PS6

221300066 季千🄵

2024 年 6 月 12 日

1 Problem 1

(a) Suppose an ideal random hash function h has been picked. The exact expected number of collisions, as a function of n and m, is

$$\binom{n}{2} \times \frac{1}{m} = \frac{n(n-1)}{2m}$$

This is because each pair of items has a probability of $\frac{1}{m}$ to collide, and there are $\binom{n}{2}$ such pairs.

(b) The exact probability that a random hash function is perfect is

$$p = \frac{m!}{m^n(m-n)!}$$

This is because there are $m!$ ways to assign n items to m slots without collisions, and there are m^n ways to assign n items to m slots in total. The denominator must be greater than zero, so this probability is only defined when $m \geq n$.

(c) The exact expected number of different random hash functions I have to test before finding a perfect hash function is

$$\frac{1}{p}$$

where p is the probability from subproblem (b).

This is because the number of trials until the first success follows a geometric distribution with parameter p.

(d) The exact probability that none of the first N random hash functions is perfect is

$$(1-p)^N$$

where p is the probability from subproblem (b).

This is because the probability of failure in each trial is 1-p, and the trials are independent.

(e) To find a perfect hash function with high probability (that is, with probability at least $1 - 1/\mathrm{n}$), I have to test at least

$$\log_{(1-p)}(1-\frac{1}{n})$$

ideal random hash functions, where p is the probability from part (b).

This is because I want to solve the inequality

$$(1-p)^N \leq \frac{1}{n}$$

for N. Taking logarithms on both sides gives

$$N \geq \log_{(1-p)}(1-\frac{1}{n})$$

2 Problem 2

伪代码实现如下（假设计数器的位数为 k，数组的下标从 0 到 k-1，其中 $A[0]$ 是最低位，$A[k-1]$ 是最高位。）

```
# 初始化一个k位的计数器
def initialize(k):
  # 创建一个长度为k的数组A，初始值全为0
  A = [0] * k
  # 创建一个变量v，初始值为0，用来记录计数器的值
  v = 0
  # 返回数组A和变量v作为计数器
  return A, v

# 增加计数器的值
def increment(A, v):
```

```
    # 将v加一
    v = v + 1
    # 将v转换为二进制字符串，并用0补齐k位
    s = bin(v)[2:].zfill(k)
    # 将s中的每个字符赋值给A中的对应元素
    for i in range(0, k):
        A[i] = int(s[i])
    # 返回更新后的A和v
    return A, v

# 重置计数器的值
def reset(A, v):
    # 将v赋值为0
    v = 0
    # 将A中的所有元素赋值为0
    for i in range(0, k):
        A[i] = 0
    # 返回更新后的A和v
    return A, v
```

下面证明这个实现的时间复杂度是 O(n)，分析每种操作的平摊代价（在一系列操作中，每个操作的平均代价）：

每次增加操作，都会改变至少一个位，最多改变 k 个位。如果改变了 i 个位，那么这次操作的代价就是 i。每当一个位从 0 变为 1 时，它消耗了一次电；当它从 1 变为 0 时，它释放了一次电。因此，每个位的电量总是平衡的。

假设进行了 n 次增加操作，把每次改变位看作是一次成功的事件，那么每个位在 n 次操作中改变的次数显然服从二项分布。例如，A[0] 在 n 次操作中改变的概率是 $\frac{1}{2}$，因为它每次都会改变；A[1] 在 n 次操作中改变的概率是 $\frac{1}{4}$，因为它每两次才会改变一次；以此类推，A[i] 在 n 次操作中改变的概率是 $\frac{1}{2^{i+1}}$。因此，总共改变的次数期望是：

$$E[X] = \sum_{i=0}^{k-1} n \cdot \frac{1}{2^{i+1}} = n \cdot \sum_{i=0}^{k-1} \frac{1}{2^{i+1}} = n \cdot (1 - \frac{1}{2^k}) < n$$

故 n 次增加操作的总代价小于 n，平摊代价小于 1。

每次重置操作，都会将所有的位赋值为 0。这个操作的代价是 k。但是在进

行重置操作之前，必然有一次增加操作使得计数器达到了最大值 $2^k - 1$。所以在重置之前，所有的位都被充满了电。进行重置操作时，所有的电都被释放了。因此重置操作并没有增加额外的电量，而是利用了之前累积的电量。假设进行了 n 次操作，其中有 m 次重置操作，那么总共改变位次数为

$$E[X] \leq n - m + m \cdot k = n + m \cdot (k-1) \leq n + n \cdot (k-1) = n \cdot k$$

