Lecture 1: Dictionary & Skiplist

(a) 散列：循值访问
(b) 跳转表：结构
(c) 跳转表：算法
(d) 应用：LSM算法

预备知识

几何分布

几何分布（Geometric distribution）是离散型概率分布。其中一种定义为：在n次伯努利试验中，试验k次才得到第一次成功的机率。详细地说，是：前k-1次皆失败，第k次成功的概率。几何分布是帕斯卡分布当r=1时的特例。

在伯努利试验中，记每次试验中事件A发生的概率为p，试验进行到事件A出现时停止，此时所进行的试验次数为X，其分布列为：

$$P(X=k)=(1-p)^{k-1}p, k=1,2,...$$

预备知识

几何分布

m = n-1次失败，第n次成功，其数学期望为：

A. $E[X] = \frac{1-p}{p}$

B. $E[X] = \frac{1}{p}$

C. Other value

预备知识

二项式

设n,k 为正整数，如何证明:

$$\left(1 - 1/2^k\right)^n \geq 1 - n/2^k$$

We are shaped by our thoughts;

we become what we think.

- Buddha

Man's thought is shaped by his tongue.

- Anonymous

The diversity of languages is not

a diversity of signs and sounds, but

a diversity of views of the world.

- Wilhelm von Humboldt, 1820

9.词典

(a) 散列：循值访问

邓俊辉

deng@tsinghua.edu.cn

联合数组

- 数组？再常见不过，比如：fib[0] = 0, fib[1] = 1, fib[2] = 1, fib[3] = 2, ...

- associative array：与此前的数组有何区别？

- 根据数据元素的取值，直接访问！

```
style["关羽"] = "云长"
style["张飞"] = "翼德"
style["赵云"] = "子龙"
style["马超"] = "孟起"
```

下标不再是整数，甚至没有大小次序——更为直观、便捷

- 支持的语言：

Snobol4、MUMPS、SETL、Rexx、AWK、Java、Python、Perl、Ruby、PHP、...

映射 + 词典 = 符号表

- 词条：`entry = (key, value)`
- 映射：`Map` = 词条的集合，其中各词条（的关键码）互异

  ADT：`get(key)`，`put(key, value)`，`remove(key)`
- 较之BST，关键码之间`未必`可比较`call-by-value`

  较之PQ，查找对象更广泛，不限于最大、最小词条
- 词典：`Dictionary`：与映射基本相同，但允许词条（的关键码）雷同

  `Sorted dictionary`：关键码之间可定义`全序`关系的词典
- 映射和词典都是`动态`的，统称符号表`symbol table`

Dictionary

```
template <typename K, typename V> //key、value
struct Dictionary { //Dictionary模板类
    virtual int size() = 0; //查询当前的词条总数
    virtual bool put( K ) = 0; //插入词条(key, value)
    virtual V* get( K ) = 0; //查找以key为关键码的词条
    virtual bool remove( K ) = 0; //删除以key为关键码的词条
};
```

- 尽管诸如Java::TreeMap等实现仍然要求支持比较器，但实际上词典中的词条，只需支持比对判等操作，而不必支持大小比较

Dictionary

- 在这里，无论对外的访问方式，还是内部的存储方式

  都更直接地依据数据对象自身的取值

  key与value地位等同，不必区分

- 循值访问 call-by-value：方式更为自然，适用范围也更广泛

- 回忆一下，初次接触程序设计时，你首先想到的应该就是这种方式

Java: HashMap + Hashtable

```
import java.util.*;
public class Hash {
    public static void main(String[] args) {
        HashMap HM = new HashMap(); //Map
            HM.put("东岳", "泰山"); HM.put("西岳", "华山"); HM.put("南岳", "衡山");
            HM.put("北岳", "恒山"); HM.put("中岳", "嵩山"); System.out.println(HM);
        Hashtable HT = new Hashtable(); //Dictionary
            HT.put("东岳", "泰山"); HT.put("西岳", "华山"); HT.put("南岳", "衡山");
            HT.put("北岳", "恒山"); HT.put("中岳", "嵩山"); System.out.println(HM);
    }
}
```

Perl: %Hash Type

❖由字符串（string）标识的一组无序标量（scalar） //亦即MAP

❖my %hero = ("云长"=>"关羽", "翼德"=>"张飞", "子龙"=>"赵云", "孟起"=>"马超");

```
 foreach $style (keys %hero) # Hash类型的变量由%引导
   { print "$style => $hero{$style}\n"; }
```

❖$hero{"汉升"} = "黄忠";

```
 foreach $style (keys %hero)
   { print "$style => $hero{$style}\n"; }
 foreach $style (reverse sort keys %hero)
   { print "$style => $hero{$style}\n"; }
```

Python: Dictionary Class

```
❖beauty = dict # Python dictionary (hashtable)
    ( { "沉鱼":"西施", "落雁":"昭君", "闭月":"貂蝉", "羞花":"玉环" } )
  print beauty
❖beauty["红颜"] = "圆圆"
  print beauty
❖for alias, name in beauty.items():
    print alias, ":", name
❖for alias, name in sorted(beauty.items()):
    print alias, ":", name
❖for alias in sorted(beauty.keys(), reverse = True):
    print alias, ":", beauty[alias]
```

Ruby: Hash Table

```
scarborough = { # declare and initialize a hash table
    "P"=>"parsley",  "S"=>"sage",
    "R"=>"rosemary", "T"=>"thyme"
}
puts scarborough # output the hash table
for k in scarborough.keys # output hash table items
    puts k + "=>" + scarborough[k] # 1-by-1
end
for k in scarborough.keys.sort # output hash table items
    puts k + "=>" + scarborough[k] # 1-by-1 in order
end
```

课后

- 了解Java中HashMap与Hashtable的异同
- 安装JDK (http://www.java.com)

  尝试HashMap和Hashtable类
- 安装Perl (http://www.perl.org)

  尝试%Hash类型
- 安装Python (http://www.python.org)

  尝试Dictionary类
- 安装Ruby (http://www.ruby-lang.org)

