧，肯定是翻到中间看看，然后再决定往左边查，还是往右边查。



二分查找 I

显然，可以把元素排序放在一个数组中，这样就可以利用二分查找了。查字典也算是二分查找的一个实际例子。二分查找的时间复杂度是 $\mathrm{O}(lb\,\mathrm{N})$

1	2	3	4	5	6	7	8

利用二分查找是有前提限制的：大家思考：啥限制？

灵机一动！

1：元素已排序。
2：元素可以随机访问。

二分查找 I

显然，可以把元素排序放在一个数组中，这样就可以利用二分查找了。查字典也算是二分查找的一个实际例子。二分查找的时间复杂度是 $O(lb N)$

1	2	3	4	5	6	7	8

利用二分查找是有前提限制的：

灵机一动！

1:元素已排序。
2:元素可以随机访问。

二分查找 II

二分查找要求元素可以随机访问，这就注定了需要把元素存储在连续内存中。这样查找确实很快，但是插入和删除元素的时候，为了保证元素的有序性，就需要大量的移动元素了。（当然如果删除操作较少的话，也可以用懒惰删除，被删除的元素做个标记，但并不实际从内存中清除，所以也不用移动元素。但是对于插入还是没有办法）

灵机一动！

综上所述，其实我们需要的是一个既能够进行二分查找，又能快速添加和删除元素的数据结构。这就非二叉查找树莫属了。

现在利用树这个模型，对 1,2,3,4,5,6 这 6 个元素构造一个二叉排序树，如下：

二分查找 II

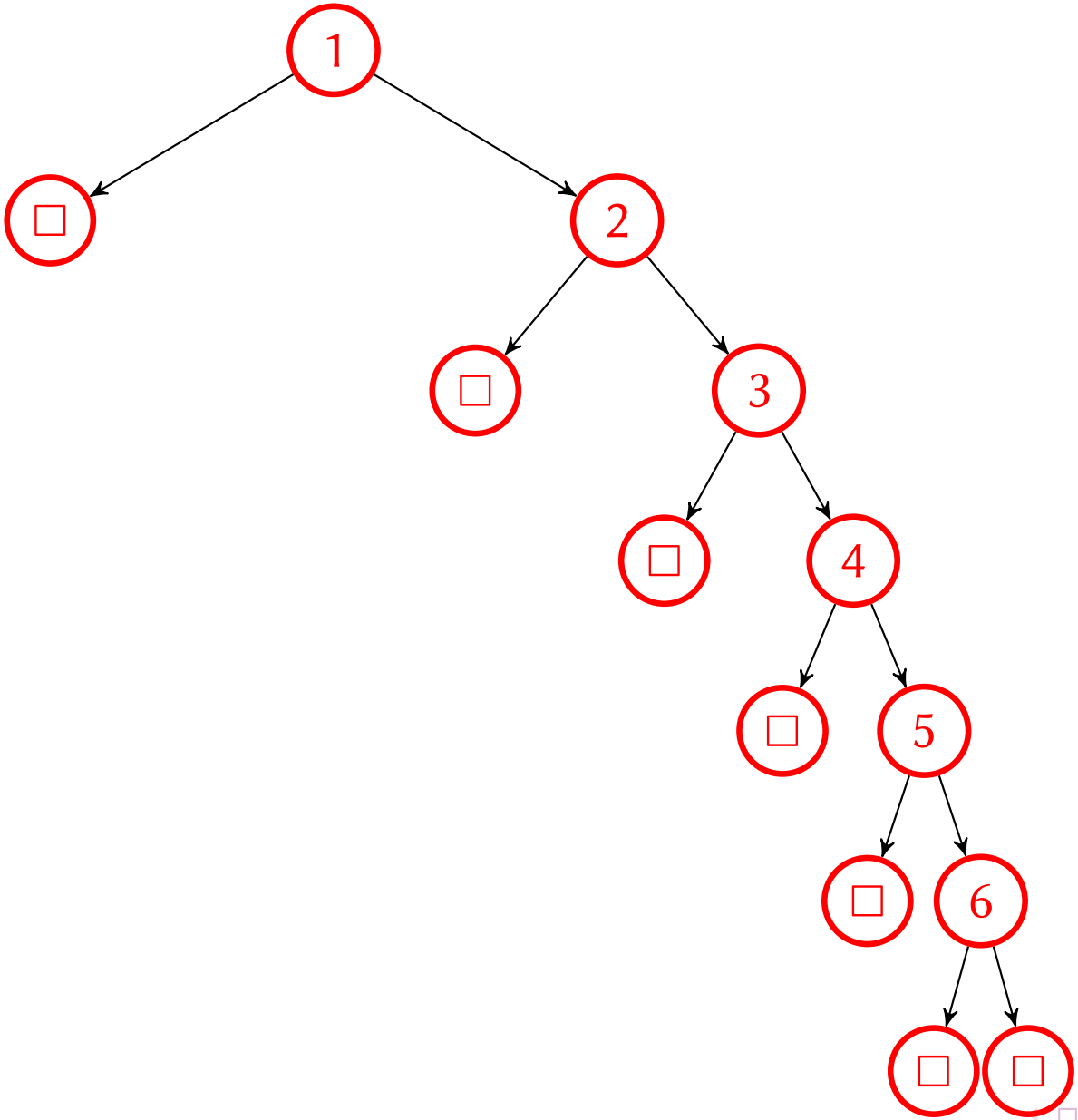
二分查找要求元素可以随机访问，这就注定了需要把元素存储在连续内存中。这样查找确实很快，但是插入和删除元素的时候，为了保证元素的有序性，就需要大量的移动元素了。（当然如果删除操作较少的话，也可以用懒惰删除，被删除的元素做个标记，但并不实际从内存中清除，所以也不用移动元素。但是对于插入还是没有办法）