所以，n 次操作的总代价小于 $n \cdot k$，平摊代价小于 k。
综上所述：任意 n 次操作的平摊代价都是 O(1)，因此总时间复杂度是 O(n)。

3 Problem 3

使用一个双向链表来存储动态多重集合 S 的元素。每个节点存储一个元素的值和出现的次数，以及指向前驱和后继节点的指针。

```
// 定义一个节点结构体
struct node {
  // 存储元素的值
  int value;
  // 存储元素的出现次数
  int count;
  // 存储指向前驱和后继节点的指针
  struct node *prev, *next;
};

// 创建一个新的多重集，返回指向该多重集的指针
function NewSet():
  // 创建一个新的多重集结构体
  set = new set()
  // 赋值头节点和尾节点为空
  set.head = set.tail = NULL
  // 赋值元素总数为0
  set.size = 0
  // 返回多重集指针
  return set
```

```
// 向多重集S中插入元素x，返回插入后的多重集指针
function Insert(S, x):
  // 如果S为空，直接返回S
  if S == NULL:
    return S
  // 如果S为空集，创建一个新的节点，赋值为x，次数为1，前后指针为空，并将其设
  if S.head == NULL and S.tail == NULL:
    node = new node()
    node.value = x
    node.count = 1
    node.prev = node.next = NULL
    S.head = S.tail = node
    S.size = 1
    return S
  // 如果S不为空集，从头节点开始遍历链表，寻找是否有值等于x的节点
  curr = S.head
  while curr != NULL:
    // 如果找到了值等于x的节点，将其次数加一，更新S的大小加一，返回S
    if curr.value == x:
      curr.count = curr.count + 1
      S.size = S.size + 1
      return S
    // 如果没有找到，继续遍历下一个节点
    else:
      curr = curr.next

  // 如果遍历完整个链表都没有找到值等于x的节点，创建一个新的节点，赋值为x，次
  // 创建一个新的节点，赋值为x，次数为1
  node = new node()
  node.value = x
  node.count = 1
```

```
// 如果x小于头节点的值，将新节点设为头节点，并将原头节点设为新节点的后继节
if x < S.head.value:
  node.prev = NULL
  node.next = S.head
  S.head.prev = node
  S.head = node
  S.size = S.size + 1
  return S

// 如果x大于尾节点的值，将新节点设为尾节点，并将原尾节点设为新节点的前驱节
if x > S.tail.value:
  node.prev = S.tail
  node.next = NULL
  S.tail.next = node
  S.tail = node
  S.size = S.size + 1
  return S

// 如果x在头尾节点之间，从头节点开始遍历链表，寻找第一个大于x的节点，并将新

// 记录当前遍历到的节点为curr（初始为头节点），记录其前驱节点为prev（初始为
curr = S.head
prev = NULL

// 循环遍历链表，直到找到第一个大于x的节点或到达尾节点
while curr != NULL and curr.value <= x:
  // 更新prev为curr，更新curr为其后继节点
  prev = curr
  curr = curr.next

// 将新节点插入到prev和curr之间，更新它们的前后指针，更新S的大小加一，返回
node.prev = prev
node.next = curr
```

```
  prev.next = node
  curr.prev = node
  S.size = S.size + 1
  return S

// 删除多重集S中最大的 |S|/2 个元素，返回删除后的多重集指针
function DeleteLargeHalf(S):
  // 如果S为空或为空集，直接返回S
  if S == NULL or (S.head == NULL and S.tail == NULL):
    return S
  // 如果S只有一个元素，删除该元素，并将头尾节点设为空，更新S的大小为0，返回
  if S.head == S.tail:
    delete S.head
    S.head = S.tail = NULL
    S.size = 0
    return S

  // 如果S有多个元素，从尾节点开始遍历链表，记录已经删除的元素个数为count（初
  count = 0
  curr = S.tail

  // 循环遍历链表，直到删除的元素个数达到 |S|/2 或到达头节点
  while count < ceil(S.size / 2) and curr != NULL:
    // 如果当前节点的次数小于等于 |S|/2 减去count，说明该节点可以完全删除，更
    if curr.count <= ceil(S.size / 2) - count:
      count = count + curr.count
      prev = curr.prev
      delete curr
      curr = prev

    // 如果当前节点的次数大于 |S|/2 减去count，说明该节点只能部分删除，更新该
    else:
      curr.count = curr.count - (ceil(S.size / 2) - count)
```

```
      count = ceil(S.size / 2)
      break

  // 如果遍历完整个链表都没有达到 |S|/2 个元素，说明最后一个元素出现了多次，
  if count < ceil(S.size / 2):
    curr.count = 1
    S.head = S.tail = curr
    S.size = 1
    return S

  // 如果遍历到了头节点或者达到了 |S|/2 个元素，将当前节点设为新的尾节点，并
  else:
    S.tail = curr
    curr.next = NULL
    S.size = S.size - count
    return S

// 打印多重集S中的所有元素（按升序排列）
function Print(S):
  // 如果S为空或为空集，打印空行并返回
  if S == NULL or (S.head == NULL and S.tail == NULL):
    print("\n")
    return

  // 如果S不为空集，从头节点开始遍历链表，并打印每个节点的值和次数（用冒号分
  else:
    curr = S.head
    while curr != NULL:
      print(curr.value, ":", curr.count, "␣")
      curr = curr.next

    print("\n")
    return
```