  尝试Hash Table

9.词典

（b）跳转表：结构

去沿江上下，或二十里，或三十里，选
高阜处置一烽火台，每台用五十军守之。

邓俊辉

deng@tsinghua.edu.cn

动机与思路

❖ 可否综合向量与列表的优势，高效地实现词典接口？

具体地，如何使得各接口的效率均为$O(\log n)$？

❖ [William Pugh, 1989] Skip Lists: A Probabilistic Alternative to Balanced Trees

Skip lists are data structures that use

probabilistic balancing rather than

strictly enforced balancing

Algorithms for insertion and deletion in skip lists

are much simpler and significantly faster than

equivalent algorithms for balanced trees

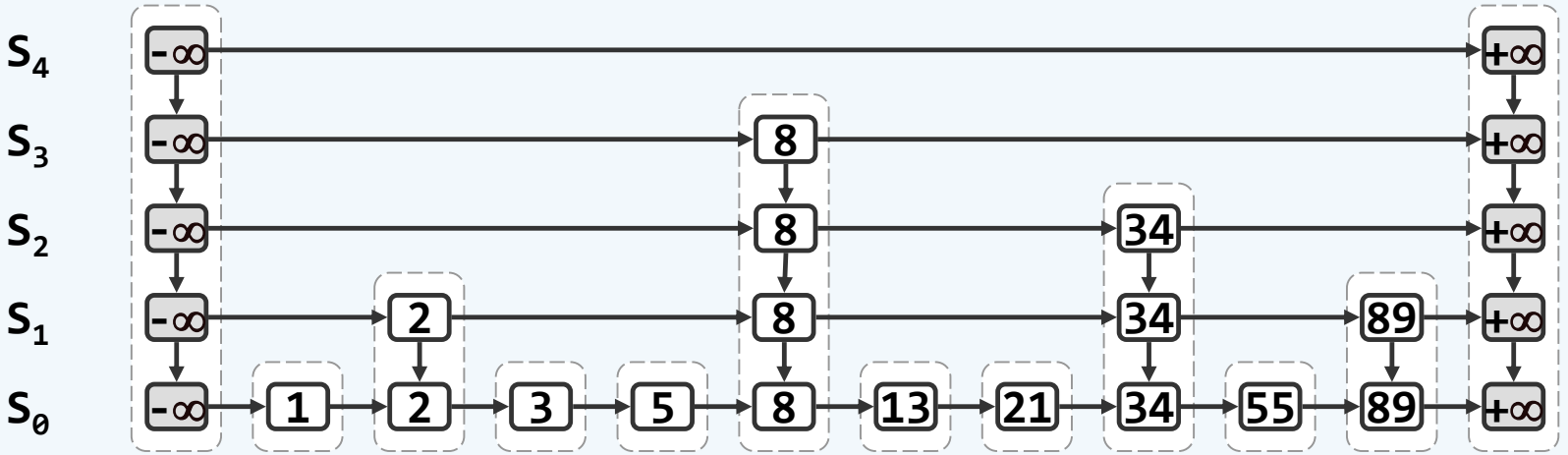
❖ 本节介绍非确定型跳转表，确定型 deterministic 跳转表可自学

结构：设计

❖ 分层次、相互耦合的多个列表：S_0, S_1, S_2, ..., S_h //层高 = h

❖ S_h称作顶层 top

S_0称作底层 bottom



❖ 各节点至多拥有四个引用：

横向为层 (level) : prev()、next() //设有头、尾哨兵

纵向成塔 (tower) : above()、below()

空间性能

❖ 较之常规的单层列表

每次操作过程中，需访问的节点是否会实质地增多？

每个节点都至多可能重复h次，空间复杂度是否因此有实质增加？ //先来回答后者...

❖ 生长概率逐层减半：S_k中的每个关键码，在S_{k+1}中依然出现的概率，均为 p = 1/2

——暂且假设成立，稍后说明如何保证

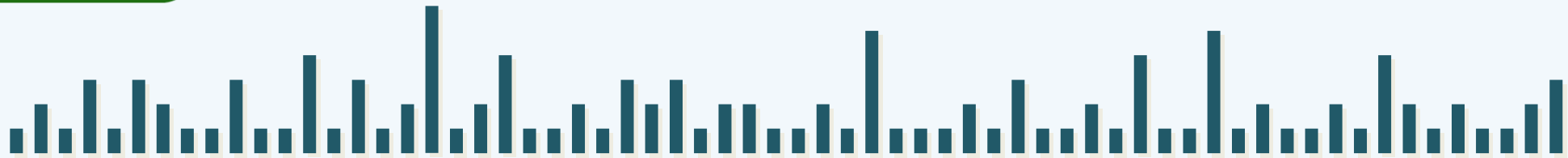
❖ 可见，各塔的高度符合几何分布：

$$Pr(h = k) = p^{k-1} \cdot (1 - p)$$

❖ 于是，期望的塔高E(h) = 1 / (1 - p) = 2

❖ 什么，没有学过概率？不要紧，有初等的解释...

空间性能

❖ 

既然生长概率逐层减半：　S_0中任一关键码在S_k中依然出现的概率均为2^{-k}

第k层节点数的期望值$E(|S_k|) = n \cdot 2^{-k} = n/2^k$

❖ 于是，所有节点期望的总数（即各层列表所需空间总和）为

$$E(\sum_k |S_k|) = \sum_k E(|S_k|) = n \times (\sum_k 2^{-k}) < 2n = \mathcal{O}(n)$$

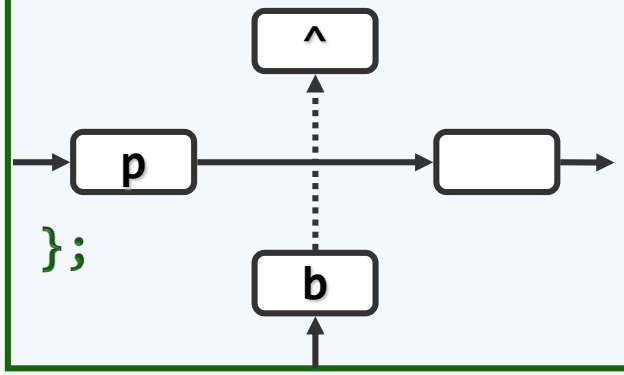
❖ 定理：　跳转表所需空间为expected-$\mathcal{O}(n)$

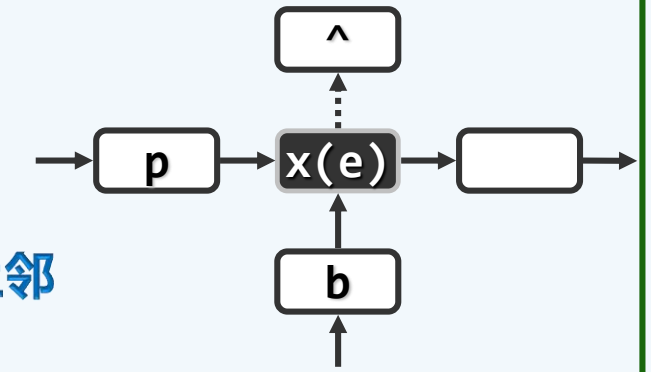
❖ 类比：　半衰期为1年的放射性物质中，各粒子的平均寿命不过2年

❖ 更为细致地，平均塔高的方差或标准差是否足够地小？　//比照稍后对层高的分析

结构：QuadList

```
template <typename T> class Quadlist { //四联表
   private: int _size; QuadlistNodePosi(T) header, trailer; //规模、哨兵
   protected: void init(); int clear(); //初始化、清除所有节点
   public:  QuadlistNodePosi(T) first() const { return header->succ; } //首节点
            QuadlistNodePosi(T) last() const { return trailer->pred; } //末节点
            T remove( QuadlistNodePosi(T) p ); //删除p
            QuadlistNodePosi(T) insertAfterAbove( //插入
                     T const & e, //数据项e，使之成为
                     QuadlistNodePosi(T) p, //p的后继，以及
                     QuadlistNodePosi(T) b = NULL ); //b的上邻
};
```





结构：Skiplist

```
template < typename K, typename V > class Skiplist : //多重继承
public Dictionary< K, V >, public List< Quadlist< Entry< K, V > > * > {
   protected:
      bool skipSearch( ListNode< Quadlist< Entry< K, V > > * > * & qlist,
                       QuadlistNode< Entry< K, V > > * & p, K & k );
   public:
      int size() { return empty() ? 0 : last()->data->size(); } //词条总数
      int level() { return List::size(); } //层高，即Quadlist总数
      bool put( K, V ); //插入（Skiplist允许词条重复，故必然成功）
      V * get( K ); //读取（基于skipSearch()直接实现）
      bool remove( K ); //删除
};
```