灵机一动！

综上所述，其实我们需要的是一个既能够进行二分查找，又能快速添加和删除元素的数据结构。这就非二叉查找树莫属了。

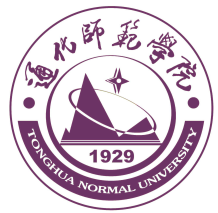
现在利用树这个模型，对 1,2,3,4,5,6 这 6 个元素构造一个二叉排序树，如下：

二叉查找树 III



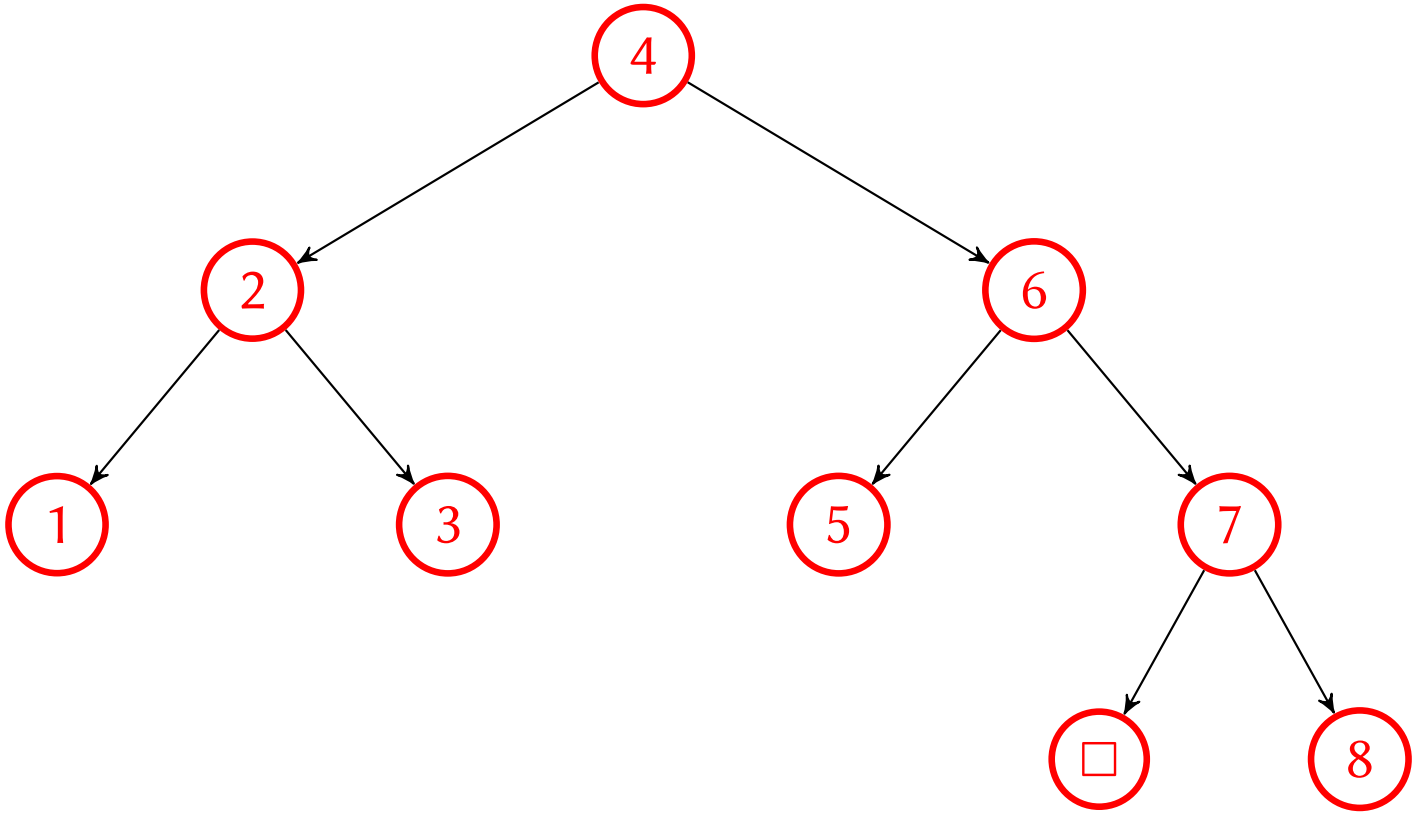
二分查找 IV

显然，上面这棵树太坑爹了。把自己的头右转 45° 或者把显示器左转 45°，我们会发现，这其实就是一个链表。这棵树不符合快速查找元素的要求，原因就是，每个节点的“左右子树之差 > 1”，也就是不平衡。不平衡就导致了树的高度过高，所以没有办法根据一个节点筛选掉一半的子结点 (左右子结点个数不相等)



