任务序列：C_{wc}为任务序列中最差代价　上界$O(n * C_{wc})$
动态表：平摊代价为常数，表空间满则扩容两倍

载入因子：$\alpha = \frac{num}{size}$（若 num，size 均为 1 则 $\alpha = 1$，$\frac{1}{2} < \alpha \leq 1$）

基本插入：将一个数据插入表中（第 i 次插入实际代价C_i）

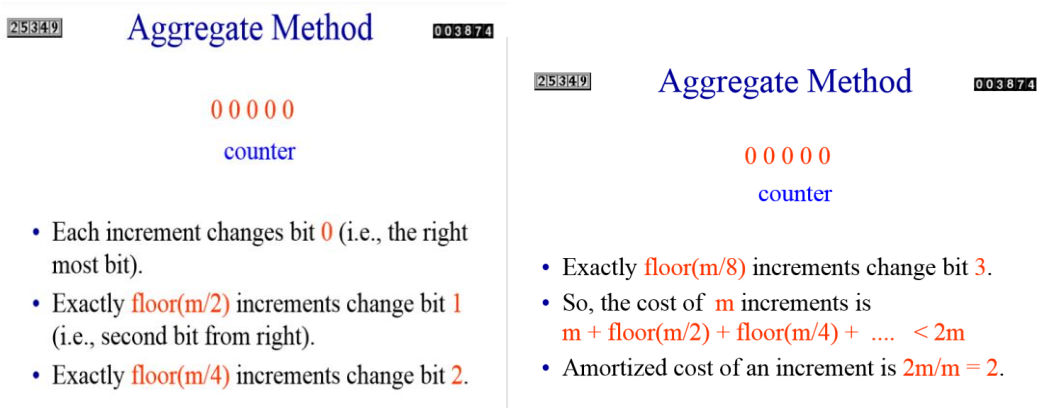
平摊分析 Amortize Complexity：

在平摊分析中，执行一系列数据结构操作所需要的时间是通过对执行的所有操作求平均而得出的。平摊分析可用来证明在一系列操作中，即使单一的操作具有较大的代价，通过对所有操作求平均后，平均代价还是很小的。平摊分析与平均情况分析的不同之处在于它不牵涉到概率。这种分析保证了在最坏情况下每个操作具有平均性能。

1. 聚类分析：n 个操作最坏总时间 T(n)，平均成本 T(n)/n（相同代价）

   保持载入因子$\frac{1}{2} \leq \alpha = \frac{表中元素}{表大小} < 1$，不满足条件时相应扩展/删除（申请新空间）

   聚集分析能够做到的就是求出一个较小的总代价，然后平摊到每一个操作上。对于单独的一个操作，是不能够确定其平摊代价是多少的。而接下来的两种方法可以做到对每一个特定的操作都可以求出其平摊代价。



2. 记账方法：决定每个操作成本，记录与实际代价差值，在平摊代价低于实际代价时补偿（存款不小于 0）
   当有一种以上的操作时（插入删除 够/不够），每种操作都可有一个不同的平摊代价（可以相同）。记账方法对操作序列中的某些操作先多收取平摊费用，将多收的部分作为对数据结构中的特定对象上的存款存起来。在该序列中稍后要用到这些存款以补偿那些对它们收费少于其实际代价的操作。

Accounting Method

- Guess that the amortized cost of an increment is 2.
- Now show that P(m) – P(0) >= 0 for all m.
- 1st increment:
  - one unit of amortized cost is used to pay for the change in bit 0 from 0 to 1.
  - the other unit remains as a credit on bit 0 and is used later to pay for the time when bit 0 changes from 1 to 0.

bits 0 0 0 0 0 → 0 0 0 0 1

credits 0 0 0 0 0 → 0 0 0 0 1

Accounting Method

- P(m) – P(0) = Σ(amortizedCost(i) –actualCost(i))
  = amount by which the first m increments have been over charged
  = number of credits
  = number of 1s
  >= 0

3. 势能方法：将总体表示为“势“，需要时释放支付操作代价（与整体数据结构联系）

插入时$\frac{1}{2} \le \alpha < 1$，删除时$\frac{1}{4} \le \alpha < \frac{1}{2}$

势能方法并不是把存款作为某个特定对象的存款，而是将存款作为数据结构整体的势能来维护。在需要的时候释放出来，支付操作代价，不需要的时候就存储起来。如果势能始终非负，那么总的平摊代价就是总的操作代价的一个上界。