9.词典

（c） 跳转表：算法

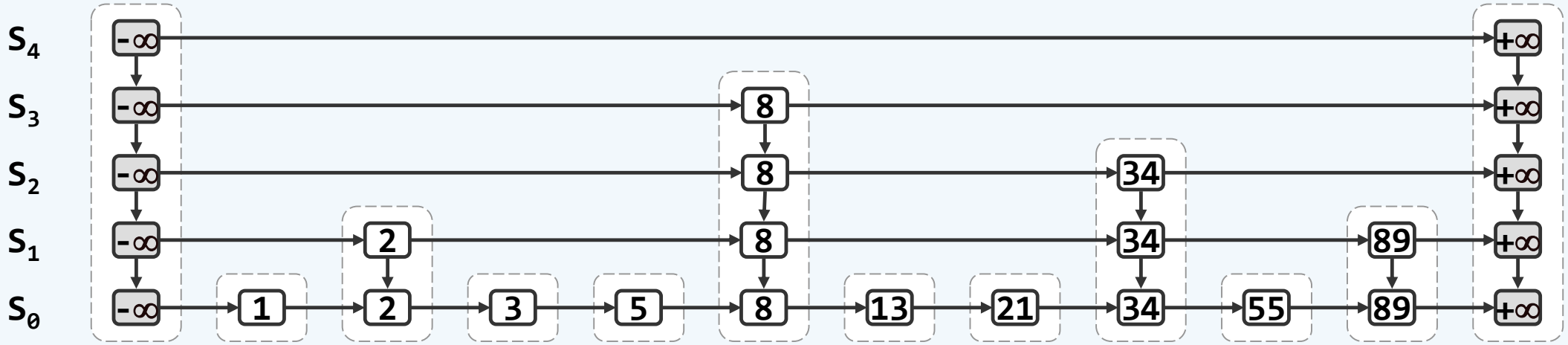
只见参仙老怪梁子翁笑嘻嘻的站起身来，向众人拱了拱手，缓步走到庭中，忽地跃起，左足探出，已落在欧阳克插在雪地的筷子之上，拉开架子，...，把一路巧打连绵的“燕青拳”使了出来，脚下纵跳如飞，每一步都落在竖直的筷子之上。

邓俊辉

deng@tsinghua.edu.cn

查找：策略与算法

❖ 减而治之：由粗到细 = 由高至低



❖ 实例：成功（21、34、1、89），失败（80、0、99）

❖ 观察：查找时间取决于横向、纵向的累计跳转次数

那么，是否可能因层次过多，首先导致纵向跳转过多？

查找：实现

```
template <typename K, typename V> bool Skiplist<K, V>::skipSearch(
    ListNode< Quadlist < Entry< K, V > > * > * & qlist, //从指定层qlist的
    QuadlistNode< Entry< K, V > > * & p, K & k ) { //首节点p出发，向右、向下查找k
    while ( true ) { //在每一层从前向后查找，直到出现更大的key，或者溢出至trailer
       while ( p->succ && ( p->entry.key <= k ) ) p = p->succ; p = p->pred;
       if ( p->pred && ( k == p->entry.key ) ) return true; //命中则成功返回
       qlist = qlist->succ; //否则转入下一层
       if ( ! qlist->succ ) return false; //若已到穿透底层，则意味着失败；否则...
       p  =  p->pred ? p->below : qlist->data->first(); //转至当前塔的下一节点
    } //确认：无论成功或失败，返回的p均为其中不大于e的最大者?
} //体会：得益于哨兵的设置，哪些环节被简化了?
```

查找：纵向跳转/层高

❖ 引理：随着k的增加，S_k 为空 的概率急剧 上升

准确地，$Pr(|S_k| = 0) = (1 - 1/2^k)^n \geq 1 - n/2^k$

❖ 于是：随着k的增加，S_k 非空 的概率急剧 下降

准确地，$Pr(|S_k| > 0) \leq n \times 2^{-k}$

❖ 推论：跳转表高度 $h = O(\log n)$ 的概率 极大

❖ 比如：第 $k = 3\log n$ 层为空（当且仅当 $h < k$）的概率为

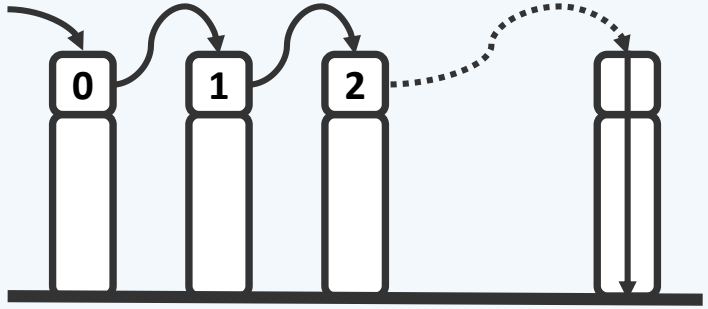
$Pr(h < k) \geq Pr(|S_k| = 0) \geq 1 - n/2^k = 1 - n/n^3 = 1 - n^{-2} \mapsto 1$

❖ 结论：查找过程中，纵向 跳转的次数，累计不过 expected-$O(\log n)$

查找：横向跳转/紧邻塔顶

❖ 那么：横向跳转是否可能很多次？比如ω(logn)，甚至Ω(n)？

❖ 观察：在同一水平列表中，横向跳转所经节点必然依次紧邻，而且每次抵达都是塔顶



❖ 于是：若将沿同层列表跳转的次数记作Y，则有几何分布：

$$Pr(Y = k) = (1 - p)^k \cdot p$$

❖ 定理：$E(Y) = (1 - p) / p = (1 - 0.5) / 0.5 =$ 1 次

同层列表中紧邻的塔顶节点，平均不过1 + 1 = 2 个

❖ 推论：查找过程中横向跳转所需时间 ≤ expected(2h) = expected-O(logn)

❖ 结论：跳转表的每次查找可在expected-O(logn)时间内完成

插入：算法实现

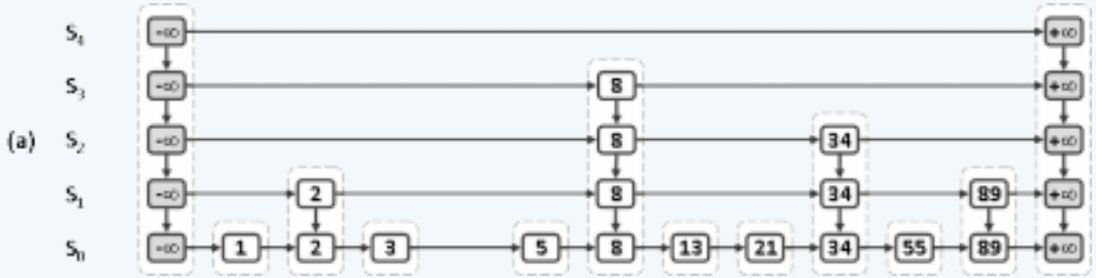
```
template <typename K, typename V> bool Skiplist< K, V >::put( K k, V v ) {
   Entry< K, V > e = Entry< K, V >( k, v ); //将被随机地插入多个副本的新词条
   if ( empty() ) insertAsFirst( new Quadlist< Entry< K, V > > ); //首个Entry
   ListNode< Quadlist< Entry< K, V > > * > * qlist = first(); //从顶层列表的
   QuadlistNode< Entry< K, V > > * p = qlist->data->first(); //首节点开始
   if ( skipSearch( qlist, p, k ) ) //查找适当的插入位置——若已有雷同词条，则
      while ( p->below ) p = p->below; //强制转到塔底
   qlist = last();
   QuadlistNode< Entry< K, V > > * b = qlist->data->insertAfterAbove( e, p );
   /* ... 以下，紧邻于p的右侧，以新节点b为基座，自底而上逐层长出一座新塔 ... */
```