二分查找 V

然后，带有高度平衡条件的 AVL 树，是下面这样的。



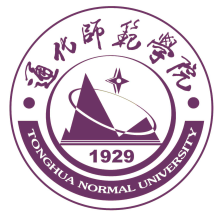
二分查找 VI

对于这样一棵树，查找元素就是变相的二分查找。（插入，删除也有了保证，直觉上只需修改几个指针）

但是，由于 AVL 树要求高度平衡，不论是插入还是查找，被操作节点到根节点的整条路径上的所有节点都要判断，是否打破了平衡条件，从而 LL, LR, RR, RL 旋转（我感觉整个世界都在旋转了！）。（当然了，实现一颗 AVL 树也不困难，但是相对于 SkipList 还是复杂很多）

RB 树是一种对平衡要求相对略低的 AVL 树。(类似于 2-3 树)

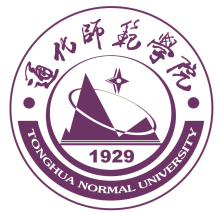
但是，无论是高度平衡的 AVL 还是 RB 树，自己编写起来难度都比较大。虽然说已经有了现成的 set/map(基于红黑树实现)。C++ 中的 STL 标准模板库就是基于 RB 树实现的。



问题

同学们思考一下：

二分查找和 AVL 树为什么那么高效?

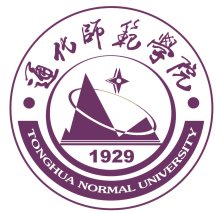


解答

原因就是:每比较一次,就能筛掉一半元素。

就像我们查字典一样,没有人会傻傻的从第一页开始一页一页的查,肯定是先翻到字典中间看看,然后决定继续查字典的前半部分还是后半部分。

接下来,我们就来看看跳跃表。



简介

之所以称为跳跃表，就是因为每个节点会额外保存几个指针，间隔的指向后继的节点。

跳跃表是一种随机化的数据结构，目前开源软件 Redis 和 LevelDB 都有用到它，它的效率和红黑树以及 AVL 树不相上下，但跳表的原理相当简单，只要你能熟练操作链表，就能轻松实现一个 SkipList。

跳跃表是 1987 年诞生的一种全新的数据结构，是一个链表的集合。它在进行查找、插入、删除等操作时的时间复杂度均为 $O(lb\ n)$，有着代替平衡树的本领，却有非常低的编程复杂度。

课后任务！

上网查看 Redis 和 LevelDB 都是什么？

简介

之所以称为跳跃表，就是因为每个节点会额外保存几个指针，间隔的指向后继的节点。

跳跃表是一种随机化的数据结构，目前开源软件 Redis 和 LevelDB 都有用到它，它的效率和红黑树以及 AVL 树不相上下，但跳表的原理相当简单，只要你能熟练操作链表，就能轻松实现一个 SkipList。

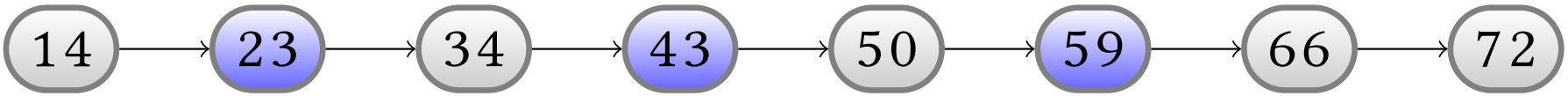
跳跃表是 1987 年诞生的一种全新的数据结构，是一个链表的集合。它在进行查找、插入、删除等操作时的时间复杂度均为 $O(lb\ n)$，有着代替平衡树的本领，却有非常低的编程复杂度。