Potential Method

- Guess a suitable potential function for which P(n) – P(0) >= 0 for all n.
- Derive amortized cost of ith operation using
  ΔP = P(i) – P(i–1)
  = amortized cost – actual cost
- amortized cost = actual cost + ΔP

Potential Method

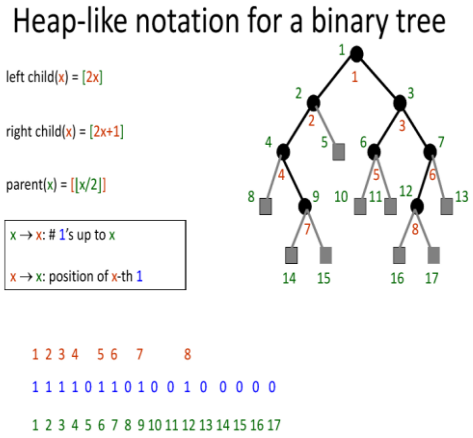
- Guess P(i) = number of 1s in counter after ith increment.
- P(i) >= 0 and P(0) = 0.
- Let q = # of 1s at right end of counter just before ith increment (01001111 => q = 4).
- Actual cost of ith increment is 1+q.
- ΔP = P(i) – P(i – 1) = 1 – q (0100111 => 0101000)
- amortized cost = actual cost + ΔP
  = 1+q + (1 – q) = 2

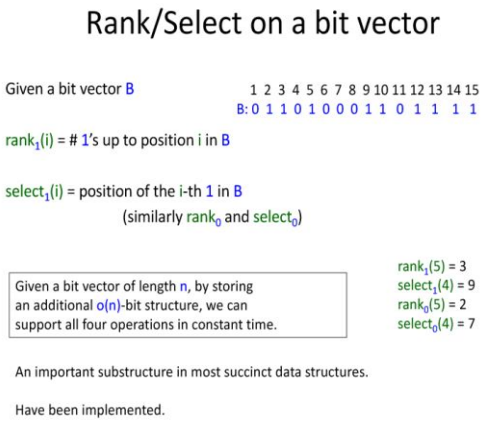
Asymptotic Analysis 渐近分析：我们用输入的大小来评估算法的性能

简洁数据结构：

二叉树标准表示：Node:{left，value，right}，使用 bits 而不使用系统指针O(2nlgn)

$\text{rank}_1(n)$：节点->B　　$\text{select}_1(n)$:B->节点

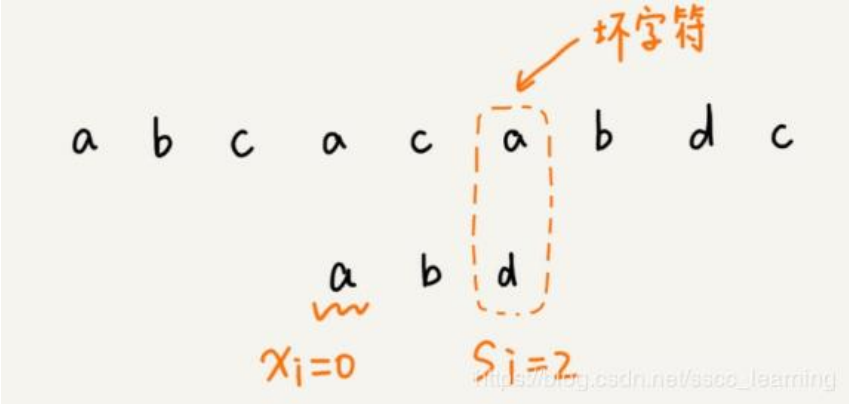




字符串匹配 BM：

坏字符规则：

1. 按照模式串下标从大到小倒着匹配
2. 当发生不匹配时，把坏字符对应的模式串中的字符下标记作 s
3. 坏字符在模式串中存在，把这个坏字符在模式串中的下标记作 xi；坏字符在模式串中不存在，把 xi 记作 -1
4. 模式串往后移动的位数就等于 s-xi。下标都是字符在模式串的下标。
5. 要说明的是，放坏字符在模式串中多次出现时，计算 xi 选择最靠后的那个，避免滑动过多从而错过匹配。