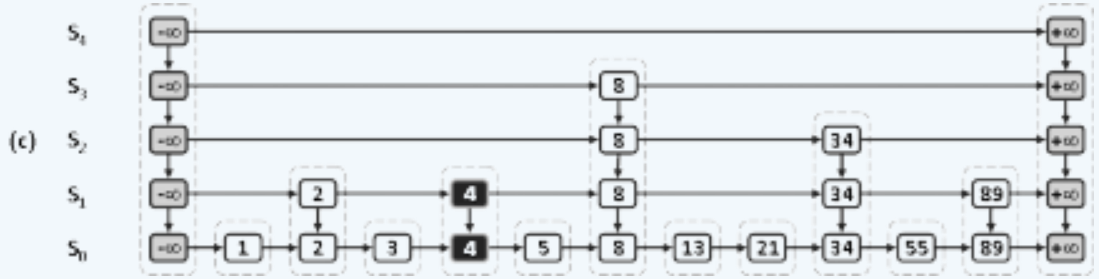
插入：算法实现

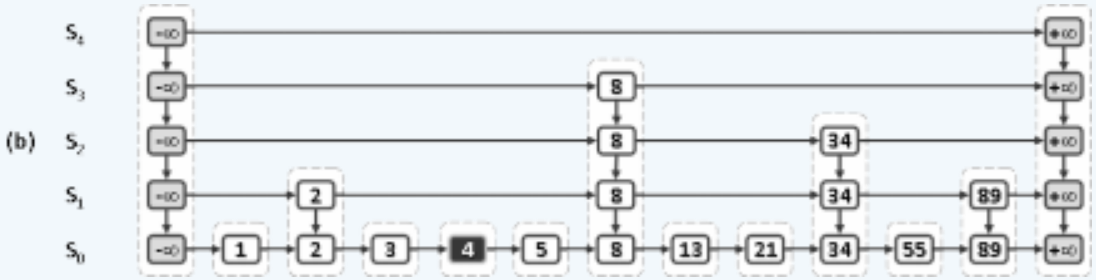
```
while ( rand() & 1 ) { //经投掷硬币，若新塔需再长高，则先找出不低于此高度的...
   while ( qlist->data->valid(p) && ! p->above ) p = p->pred; //最近前驱
   if ( ! qlist->data->valid(p) ) { //若该前驱是header
      if ( qlist == first() ) //且当前已是最顶层，则意味着必须
         insertAsFirst( new Quadlist< Entry< K, V > > ); //先创建新层，再
      p = qlist->pred->data->first()->pred; //将p转至上一层的header
   } else p = p->above; ////否则，可径自将p提升至该高度
   qlist = qlist->pred; //上升一层，并在该层将新节点
   b = qlist->data->insertAfterAbove( e, p, b ); //插至p之后、b之上
} //while ( rand() & 1 )
return true; //Skiplist允许重复元素，故插入必成功
}
```

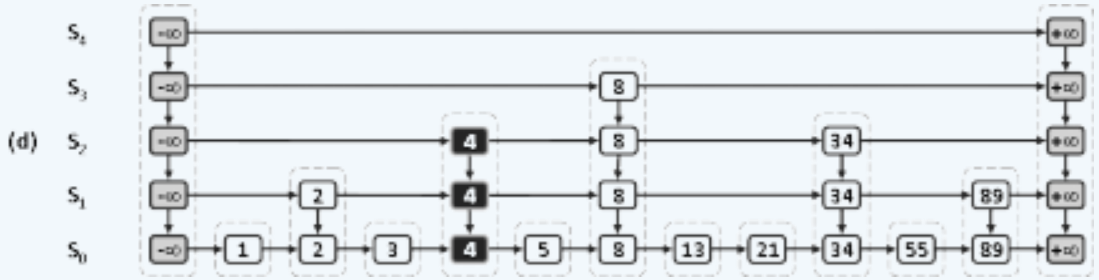
插入：实例

❖ 实例：一般（4、20、40），边界（0、99）









删除：算法实现

```
//插入的逆过程

template <typename K, typename V> bool Skiplist< K, V >::remove( K k ) {
   if ( empty() ) return false; //空表
   ListNode< Quadlist< Entry< K, V > > * > * qlist = first(); //从顶层Quadlist
   QuadlistNode< Entry< K, V > > * p = qlist->data->first(); //的首节点开始
   if ( ! skipSearch( qlist, p, k ) ) //目标词条不存在，则
      return false; //直接返回
   /* ... */
```

删除：算法实现

```
do { //若目标词条存在，则逐层拆除与之对应的塔
   QuadlistNode< Entry< K, V > > * lower = p->below; //记住下一层节点
   qlist->data->remove(p); //删除当前层的节点后，再
   p = lower; qlist = qlist->succ; //转入下一层
} while ( qlist->succ ); //如上不断重复，直到塔基
while ( ! empty() && first()->data->empty() ) //逐一地
   List::remove( first() ); //清除已可能不含词条的顶层Quadlist
return true; //删除操作成功完成
} //体会：得益于哨兵的设置，哪些环节被简化了？
```