课后任务！

上网查看 Redis 和 LevelDB 都是什么？

一个顺序表的简单搜索

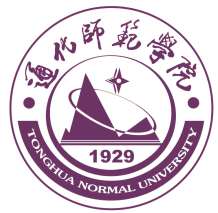
考虑一个有序表：



从该有序表中搜索元素 $< 23, 43, 59 >$，需要比较的次数分别为：

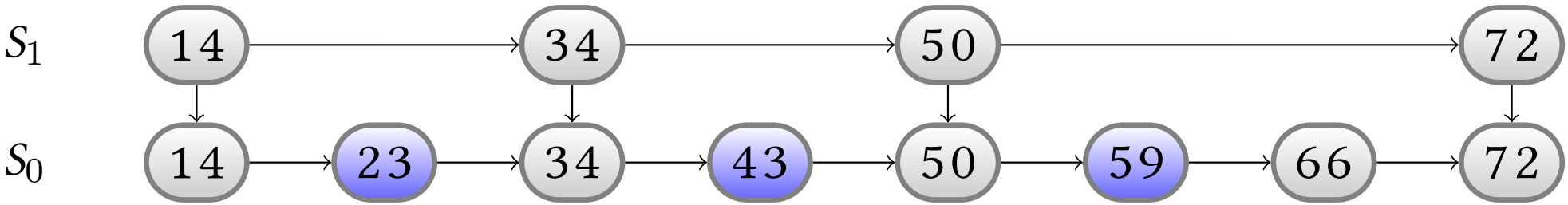
$2 + 4 + 6 = 12$ 次

有没有优化的算法吗? 链表是有序的，但不能使用二分查找。



第一次改进

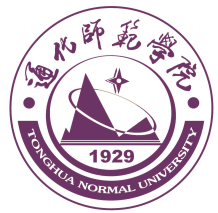
似二叉搜索树，我们把一些节点提取出来，作为索引。得到如下结构：



这里我们把 $< 14, 34, 50, 72 >$ 提取出来作为一级索引，这样搜索的时候就可以减少比较次数了。还能不能更快？找到问题，提出梦想！

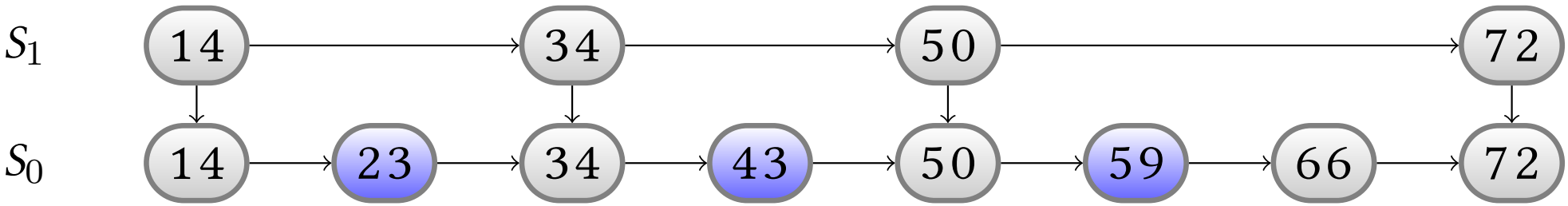
灵机一动！

查 59 需要经过 34，比较慢。



第一次改进

似二叉搜索树，我们把一些节点提取出来，作为索引。得到如下结构：



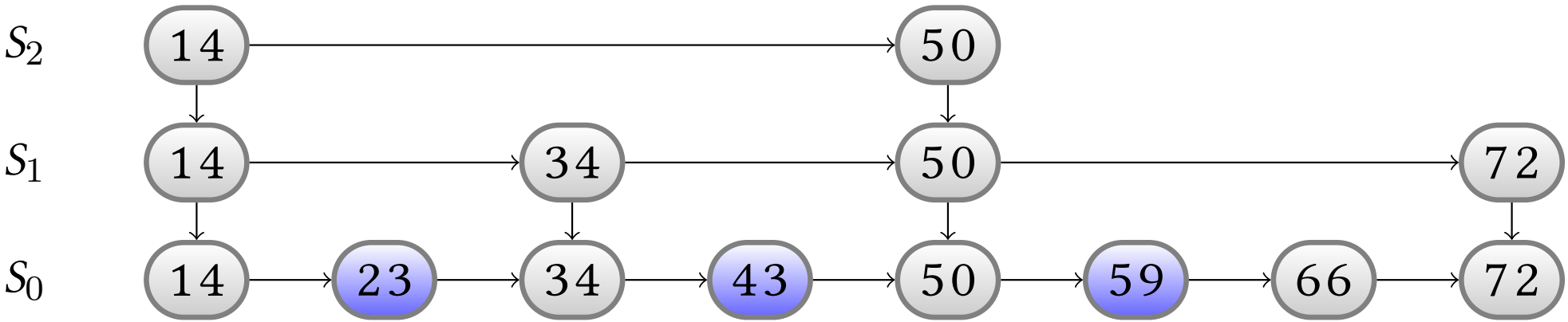
这里我们把 $< 14, 34, 50, 72 >$ 提取出来作为一级索引，这样搜索的时候就可以减少比较次数了。