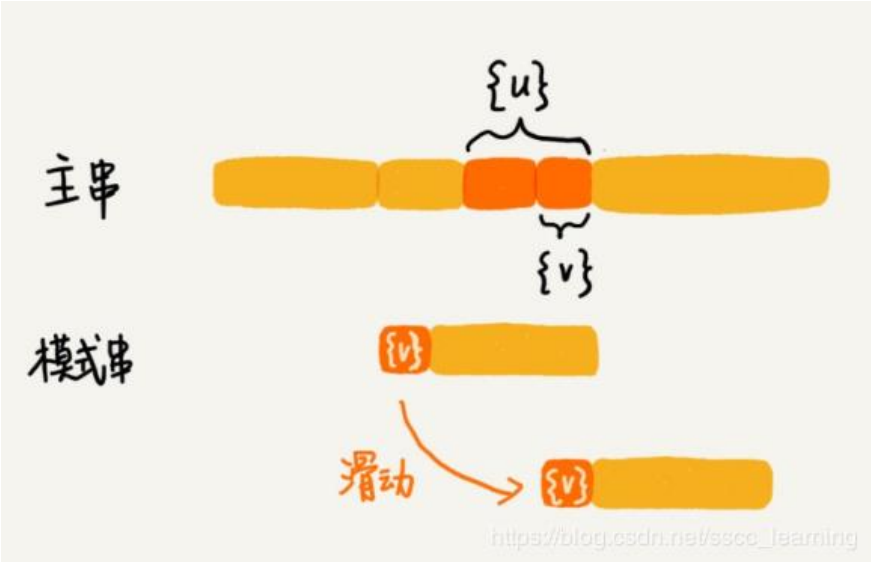
但是只是用坏字符规则还不够，比如，主串 aaaaaaaaaaaaaaa，模式串 baaa，si=0 xi=3 si-xi=-3，为负值，出现倒退情况。

好后缀规则：

规则一（坏字符首部与模式串相同），规则二（首部与模式串不同）

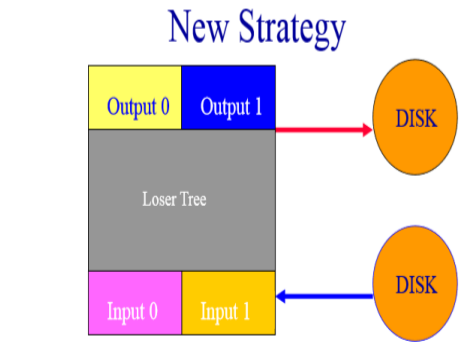
1. 当遇到坏字符时，完成整串匹配，已经匹配的好后缀记做{u}。
2. {u}在模式串中查找，如果找到相同子项，记做{u*}，将{u*}与主串中的{u}对齐。
3. 如果在模式串中没有找到另一个{u}，则下一步要查看好后缀的后缀子串，是否存在跟模式串的前缀子串匹配的。

4. 从好后缀的后缀子串中找到能与之匹配的模式串中的最长的前缀子串，记为{v}，将模式串滑动使后缀子串与{v}匹配的位置。
5. 假如后缀子串与前缀子串没有找到匹配，则将模式串移动到主串中 {u} 的后面。

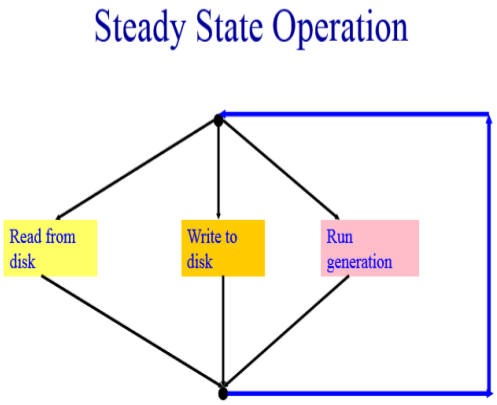


外部排序：

采用适当的内部排序方法对输入文件的每个片段进行排序，将排好序的片段（成为归并段）写到外部存储器中（通常由一个可用的磁盘作为临时缓冲区），这样临时缓冲区中的每个归并段的内容是有序的。



- Use 2 input and 2 output buffers.
- Rest of memory is used for a min loser tree.
- Actually, 3 buffers adequate.