插入操作只需要在链表的末尾添加一个节点，时间复杂度为 O(1)。删除最大一半操作需要先找到链表中的中位数，然后用快速排序的划分算法将链表划分为两部分，最后删除较大的一部分，时间复杂度为 O(|S|)。打印操作只需要遍历链表中的所有节点，时间复杂度为 O(|S|)。

对于 m 次插入和删除最大一半操作的总时间复杂度为 O(m)，使用势能方法进行摊还分析。定义势能函数 Φ(Di) 为链表中节点的个数，其中 Di 表示第 i 次操作后的数据结构。显然可以证明，对于任意的 i，有：

- 如果第 i 次操作是插入，那么实际代价 ci = 1，势能变化 ΔΦi = 1，摊还代价 $hat\{ci\} = ci + i = 2$。
- 如果第 i 次操作是删除最大一半，那么实际代价 ci = |S|，势能变化 $i = -|S|/2 \leq -|S|/2$，摊还代价 $hat\{ci\} = ci + i \leq |S| - |S|/2 = |S|/2$。

因此，m 次操作的总摊还代价为：

$$\sum_{i=1}^{m} \hat{c_i} \leq \sum_{i=1}^{m} (c_i - \Phi(D_i) + \Phi(D_0)) = \sum_{i=1}^{m} c_i - \Phi(D_m) + \Phi(D_0) \leq m - 0 + n = O(m)$$

其中 n 是初始时链表中节点的个数。这说明了任意 m 次插入和删除最大一半操作的总时间复杂度为 O(m)。

4 Problem 4

(a) Using the following formula to determine which bit flips, given the index i:

$$\text{bit number} = \lfloor \log_2(i \oplus (i-1)) \rfloor$$

where $\oplus$ denotes the bitwise XOR operation and $\lfloor x \rfloor$ denotes the floor function that rounds down x to the nearest integer.

This formula works because the XOR of two binary numbers gives a 1 in every position where they differ, and a 0 otherwise. Therefore, the position of the most significant 1 in the XOR result is the same as the position of the bit that flips. The logarithm base 2 gives us the index of that position, and the floor function ensures that we get an integer value.

For instance, suppose we want to find which bit flips when going from the 5-th integer to the 6-th integer in the binary reflected Gray code for $k = 3$. The 5-th integer is 0110 and the 6-th integer is 0100 in binary. The

XOR of these two numbers is 0010, which has a 1 in the second position from the right. The logarithm base 2 of 0010 is 1, and the floor function does not change it. Therefore, the bit number that flips is 1.

(b) Using the following recursive algorithm:

The base case is when $k = 1$, in which case we return the sequence $\langle 0, 1 \rangle$.

For larger values of k, we first compute the binary reflected Gray code for $k - 1$, and store it in an array A. Then we create a new array B of size 2^k, and copy the elements of A to the first half of B. Next, we reverse the elements of A, and add 2^{k-1} to each of them, and copy them to the second half of B. Finally, we return B as the result.

The pseudocode for this algorithm is given below:

```
function gray_code(k):
    if k == 1:
        return [0, 1]
    else:
        A = gray_code(k - 1)
        B = new array of size 2^k
        for i from 0 to 2^(k-1) - 1:
            B[i] = A[i]
            B[2^k - i - 1] = A[i] + 2^(k-1)
        return B
```

The time complexity of this algorithm is $\Theta(2^k)$, because each recursive call takes linear time to copy and reverse the elements of the previous array, and there are $\log_2(2^k) = k$ recursive calls in total.

5 Problem 5

(a) The worst-case running time is $O(n^2)$. This is because, without path compression, the height of the trees can be as large as n, and each Find operation can take O(n) time in the worst case. Since we perform n Find operations in the loop, the total time is $O(n^2)$.

(b) With path compression, the height of the trees is reduced to $O(logn)$ after each Find operation. Therefore, each Find operation takes $O(logn)$ time in the worst case. Moreover, using a simple amortized analysis below, I can show that the total time for n Find operations with path compression is O(n).