灵机一动！

查 59 需要经过 34，比较慢。

再一次改进

还可以再从一级索引提取一些元素出来，作为二级索引，变成如下结构：



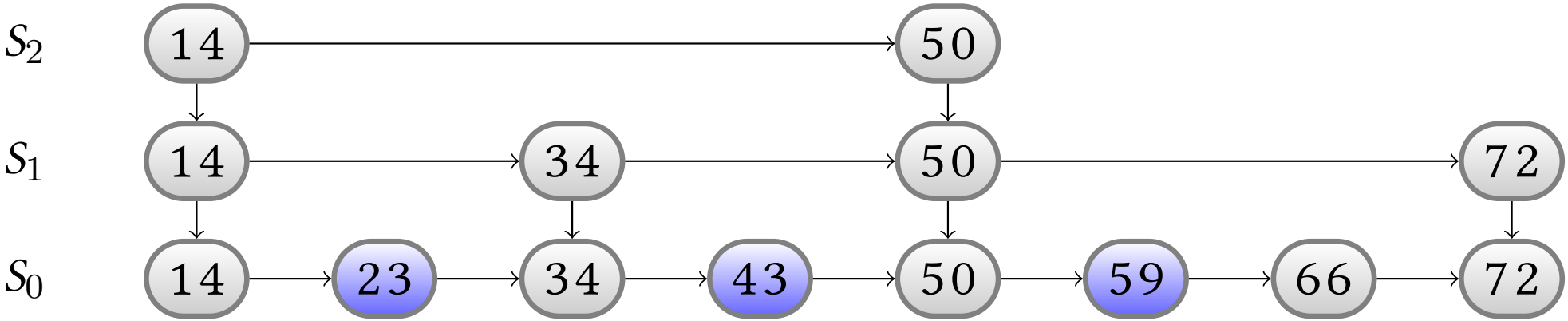
这里元素不多，体现不出优势，如果元素足够多，这种索引结构就能体现出优势来了。

灵机一动！

查 59 需要不经过 34，快一点儿！

再一次改进

还可以再从一级索引提取一些元素出来，作为二级索引，变成如下结构：



这里元素不多，体现不出优势，如果元素足够多，这种索引结构就能体现出优势来了。

灵机一动！

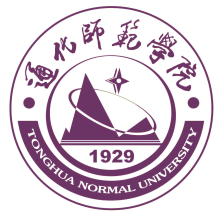
查 59 需要不经过 34，快一点儿！

跳跃表的性质

跳跃表由多条链构成（$S_0, S_1, S_2, \cdots, S_n$），且满足如下 3 个条件：

1、每条链必须包含两个特殊元素：∞ 和 $+\infty$。
2、S_0 包含所有的元素，并且所有链中的元素按照升序排列。
3、每条链中的元素集合必须包含序数较小的链的元素集合，即

$$S_0 \supseteq S_1 \supseteq S_2 \supseteq \cdots \supseteq S_h$$



跳跃表的性质

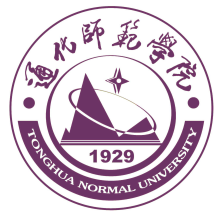
跳跃表由多条链构成（$S_0, S_1, S_2, \cdots, S_n$），且满足如下 3 个条件：

1、每条链必须包含两个特殊元素：∞ 和 $+\infty$。

2、S_0 包含所有的元素，并且所有链中的元素按照升序排列。

3、每条链中的元素集合必须包含序数较小的链的元素集合，即

$$S_0 \supseteq S_1 \supseteq S_2 \supseteq \cdots \supseteq S_h$$

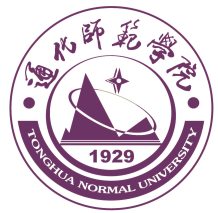


跳跃表的性质

跳跃表由多条链构成（$S_0, S_1, S_2, \cdots, S_n$），且满足如下 3 个条件：

1、每条链必须包含两个特殊元素：∞ 和 $+\infty$。
2、S_0 包含所有的元素，并且所有链中的元素按照升序排列。
3、每条链中的元素集合必须包含序数较小的链的元素集合，即

$$S_0 \supseteq S_1 \supseteq S_2 \supseteq \cdots \supseteq S_h$$

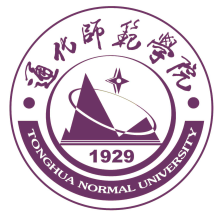


跳跃表的性质

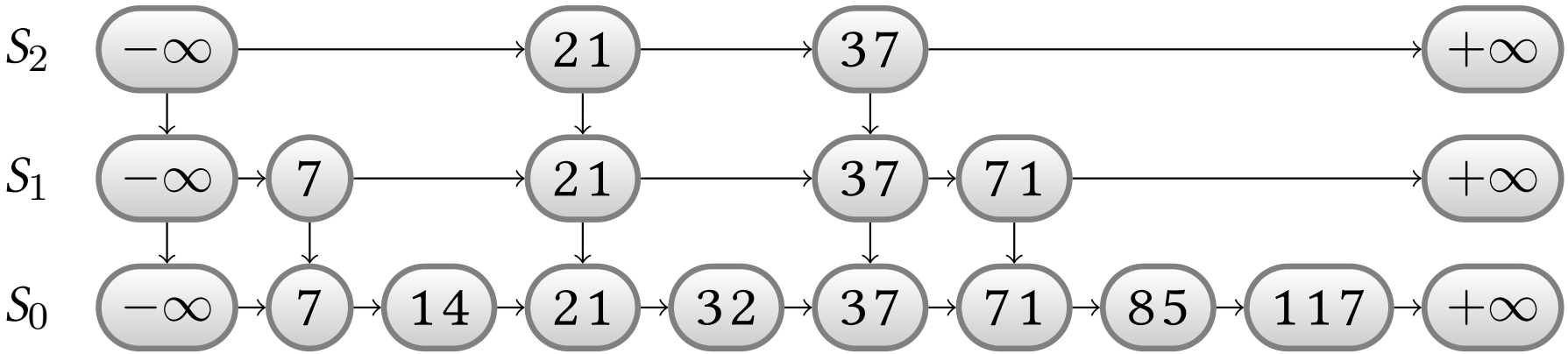
跳跃表由多条链构成（$S_0, S_1, S_2, \cdots, S_n$），且满足如下 3 个条件：