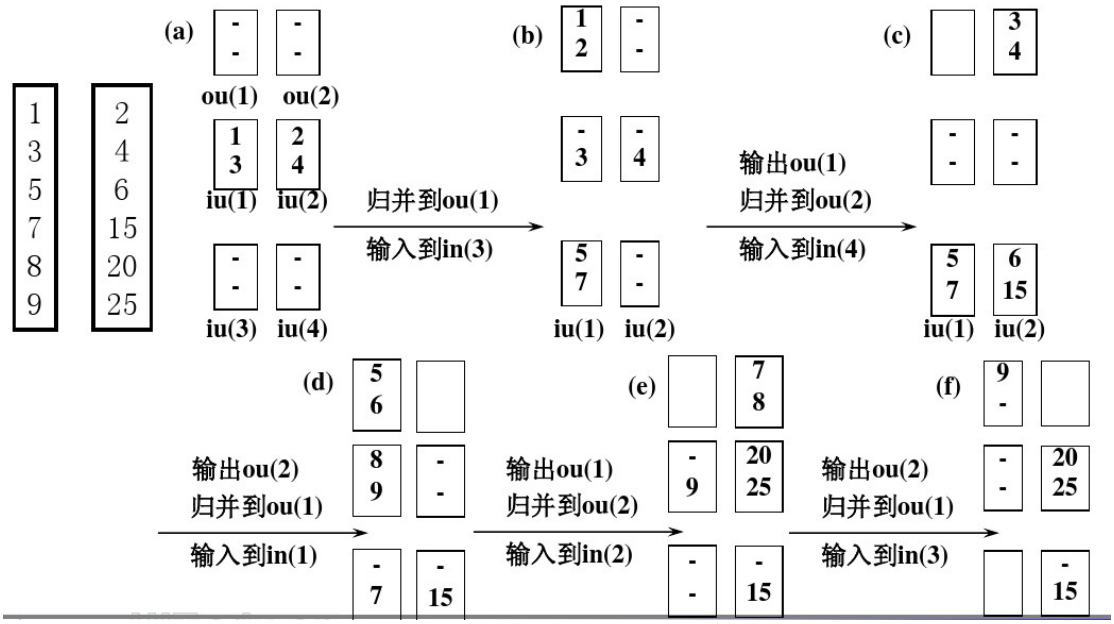
- Synchronization is done when the active input buffer gets empty (the active output buffer will be full at this time).

缓冲区策略：将数据加入最近序号最小的路

1. 建立（k 输入缓存，并读入数据，设置 activeOutputBuffer=0）
2. 记录未用缓冲池
3. 将输入缓冲归并到输出缓冲直到输出缓冲满/结束字符，输入空闲则继续读入

2）并行操作的缓冲区处理——使输入、输出和 CPU 处理尽可能重叠

对k个归并段进行 k 路归并至少需要k个输入和1个输出缓冲区，要使输入、输出和归并同时进行，k+1个缓冲区是不够的，需要2k个输入缓冲区实现并行操作。



Merge k Runs

```
repeat
    kWayMerge;
    wait for input/output to complete;
    add new input buffer (if any) to queue for its run;
    determine run that will exhaust first;
    if (there is more input from this run)
       initiate read of next block for this run;
    initiate write of active output buffer;
    activeOutputBuffer = 1 – activeOutputBuffer;
until end-of-run key merged;
```

问题：输入空间不足，无文件可读

Initializing For Next k-way Merge

```
Change
if (there is more input from this run)
      initiate read of next block for this run;
to
if (there is more input from this run)
      initiate read of next block for this run;
else
      initiate read of a block for the next k-way merge;
```

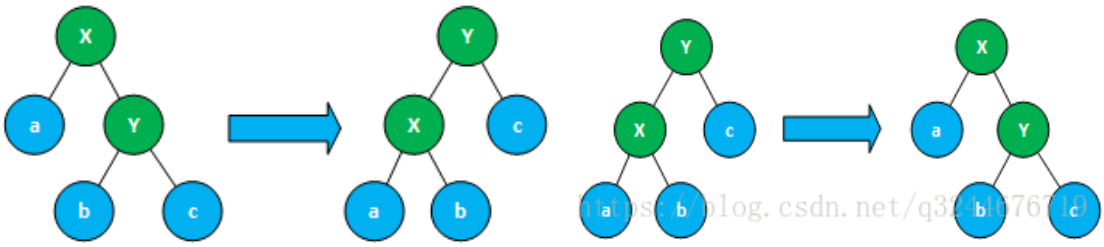
改进：空闲时从其它路中获取数据

红黑树：

1. 平衡二叉树
2. 外部节点及与其相邻的边为黑
3. 没有连续的红色节点/边
4. 根到每个外部节点路径上的黑色顶点/边数目相等

定理：n 个节点的红黑树高度$\log_2(n+1) \le H \le 2\log_2(n+1)$，删除红点/边形成坍塌树树高均等（节点数$n \ge 2^{\frac{h'}{2}} - 1 \ge 2^{\frac{h}{4}} - 1$）