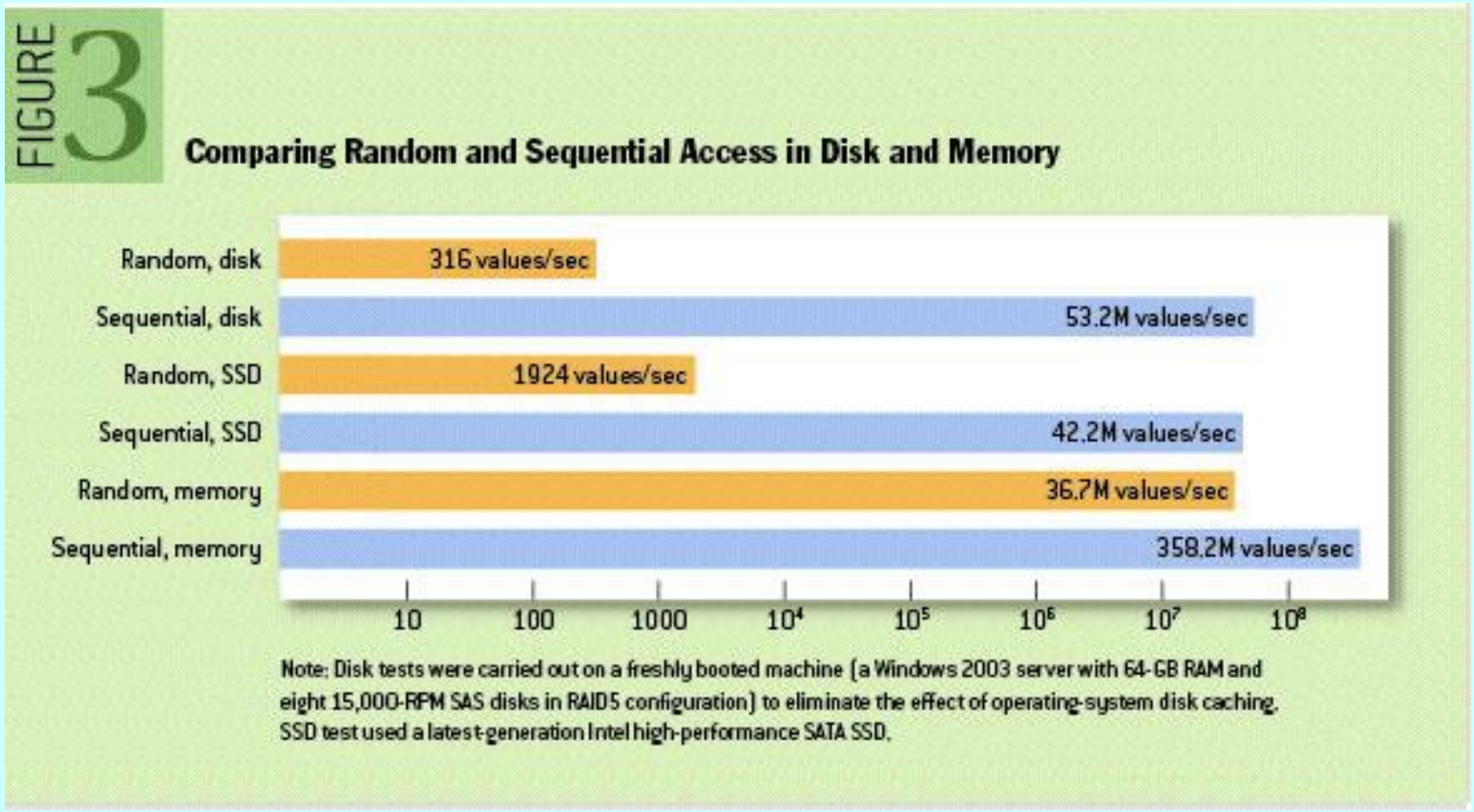
Performance

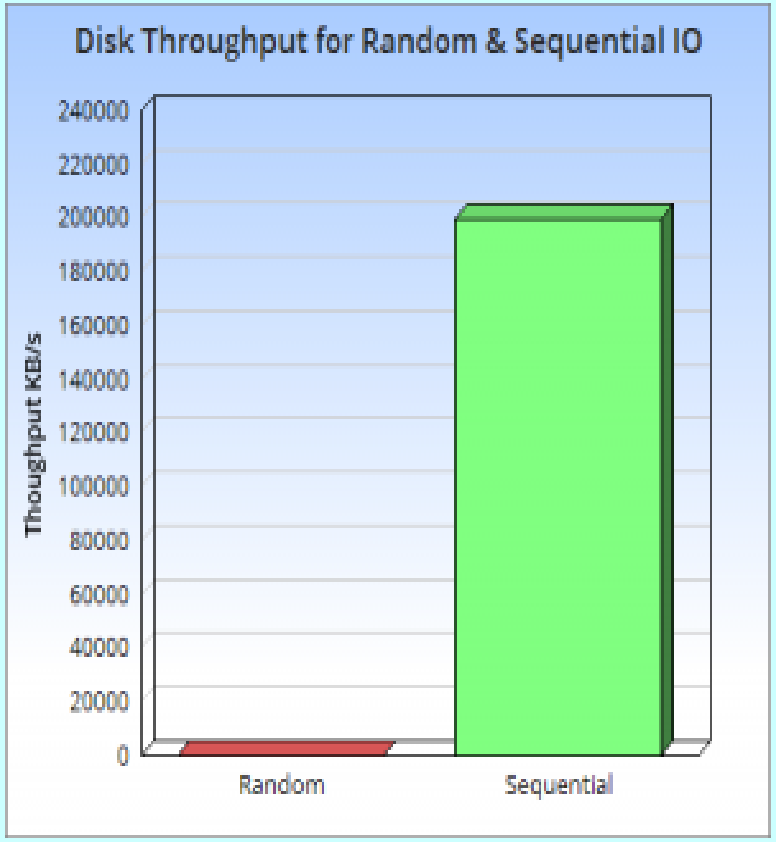
Implementation	Search Time	Insertion Time	Deletion Time
Skip lists	0.051 msec (1.0)	0.065 msec (1.0)	0.059 msec (1.0)
non-recursive AVL trees	0.046 msec (0.91)	0.10 msec (1.55)	0.085 msec (1.46)
recursive 2–3 trees	0.054 msec (1.05)	0.21 msec (3.2)	0.21 msec (3.65)
Self–adjusting trees:			
top-down splaying	0.15 msec (3.0)	0.16 msec (2.5)	0.18 msec (3.1)
bottom-up splaying	0.49 msec (9.6)	0.51 msec (7.8)	0.53 msec (9.0)

Table 2 - Timings of implementations of different algorithms

Source: William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees.