1、每条链必须包含两个特殊元素：∞ 和 $+\infty$。
2、S_0 包含所有的元素，并且所有链中的元素按照升序排列。
3、每条链中的元素集合必须包含序数较小的链的元素集合，即

$$S_0 \supseteq S_1 \supseteq S_2 \supseteq \cdots \supseteq S_h$$

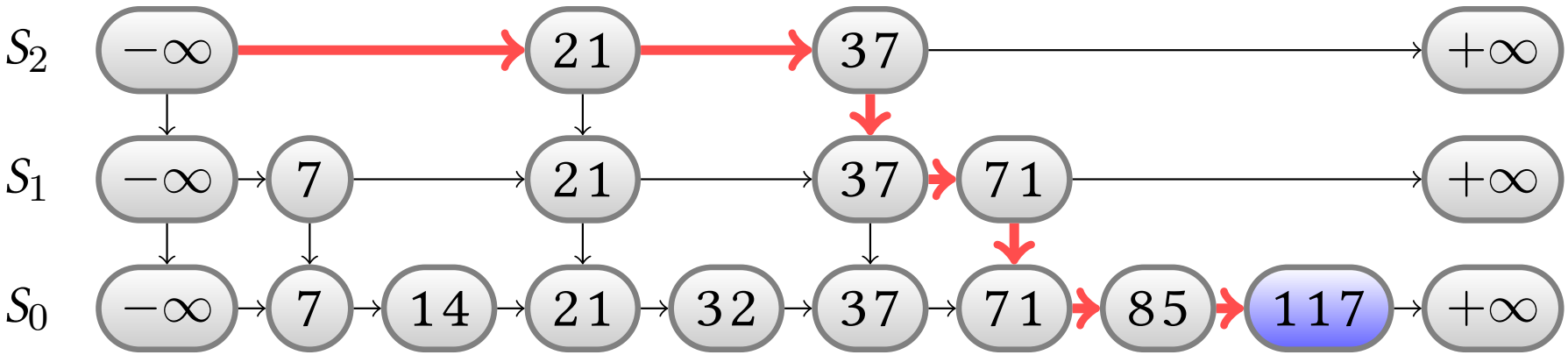


样例跳跃表



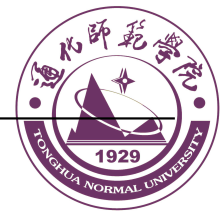
跳表的搜索

查找元素 117，或者 118。即：find(117)，find(118) 如下图：



跳跃表查找源代码

```
1  /* 如果存在 x，返回 x 所在的节点，
2   * 否则返回 x 的后继节点 */
3  find(x){
4    p = top;
5    while (1) {
6      while (p->next->key < x)
7        p = p->next;
8      if (p->down == NULL)
9        return p->next;
10     p = p->down;
11   }
12 }
```