左小右大，正常插入并判断是否平衡，否则左/右旋并更改颜色（删除同理）



左旋　　右旋

字节搜索树：

1. 非空，二叉树，所有子树均为字节搜索树
2. 根为一比特位数
3. K 层中位于左子树的比特位数第 K 位是 0，位于右子树的第 K 位是 1

每步复杂度 O(key 中比特位数)，key 比较复杂度 O(树高)

二叉树：

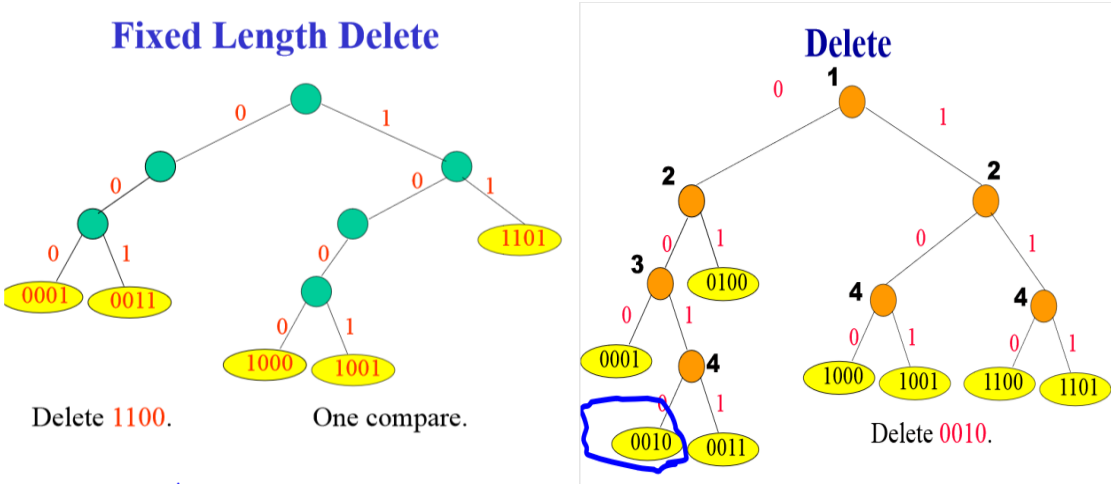
1. Branch（树枝节点，用于存放左右指针），Element（元素节点，存放数据无子节点）
2. 当无兄弟节点时，元素节点可直接连至父域

插入时增加树枝节点，无需挪动其它节点

二叉树搜索树 Compressed：

1. 无度为 1 的节点，
2. 树枝节点增加 bit#位用于指示查找方向

插入时增加树枝节点并挪动兄弟节点



二叉树　　二叉搜索树

最大最小堆：

min 层小于所有子节点，max 层大于所有子节点

```
template <class Type>
void MinMaxHeap<Type>::Insert(const Element<Type>&x ) {
  if (n == MaxSize ) { MinMaxFull( ); return;}
  n++;
  int p = n/2;
  if (!p) {h[1] = x; return;}
  switch (level(p)) {
    case MIN:
      if (x.key < h[p].key) {
        h[n]=h[p];
        VerifyMin(p, x);
      }
      else VerifyMax(n, x);
      break;
    case MAX:
      if (x.key > h[p].key) {
        h[n]=h[p];
        VerifyMax(p, x);
      }
      else VerifyMin(n, x);
  }
}
```

Insert a new node x into a min-max heap with no root:

(1) empty heap: x is the new root;

(2) at least one child of root:

find node k with the smallest data;

(a) x.key≤h[k].key。

x is the new root;

(b) x.key > h[k].key and k is a child of root:

replace k with x;

k is the new root.

(c) x.key > h[k].key and k is a grandchild of root:

k is the new root

if x.key > h[p].key, exchange x and h[p]

insert x into (sub)min_max heap rooted with k

最大最小堆插入

双堆 Deap：

1. 根为空的完全二叉树
2. 左子树最小堆，右子树最大堆
3. 左子树节点不大于其对应右子树节点

删除时检查是否为完全二叉树，记录不符合节点并整理左右子树，腾出空位插入

(1) **Deap::MaxHeap(int p)**

for p > 1, if $2^{\lfloor \log_2 p \rfloor} + 2^{\lfloor \log_2 p \rfloor - 1} \le p < 2^{\lceil \log_2 p \rceil}$, p is in the max-heap

(2) **Deap::MinPartner(int p)**

P is in max-heap and P is the last one in Deap:

$p - 2^{\lfloor \log_2 p \rfloor - 1}$

(3) **Deap::MaxPartner(int p)**

P in min-heap and P is the last one in Deap:

$(p + 2^{\lfloor \log_2 p \rfloor - 1}) / 2$