(d) 应用：LSM算法

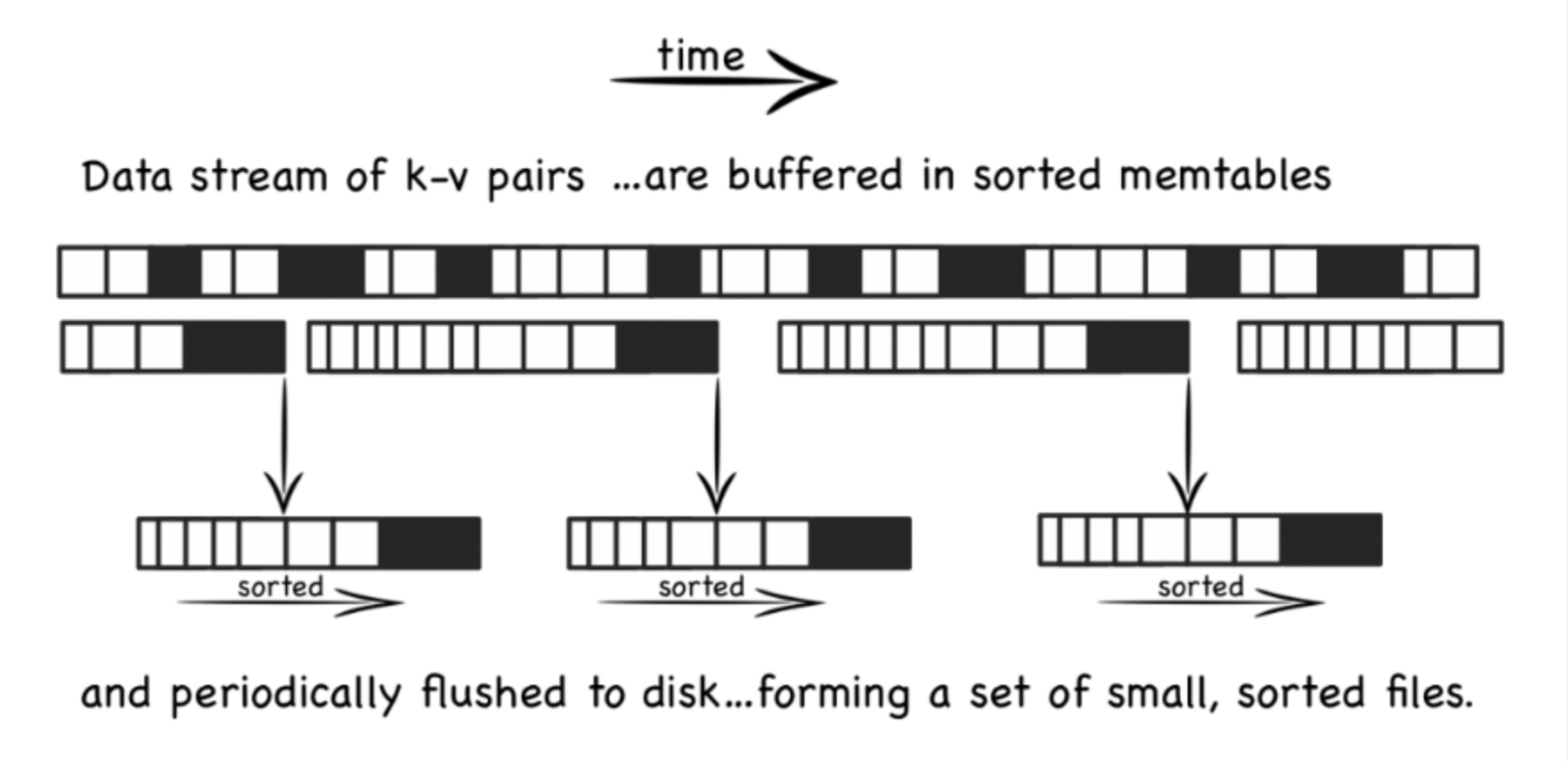




LSM模型利用批量写入解决了随机写入的问题，虽然牺牲了部分读的性能，但是大大提高了写的性能。

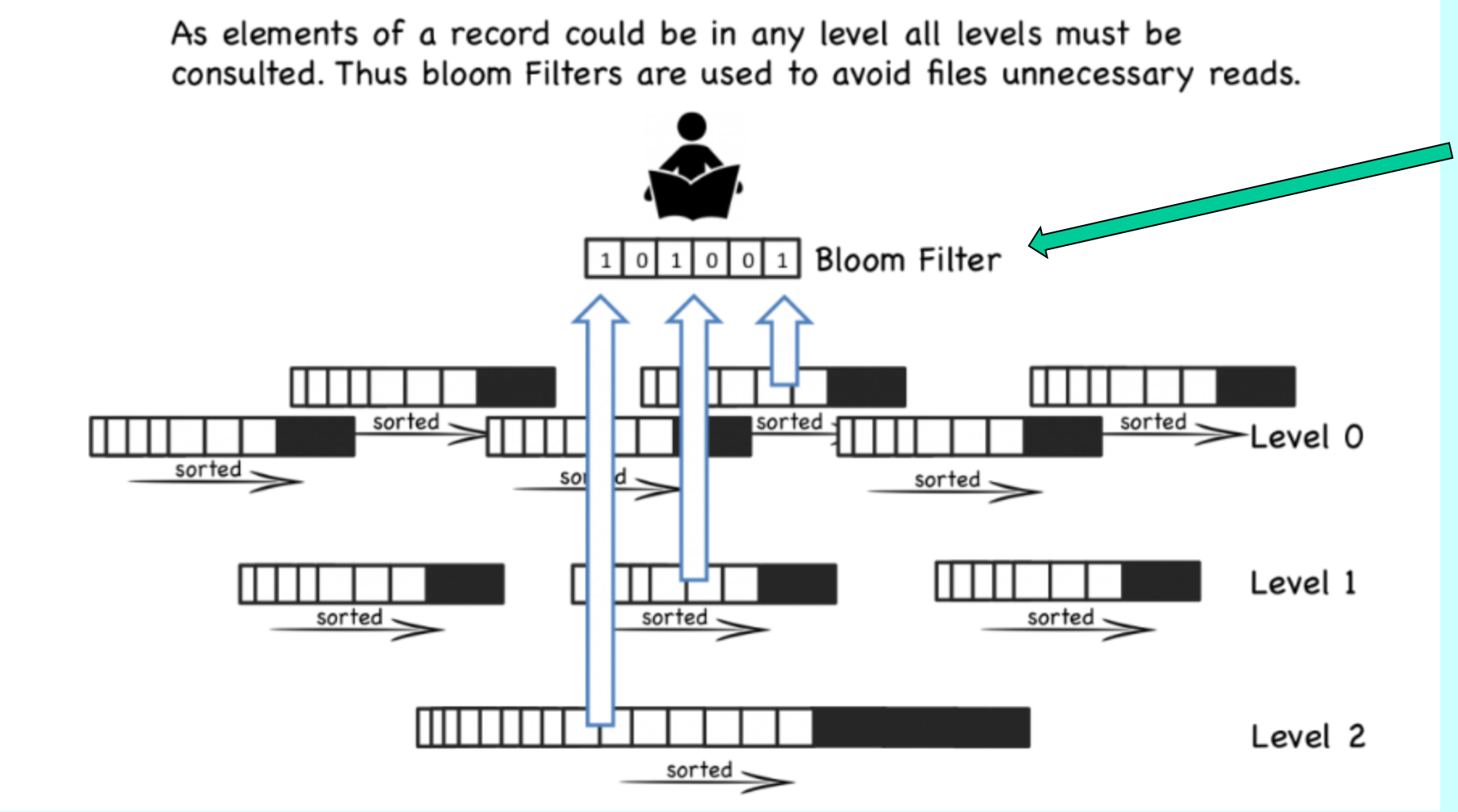
Source: https://queue.acm.org/detail.cfm?id=1563874

(d) 应用：LSM算法



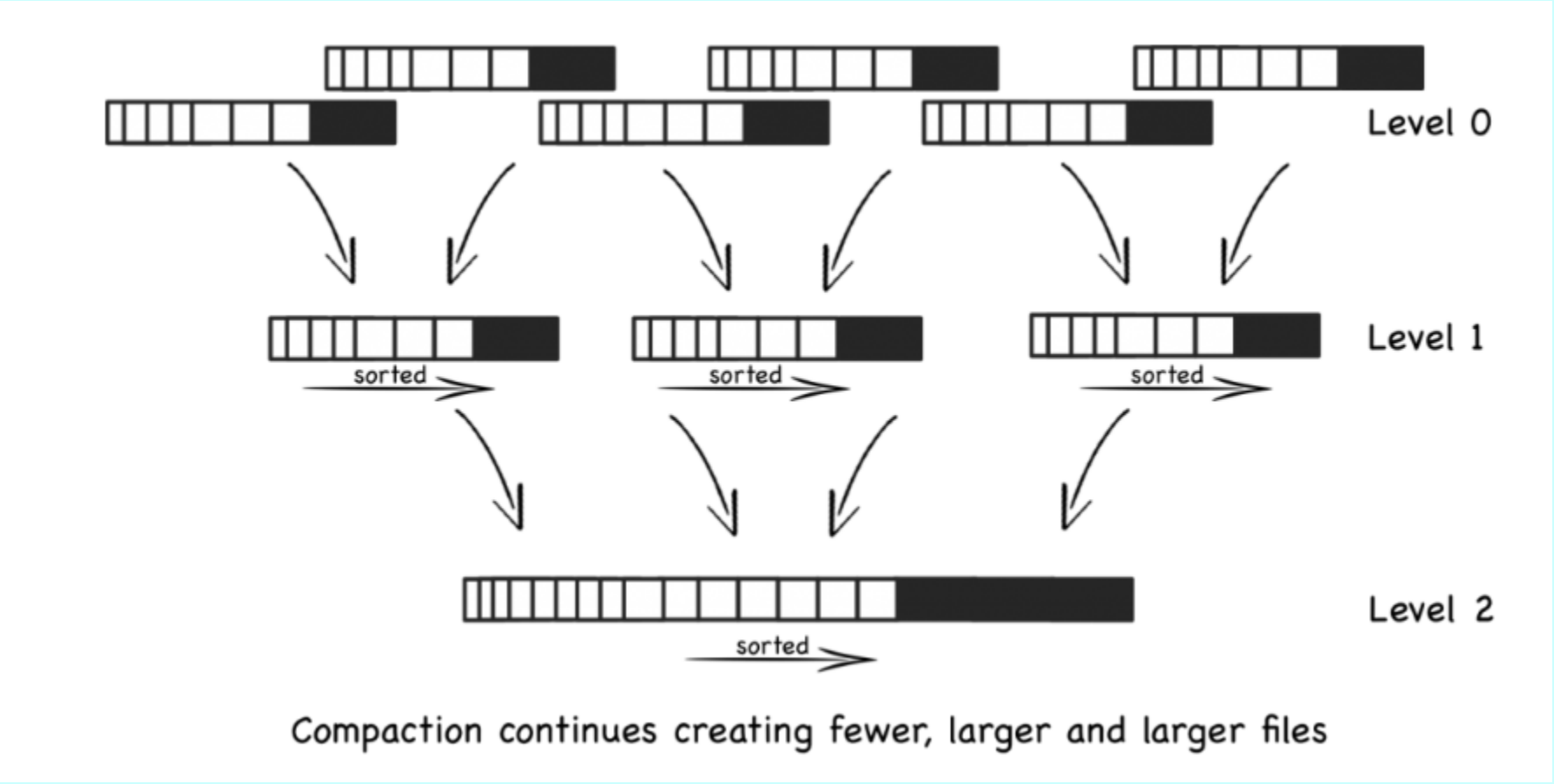
Source: http://www.benstopford.com/2015/02/14/log-structured-merge-trees/