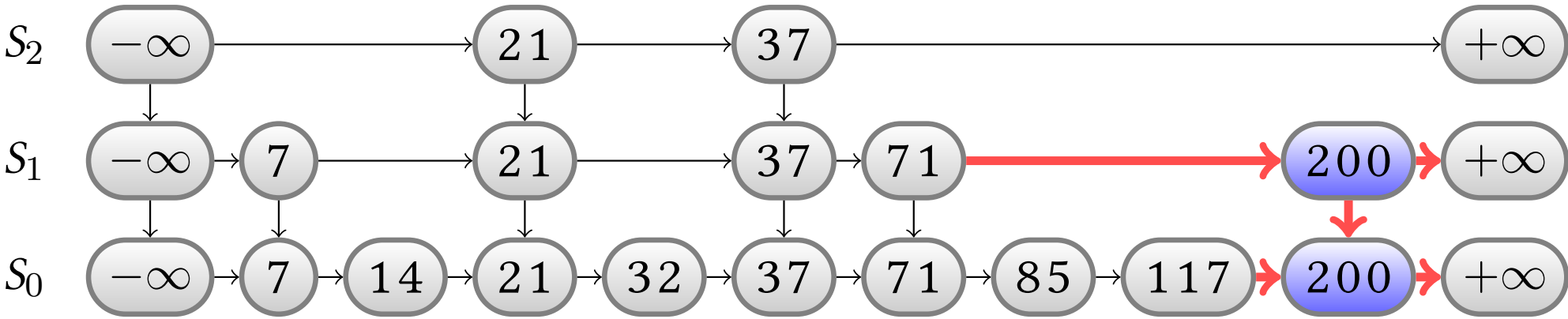


跳跃表插入 I

先确定该元素要占据的层数 k（采用丢硬币的方式，这完全是随机的）。

然后在 $S_0, \cdots, S_k$ 各个层的链表都插入元素。

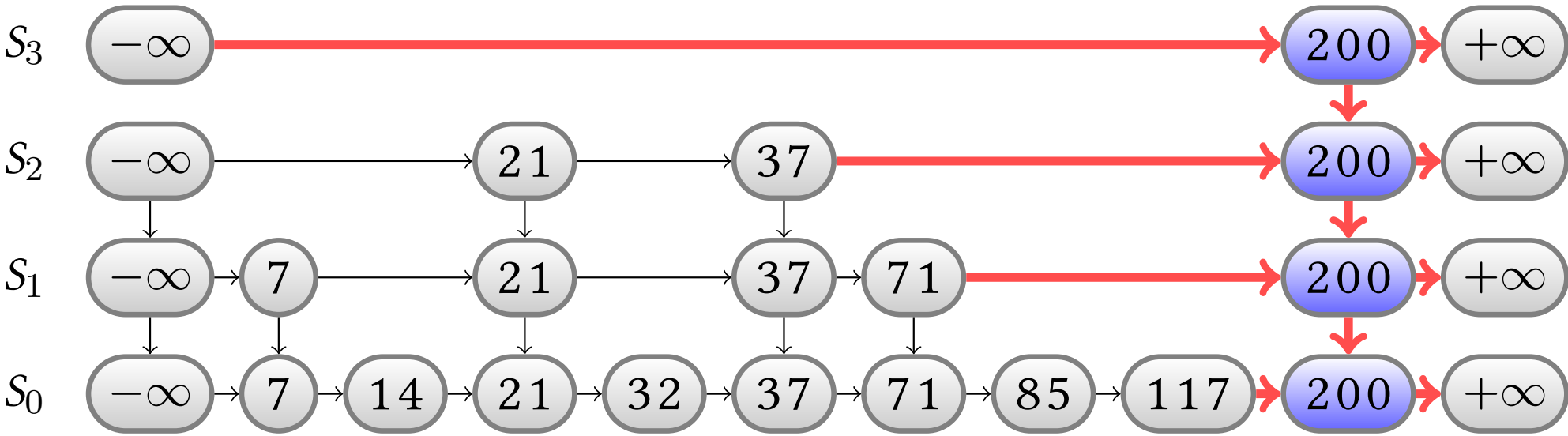
例子：插入 200，$k = 2$。



跳跃表插入 II

如果 k 大于链表的层数，则要添加新的层。

例子：插入 200，$k = 4$。



丢硬币决定 k

插入元素的时候，元素所占有的层数完全是随机的，通过以下随机算法产生：

```
1 int random_level( ){
2    int k = 1;
3    while (random(0,1))
4       k++;
5    return k;
6 }
```

相当于做一次丢硬币的实验，遇到正面，则继续丢，如果遇到反面，则停止；用实验中丢硬币的次数 k 作为元素占有的层数。显然随机变量 k 满足参数为 $p = 1/2$ 的几何分布，k 的期望值 $E[K] = 1/p = 2$。就是说，各个元素的层数，期望值是 2 层。(还有其他的算法，但是期望值一定是 2)。

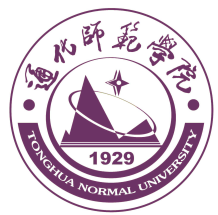
背景知识 期望值

在概率论和统计学中，一个离散性随机变量的期望值（或数学期望、或均值，亦简称期望）是试验中每次可能结果的概率乘以其结果的总和。换句话说，期望值是随机试验在同样的机会下重复多次的结果计算出的等同“期望”的平均值。需要注意的是，期望值并不一定等同于常识中的“期望”——“期望值”也许与每一个结果都不相等。（换句话说，期望值是该变量输出值的平均数。期望值并不一定包含于变量的输出值集合里。）

例如，掷一枚六面骰子的期望值是 3.5，计算如下：

$$1 \times \frac{1}{6} + 2 \times \frac{1}{6} + 3 \times \frac{1}{6} + 4 \times \frac{1}{6} + 5 \times \frac{1}{6} + 6 \times \frac{1}{6} = 3.5$$

3.5 不属于可能结果中的任一个。

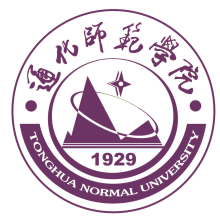


赌博不会发家致富

赌博是期望值的一种常见应用。例如，美国的轮盘中常用的轮盘上有 38 个数字，每一个数字被选中的概率都是相等的。赌注一般押在其中某一个数字上，如果轮盘的输出值和这个数字相等，那么下赌者可以将相当于赌注 35 倍的奖金（原注拿回）共 36，若输出值和下注数字不同，则赌注就输掉了。因此，考虑到 38 种所有的可能结果，以 1 美元赌注押一个数字上获利的期望值为：

$$E(x) = -1 \times \frac{37}{38} + 35 \times \frac{1}{38} = -\frac{1}{19} = \$ - 0.0526$$

结果约等于 -0.0526 美元。也就是说，平均起来每赌 1 美元就会输掉 5 美分，即相当于投注 38 次，赢了 1 次。



穿越回当下！掷硬币！

一次正面的可能性 $p = \frac{1}{2}$，两次正面的可能性 $p^2 = \frac{1}{4}$，依次类推！

$E(X) = 1 \times p + 2 \times p^2 \cdots$ 咋算？

$$\begin{cases} E(X) = 1 \times p + 2 \times p^2 \cdots & \cdots\cdots ① \\ p \times E(X) = 1 \times p^2 + 2 \times p^3 \cdots & \cdots\cdots ② \end{cases}$$

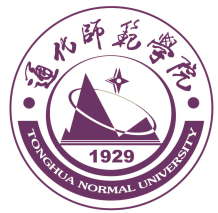
① − ② 得：

$$(1-p)E(x) = p + p^2 + P3 + \cdots = p \times \frac{1-p^n}{1-p} = 1 - p^n = 1$$

$$\therefore E(x) = \frac{1}{1-p}$$

∴ 丢硬币某一面一直向上的期望值是

同学们想起高中的数学了??!!!