Proof:

My idea is to assign a potential function to each node that represents how much work it can save in the future. Initially, each node has a potential of 0. When we perform a Find operation on a node, we increase its potential by 1 for each ancestor that it passes through until it reaches the root. This represents the fact that each ancestor is moved one level closer to the root by path compression, and thus will save one unit of work in the future. The total potential of all nodes is increased by at most O(log n) for each Find operation.

Now, let us consider the actual work done by a Find operation on a node. We can charge this work to the potential of the node and its ancestors. For each ancestor that we visit, we do one unit of work and decrease its potential by 1. Therefore, the net work done by a Find operation is zero.

Hence, the total work done by n Find operations is bounded by the total increase in potential, which is O(n log n). Since we also have to initialize and fill the label array, which takes O(n) time, the overall running time is O(n + n log n) = O(n).

6 Problem 6

使用平衡二叉树来存储字符串进行维护。每个节点存储一个字符，以及它的左子树和右子树的高度和长度。
每个操作的伪代码如下

```
// 定义一个节点结构体
struct node {
  // 存储字符
  char c;
  // 存储左子树和右子树的高度
```

```
  int height_left, height_right;
  // 存储左子树和右子树的长度
  int length_left, length_right;
  // 存储指向左子节点和右子节点的指针
  struct node *left, *right;
};

// 创建一个新的字符串，只包含字符c，返回指向该字符串的指针
function NewString(c):
  // 创建一个新的节点
  node = new node()
  // 赋值字符c
  node.c = c
  // 赋值高度为1，长度为1
  node.height_left = node.height_right = node.length_left = node.length_righ
  // 赋值左右子节点为空
  node.left = node.right = NULL
  // 返回节点指针
  return node

// 连接两个字符串S和T，删除原来的S和T，返回指向新字符串ST的指针
function Concat(S, T):
  // 如果S或T为空，直接返回另一个字符串的指针
  if S == NULL:
    return T
  if T == NULL:
    return S
  // 创建一个新的节点，赋值字符为空格（或其他分隔符）
  node = new node()
  node.c = ' '
  // 赋值左右子节点为S和T
  node.left = S
  node.right = T
  // 赋值左右子树的高度为S和T的高度之和
```

```
  node.height_left = S.height_left + S.height_right
  node.height_right = T.height_left + T.height_right
  // 赋值左右子树的长度为S和T的长度之和
  node.length_left = S.length_left + S.length_right
  node.length_right = T.length_left + T.length_right
  // 返回节点指针
  return node
// 翻转一个字符串S，删除原来的S，返回指向新字符串S'的指针
function Reverse(S):
  // 如果S为空或只有一个字符，直接返回S的指针
  if S == NULL or (S.left == NULL and S.right == NULL):
    return S
  // 创建一个新的节点，赋值字符为S的字符（如果有）
  node = new node()
  node.c = S.c
  // 赋值左右子节点为翻转后的S的右子树和左子树（如果有）
  node.left = Reverse(S.right)
  node.right = Reverse(S.left)
  // 赋值左右子树的高度为S的右子树和左子树的高度（如果有）
  node.height_left = S.height_right
  node.height_right = S.height_left
  // 赋值左右子树的长度为S的右子树和左子树的长度（如果有）
  node.length_left = S.length_right
  node.length_right = S.length_left
  // 返回节点指针
  return node

// 查找字符串S中第k个字符，返回该字符或空（如果不存在）
function Lookup(S, k):
   // 如果S为空或k小于等于0，返回空
   if S == NULL or k <=0:
     return NULL
   // 如果k等于S的左子树长度加一，返回S的字符（如果有）
```

```
if k == S.length_left +1:
  return S.c
// 如果k小于等于S的左子树长度，递归查找S的左子树中第k个字符
if k <= S.length_left:
  return Lookup(S.left, k)
// 如果k大于S的左子树长度加一，递归查找S的右子树中第k–S.length_left–1个字
if k > S.length_left + 1:
  return Lookup(S.right, k - S.length_left - 1)
```

时间复杂度分析如下：
NewString 操作只涉及到创建一个新的节点，赋值一些常数，返回一个指针，因此它的时间复杂度是 O(1)。
Concat 操作只涉及到创建一个新的节点，赋值一些常数，连接两个子节点，返回一个指针，因此它的时间复杂度也是 O(1)。
Reverse 操作涉及到递归地翻转每个节点的左右子树，并交换它们的高度和长度，因此它的时间复杂度是 O(n)，其中 n 是字符串的长度。
Lookup 操作涉及到递归地查找每个节点的左右子树，并比较它们的长度，因此它的时间复杂度是 O(logn)，其中 n 是字符串的长度。