(d) 应用：LSM算法



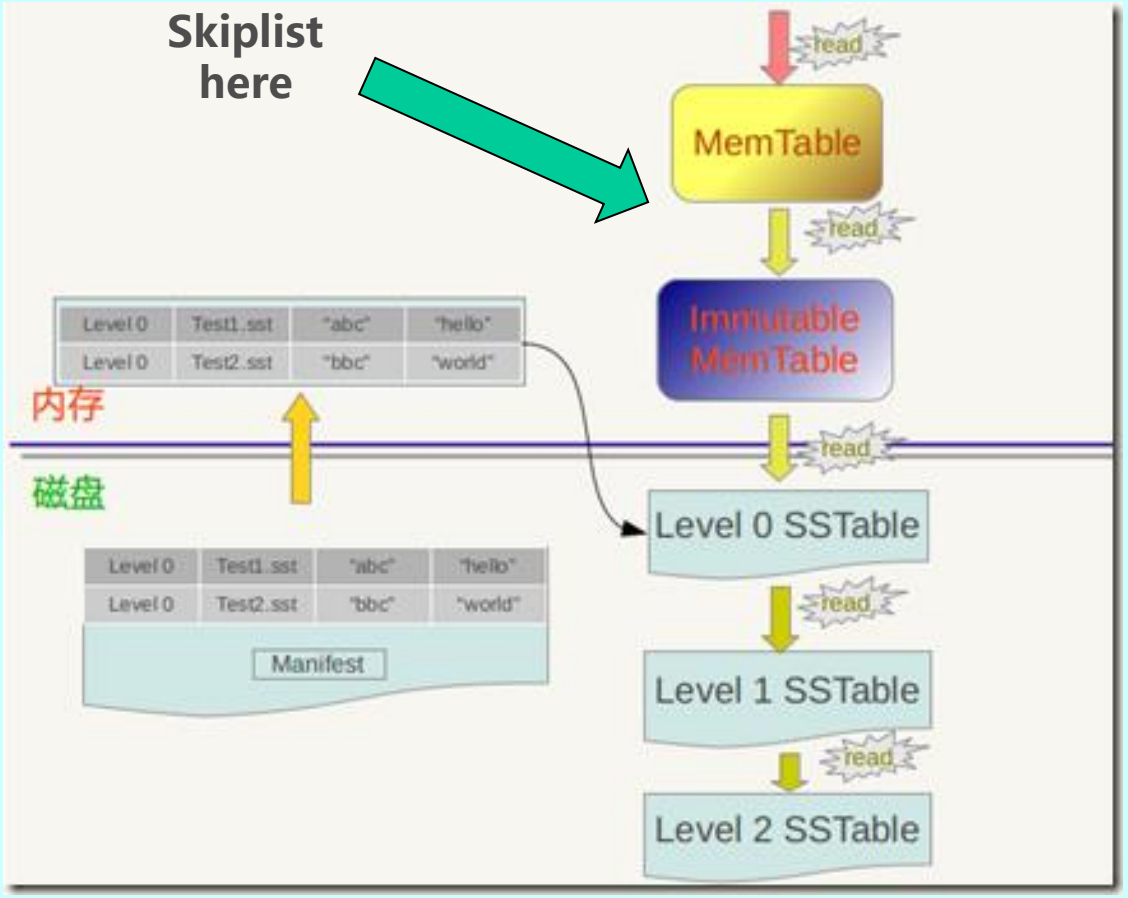
Source: http://www.benstopford.com/2015/02/14/log-structured-merge-trees/

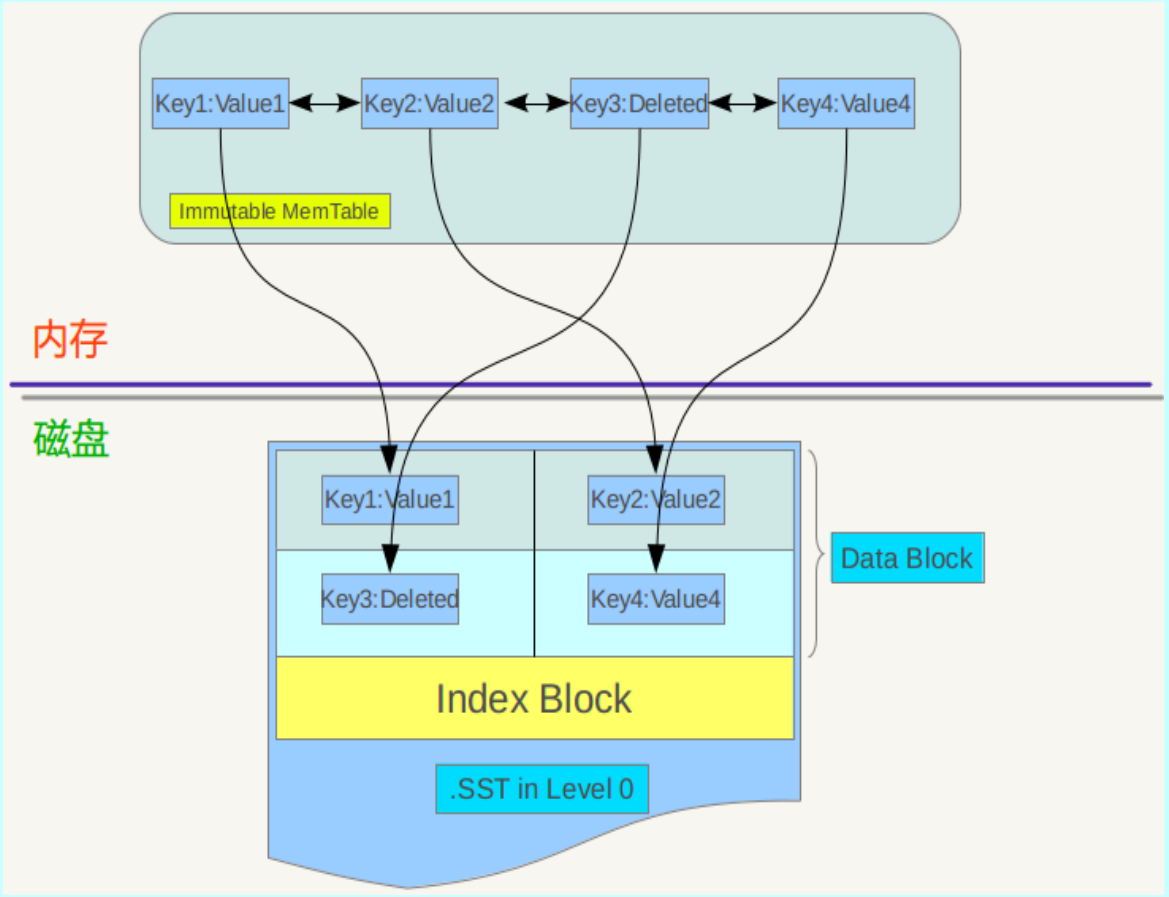
(d) 应用：LSM算法



Source: http://www.benstopford.com/2015/02/14/log-structured-merge-trees/

(d) 应用：LSM算法





Source: https://blog.csdn.net/anderscloud/article/details/7182165

Reference

1. William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. Commun. ACM 33, 6 (June 1990), 668–676. DOI:https://doi.org/10.1145/78973.78977
2. Log Structured Merge Trees. http://www.benstopford.com/2015/02/14/log-structured-merge-trees/