穿越回当下！掷硬币！

一次正面的可能性 $p = \frac{1}{2}$，两次正面的可能性 $p^2 = \frac{1}{4}$，依次类推！

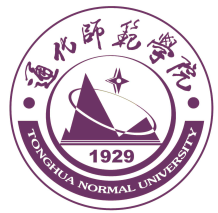
$$\begin{cases} E(X) = 1 \times p + 2 \times p^2 \cdots & \cdots\cdots ❶ \\ p \times E(X) = 1 \times p^2 + 2 \times p^3 \cdots & \cdots\cdots ❷ \end{cases}$$

❶ − ❷ 得：

$$(1-p)E(x) = p + p^2 + P3 + \cdots = p \times \frac{1-p^n}{1-p} = 1 - p^n = 1$$

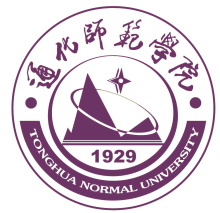
$$\therefore E(x) = \frac{1}{1-p}$$

∴ 丢硬币某一面一直向上的期望值是 2 .



跳跃表插入源代码 I

```
1  int insert(val x){
2    int i;
3    int j = n; //n 是当前表所拥有的 level 数
4    cell *p[k]; //指针数组,用来保存每一层要插入元素的前驱
5    cell *p1;  p1 = top->next;
6    while(p1){
7      while(p1->next->val < x) p1=p1->next;
8      if(j <= k){
9        p[j-1] = p1; //保存每一层的指针
10       p1 = p1->down; //指向下一层
11       j--;
12     }
13   }
14   //下面的代码是将 x 插入到各层
15   for (i = 0; i<k; i++){
```

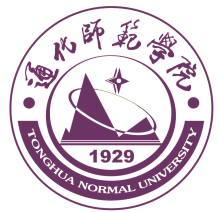


跳跃表插入源代码 II

```
16      if(p[i]==NULL){//k>n 的情况,需要创建一个层
17        //创建层的第一个元素,并将 top 指向它
18        cell *elementhead =
19                  (cell *) malloc(sizeof(cell));
20        element->val = -1;  element->down = top;
21        top = elementhead;
22        //创建最后一个元素
23        cell *elementtail =
24                  (cell *) malloc(sizeof(cell));
25        elementtail->val = 1;
26        elementtail->next =
27                  elementtail->down = NULL;
28        //在该层中创建并插入 x
29        cell *element =
30                  (cell *) malloc(sizeof(cell));
```

跳跃表插入源代码 III

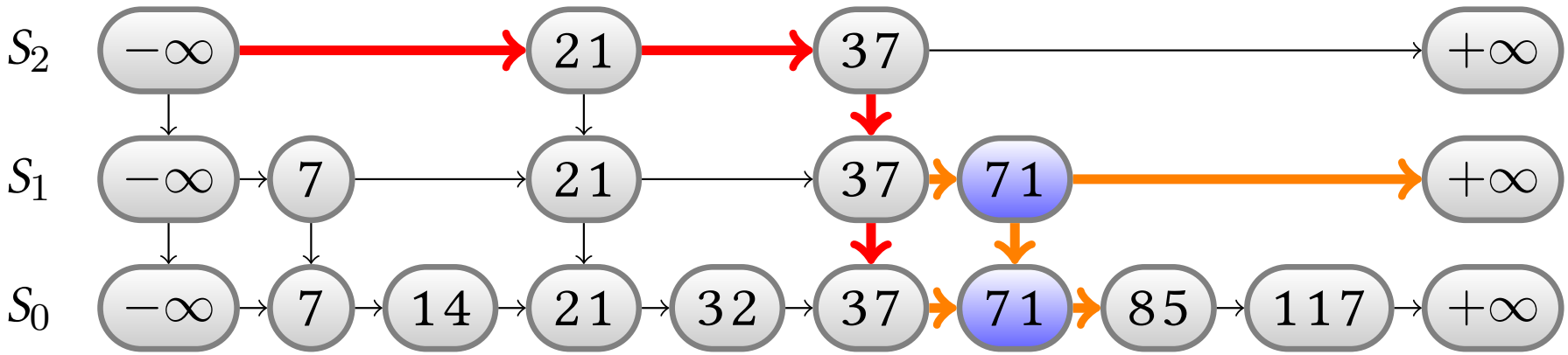
```
31          element->val = x;
32          elementhead->next = element;
33          element->next = elementtail;
34          element->down = p[i-1]->next;
35        }
36        //正常插入一个元素
37        cell *element =
38                        (cell *) malloc(sizeof(cell));
39        element->val = x;
40        element->next = p[i]->next;
41        element->down = (i=0?NULL:(p[i-1]->next));
42        p[i]->next = element;
43      }
44      return 0;
45  }
```



跳表的删除

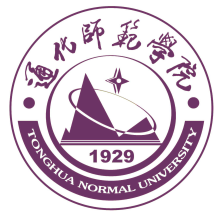
在各个层中找到包含 x 的节点，使用标准的 delete from list 方法删除该节点。

例子：删除 71（红色查找，橘色删除）



跳跃表删除源代码

```
1  int delete(val x){
2   int i = n; //n 表示当前总层数
3   cell *p, *p1;      p = top;
4   while(n>0){ //⇓ 第七行,该层只一个节点
5    while(p->next->val < x) p=p->next;
6    if(p->next->val == x){//假如当前层存在节点 x, 删除
7     if(p = top && p->next->next->val == INT_MAX){
8      top = p->down;   free(p->next->next);
9      free(p->next);   free(p);   p = top;
10    }else{
11     p1 = p->next; p->next = p1->next; free(p1);
12    }
13   } p = p->down;   n--;
14  }
15 }
```



加油!!!

同学们课后要努力!!!!