Next

- Set and Bloom filters

Backup

Formal Proof

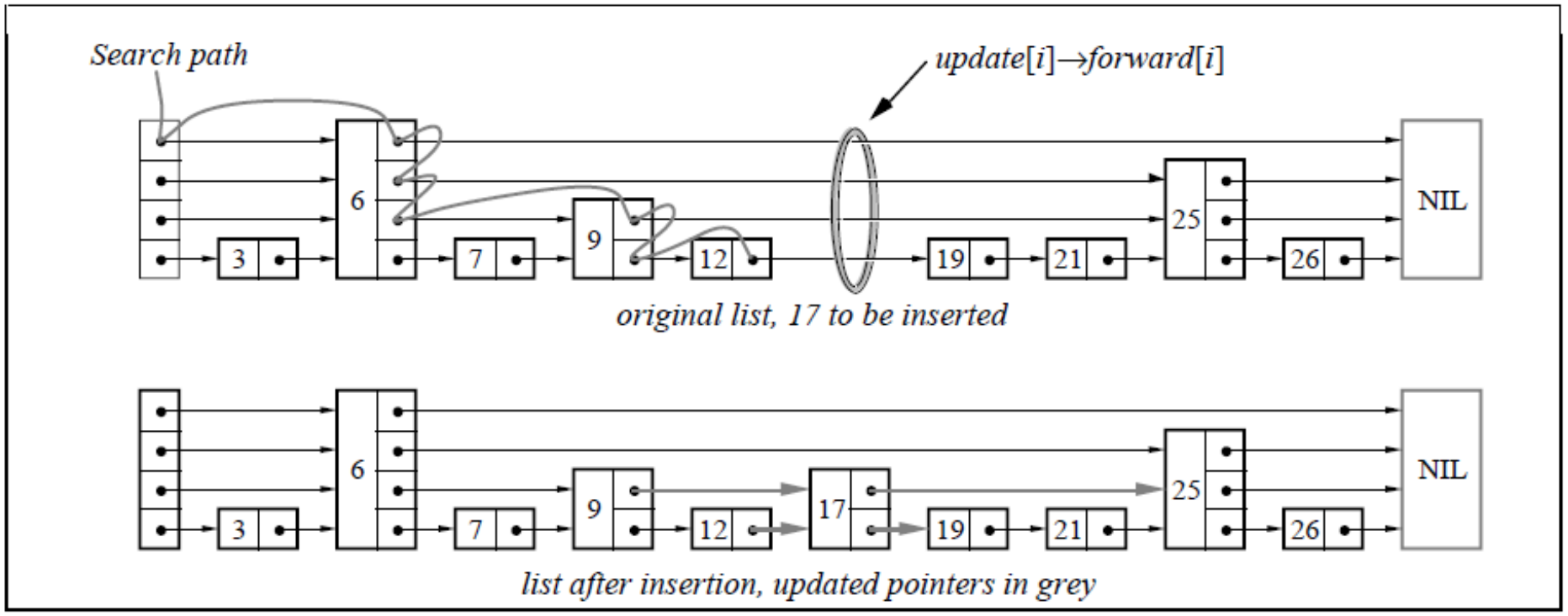


FIGURE 3 - Pictorial description of steps involved in performing an insertion

This happens when $L = \log_{1/p} n$. Since we will be referring frequently to this formula, we will use $L(n)$ to denote $\log_{1/p} n$.

Source: William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees.

Formal Proof –cont.

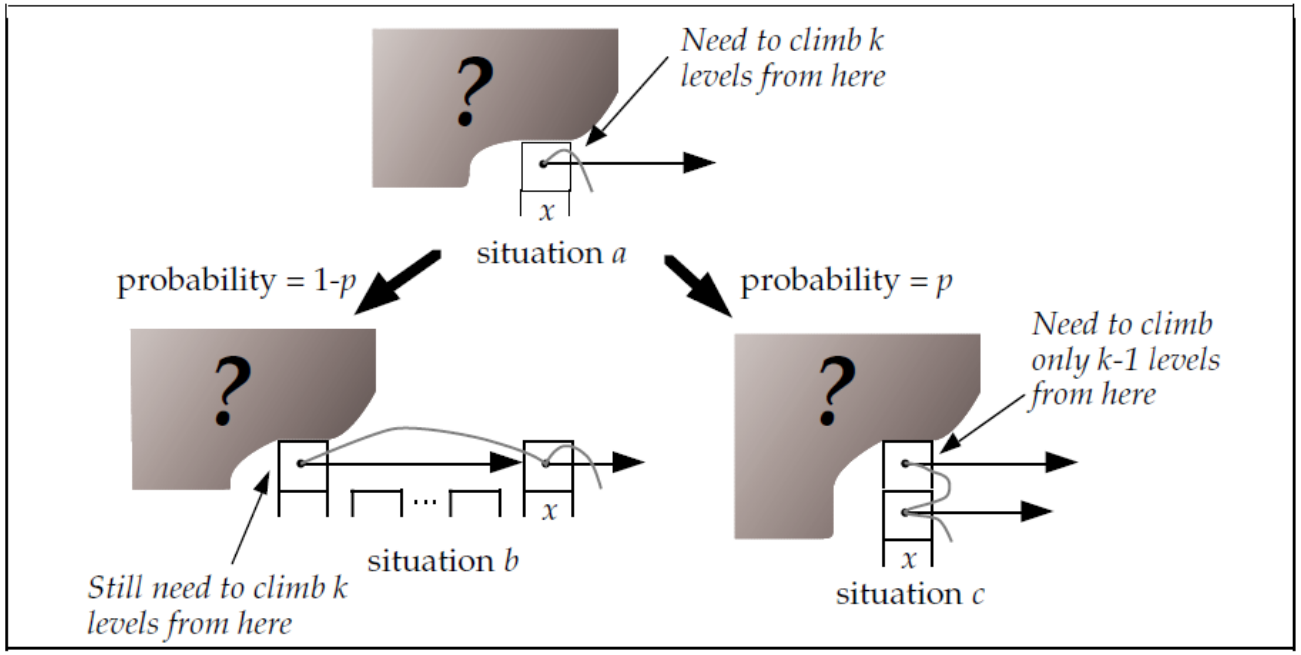


FIGURE 6 - Possible situations in backwards traversal of the search path

Let $C(k)$ = the expected cost (i.e, length) of a search path that climbs up k levels in an infinite list:

$C(0) = 0$

$C(k) = (1-p)$ (cost in situation b) + p (cost in situation c)

By substituting and simplifying, we get:

$C(k) = (1-p)\ (1 + C(k)) + p\ (1 + C(k-1))$

$C(k) = 1/p + C(k-1)$

$C(k) = k/p$

Our assumption that the list is infinite is a pessimistic assumption. When we bump into the header in our backwards climb, we simply climb up it, without performing any leftward movements. This gives us an upper bound of $(L(n)-1)/p$ on the expected length of the path that climbs from level 1 to level $L(n)$ in a list of n elements.

We use this analysis go up to level $L(n)$ and use a different analysis technique for the rest of the journey. The number of leftward movements remaining is bounded by the number of elements of level $L(n)$ or higher in the entire list, which has an expected value of $1/p$.

We also move upwards from level $L(n)$ to the maximum level in the list. The probability that the maximum level of the list is a greater than k is equal to $1-(1-p^k)^n$, which is at most np^k. We can calculate the expected maximum level is at most $L(n) + 1/(1-p)$. Putting our results together, we find

Total expected cost to climb out of a list of n elements
$\leq L(n)/p + 1/(1-p)$

which is $O(\log n)$.

Source: William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees.