Adv Data Struct & Algorithm: Homework 4

Zimian Li

May 16, 2018

1. Consider a universe $\mathcal{U}$ of keys each of which is b bits long. You want to construct a hash table of size $m = 2^a$. To select your hash function h, you build a random binary matrix M and let $h_M(x) = M \times x$, with the caveat that addition is done modulo 2. Your set $\mathcal{H}$ of hash functions is the set of all possible binary matrices.

   (a) What should be the dimensions of matrix M?

   We can write the hash function as $h_M(x) = M \times (x_1, x_2, ..., x_b) = (y_1, y_2, ..., y_a)$. Obviously, matrix M should be $b \times a$.

   (b) How big is $\mathcal{H}$?

   Because the size of M is ba, and every element of it could be 0 or 1, therefore, the total number of all kinds of M is 2^{ba}. It's also the size of $\mathcal{H}$.

   (c) Show that $\mathcal{H}$ is universal.

   Proof: Suppose k, l be two arbitrary distinct keys from $\mathcal{U}$, and $k = (k_1, k_2, ..., k_b)$ and $l = (l_1, l_2, ..., l_b)$. And Pick a random hash function $h_M(x) = M \times x$ from $\mathcal{H}$.
   Therefore, $h(k) = ((\sum_{i=1}^{b} k_i M_{i1}) \mod 2, (\sum_{i=1}^{b} k_i M_{i2}) \mod 2, ..., (\sum_{i=1}^{b} k_i M_{ia}) \mod 2)$, and $h(l) = ((\sum_{i=1}^{b} l_i M_{i1}) \mod 2, (\sum_{i=1}^{b} l_i M_{i2}) \mod 2, ..., (\sum_{i=1}^{b} l_i M_{ia}) \mod 2)$. If $h(k) = h(l)$, then for an arbitrary j(j is in range 1 to a), $(\sum_{i=1}^{b} k_i M_{ij}) \mod 2 = (\sum_{i=1}^{b} l_i M_{ij}) \mod 2$. That means $(\sum_{i=1}^{b} k_i M_{ij})$ and $(\sum_{i=1}^{b} l_i M_{ij})$ have the same parity.
   Claim: $\Pr((\sum_{i=1}^{b} k_i M_{ij}) \text{ is even}) = \Pr((\sum_{i=1}^{b} k_i M_{ij}) \text{ is odd}) = 1/2$.
   Proof: k_i and M_{ij} are all 0 or 1, so there are 4 cases (with equal possibility) of $k_i M_{ij}$:
   1) $0 \cdot 0 = 0$
   2) $0 \cdot 1 = 0$
   3) $1 \cdot 0 = 0$
   4) $1 \cdot 1 = 1$
   Obviously, the parity of $(\sum_{i=1}^{b} k_i M_{ij})$ only depends on the number of $1 \cdot 1$. And the chance is equal for the number of $1 \cdot 1$ is even or odd. Therefore, $\Pr((\sum_{i=1}^{b} k_i M_{ij}) \text{ is even}) = \Pr((\sum_{i=1}^{b} k_i M_{ij}) \text{ is odd}) = 1/2$. It's also same for $(\sum_{i=1}^{b} l_i M_{ij})$.
   Hence, $\Pr((\sum_{i=1}^{b} k_i M_{ij}) \text{ and } (\sum_{i=1}^{b} l_i M_{ij}) \text{ have the same parity}) = \Pr((\sum_{i=1}^{b} k_i M_{ij}) \text{ is even and } (\sum_{i=1}^{b} l_i M_{ij}) \text{ is even}) + \Pr((\sum_{i=1}^{b} k_i M_{ij}) \text{ is odd and } (\sum_{i=1}^{b} l_i M_{ij}) \text{ is odd}) =$

$1/2 \cdot 1/2 + 1/2 \cdot 1/2 = 1/2$.
Therefore, $\Pr(h(k) = h(l)) = \Pr((\sum_{i=1}^{b} k_i M_{i1})$ and $(\sum_{i=1}^{b} l_i M_{i1})$ have the same parity) $\cdot \Pr((\sum_{i=1}^{b} k_i M_{i2})$ and $(\sum_{i=1}^{b} l_i M_{i2})$ have the same parity) $\cdot$ ... $\cdot \Pr((\sum_{i=1}^{b} k_i M_{ia})$ and $(\sum_{i=1}^{b} l_i M_{ia})$ have the same parity) $= (\frac{1}{2})^a = (\frac{1}{2^a}) = \frac{1}{m}$. So finally, I can conclude $\mathcal{H}$ is universal.

2. Hashing a set of n keys into a table of size m requires $\Omega(n + m)$ time due to the fact that the table needs to be initialized. This is inefficient when $n = o(m)$. Describe a method that avoids initializing the hash table, possibly at the expense of some additional memory.

   If not initializing the hash table, we need another container to save these messages.
   1) If choosing a linear container such as an array, every time hashing a new key, I need to check this array if this key already exists, if so just update this key, if not add this key to the array. Thus, the time cost of hashing n keys is $O(n^2)$, I can choose it if $n^2 < n + m$.
   2) I can also choose to maintain a binary search tree for keys, then the time cost of hashing n keys is $O(n \log n)$, it's better than the linear container, I can choose it if $n \log n < n+m$.

3. Using comparison trees, prove lower bounds for the following problems. In each case, make sure to clearly indicate the number of leaves in the tree from which you compute the lower bound.

   (a) Given two sorted lists A and B of size n each, merge them into a single sorted list C.

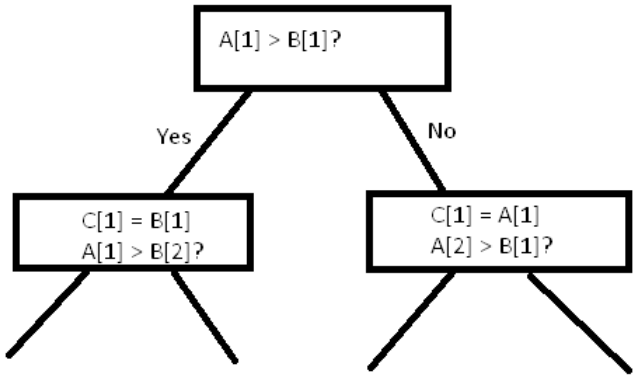

Figure 1: (ct1)

The image above is the initial part of the comparison tree, obviously, the number of leaves is the number of permutations of C. For the size of A and B are both n, the

number of permutations of C should be $\binom{2n}{n}$. Therefore, the height of this comparison tree is $\log(2 \cdot \binom{2n}{n} - 1)$.
$\binom{2n}{n} = \frac{2n!}{(n!)^2} = \frac{(2n)\cdot(2n-1)\cdot\ldots\cdot(n+1)}{n\cdot(n-1)\cdot\ldots\cdot 1} = \frac{2n}{n} \cdot \frac{2n-1}{n-1} \cdot \ldots \cdot \frac{n+1}{1} > (\frac{2n}{n})^n = 2^n$.
Therefore, $\log(2 \cdot \binom{2n}{n} - 1) \in \Omega(n)$. It's the lower bound of this problem.

(b) Given an array $A[1:n]$ of values from a totally ordered set, rearrange A so that the $n/2$ smallest values appear on the left and the $n/2$ largest appear on the right.

Consider the comparison tree, the leaves are all permutations that meeting the $n/2$ smallest values appear on the left and the $n/2$ largest appear on the right. The number is $(n/2)!^2$. So the height of the comparison tree is $\log(2\cdot(n/2)!^2 - 1) \in \Omega(n \log n)$. Hence, the lower bound is $\Omega(n \log n)$.

4. Consider a min-max heap H with n elements. H can be viewed a complete binary tree that uses the same array representation used in ordinary heaps.

(a) INSERT(H, n, x) inserts an element with key x into H. Describe how to implement INSERT in $O(\log \sqrt{n})$ time.

It's similar with the insertion of the ordinary heap, the difference is that we need to compare the new key with the parent of the parent of itself (grandparent!). And we need to calculate its depth ($\lfloor \log(i+1) \rfloor$) to decide it's in min-level or max-level. The time cost is $O(\frac{1}{2} \log n) = O(\log \sqrt{n})$.

Algorithm 1 INSERT

```
1: procedure INSERT(H, n, x)
2:     increase H with a new slot, n += 1
3:     H[n-1] = x, i = n-1
4:     if ⌊log(i + 1)⌋ is even then
5:         while i ≥ 0 and H[Parent(Parent(i))] > H[i] do
6:             exchange H[Parent(Parent(i))], H[i]
7:             i ← Parent(Parent(i))
8:         end while
9:     else
10:        while i ≥ 1 and H[Parent(Parent(i))] < H[i] do
11:            exchange H[Parent(Parent(i))], H[i]
12:            i ← Parent(Parent(i))
13:        end while
14:    end if
15: end procedure
```

(b) DELETEMIN(H, n) deletes and returns the element with minimum key from H. Describe an efficient implementation of DELETEMIN and analyze its running time.

It's also similar with the ordinary heap. The DELETEMIN operation is based on MIN-HEAPIFY, the differences is that we need to consider its all 4 grandchildren, the time cost is also $O(\log \sqrt{n})$ for only need to consider min-level.

Algorithm 2 MIN-HEAPIFY

```
1: procedure MIN-HEAPIFY(H, n, i)
2:     ll ← LEFT(LEFT(i))
3:     lr ← LEFT(RIGHT(i))
4:     rl ← RIGHT(LEFT(i))
5:     rr ← RIGHT(RIGHT(i))
6:     if ll < n and H[ll] < H[i] then
7:         smallest = ll
8:     else
9:         smallest = i
10:    end if
11:    if lr < n and H[lr] < H[smallest] then
12:        smallest = lr
13:    end if
14:    if rl < n and H[rl] < H[smallest] then
15:        smallest = rl
16:    end if
17:    if rr < n and H[rr] < H[smallest] then
18:        smallest = rr
19:    end if
20:    if smallest != i then
21:        exchange H[smallest], H[i]
22:        MIN-HEAPIFY(H, n, smallest)
23:    end if
24: end procedure
```

Algorithm 3 DELETEMIN

```
1: procedure DELETEMIN(H, n)
2:     min ← H[0]
3:     H[0] ← H[n-1]
4:     n -= 1
5:     MIN-HEAPIFY(H, n, 0)
6:     return min
7: end procedure
```

(c) DELETEMAX(H, n) deletes and returns the element with maximum key from H. Describe an efficient implmentation of DELETEMAX and analyze its running time.

It's similar with DELETEMIN, based on MAX-HEAPIFY. Another point is that the second level has 2 max nodes, we need to compare them to decide which one is the true max. The time complexity is $O(\log \sqrt{n})$.

Algorithm 4 MAX-HEAPIFY

```
1: procedure MAX-HEAPIFY(H, n, i)
2:     ll ← LEFT(LEFT(i))
3:     lr ← LEFT(RIGHT(i))
4:     rl ← RIGHT(LEFT(i))
5:     rr ← RIGHT(RIGHT(i))
6:     if ll < n and H[ll] > H[i] then
7:         largest = ll
8:     else
9:         largest = i
10:    end if
11:    if lr < n and H[lr] > H[largest] then
12:        largest = lr
13:    end if
14:    if rl < n and H[rl] > H[largest] then
15:        largest = rl
16:    end if
17:    if rr < n and H[rr] > H[largest] then
18:        largest = rr
19:    end if
20:    if largest != i then
21:        exchange H[largest], H[i]
22:        MAX-HEAPIFY(H, n, largest)
23:    end if
24: end procedure
```

Algorithm 5 DELETEMAX

```
1: procedure DELETEMAX(H, n)
2:     if H[1] > H[2] then
3:         max = H[1]
4:         max-index = 1
5:     else
6:         max = H[2]
7:         max-index = 2
8:     end if
9:     H[max-index] ← H[n-1]
10:    n -= 1
11:    MAX-HEAPIFY(H, n, max-index)
12:    return max
13: end procedure
```

(d) Given an array $A[1:n]$ of integers, with no particular structure, can you rearrange A into a min-max heap on $o(n \log n)$ time? Explain.

All I can do is like below. Its time complexity is $2 \cdot \frac{n}{4} \cdot O(\log \sqrt{n}) = O(\frac{n}{4} \log n) = O(n \log n)$. Consider a comparison tree, to build a min-max heap is based on comparison. The number of leaves is $n!$, thus the height is at least $\Omega(\log n!) \approx \Omega(n \log n)$. The

lower bound is $\Omega(n \log n)$, not possible to find a $o(n \log n)$ alorithm.

Algorithm 6 BUILD-MIN-MAX-HEAP

```
1: procedure BUILD-MIN-MAX-HEAP(A)
2:     for i ← ⌊length(A)/2⌋ do downto 1
3:         if ⌊log(i)⌋ is even then
4:             MIN-HEAPIFY(A, length(A), i)
5:         else
6:             MAX-HEAPIFY(A, length(A), i)
7:         end if
8:     end for
9: end procedure
```

5. A *rectangular* heap is a $p \times q$ matrix $\mathcal{R}$ such that the entries in every row and in every column appear in ascending order. Some of the entries may be non-existent in which case we assign them the value ∞. This means that $\mathcal{R}$ can store $n \leq pq$ actual values. Below is a rectangular heap storing the set $\{2, 3, 4, 5, 8, 9, 11, 12, 14, 15\}$.

2	4	8	15
3	9	11	∞
5	14	∞	∞
12	∞	∞	∞

   (a) How do you determine efficiently if the heap is empty (i.e., $n = 0$), and if the heap is full (i.e., $n = pq$)? Prove your claim.

   If $\mathcal{R}[0][0]$ is ∞, the heap is empty, if $\mathcal{R}[p-1][q-1]$ is not ∞, the heap is full.
   Proof: 1) Because in every row and in every column, the entries are all in ascending order, if $\mathcal{R}[0][0]$ is ∞, then row 1 and column 1 are all ∞. Another fact is that row 1 contains all smallest number of all columns and column 1 contains all smallest number of all rows, so row 2 to p and column 2 to p are all ∞, therefore, the heap is empty.
   2) Similar with the previous case, if $\mathcal{R}[p-1][q-1]$ is not ∞, row p and column q are all not ∞, and row p and column q contain the largest numbers of all columns and all rows, so all rows and columns don't contain ∞. Hence, the heap is full.

   (b) Describe and analyze efficient implementations of MIN, EXTRACTMIN, and INSERT. Your solutions should run in $O(max\{p, q\})$ time.

Algorithm 7 MIN

```
1: procedure MIN(R)
2:     return R[0][0]
3: end procedure
```

MIN is very simple, it only takes $O(1)$ time.

Algorithm 8 EXTRACTMIN

```
1: procedure EXTRACTMIN(R)
2:     min ← R[0][0]
3:     R[0][0] ← ∞
4:     i = 0, j = 0
5:     while i ¡ p and j ¡ q do
6:         if R[i+1][j] ≤ R[i][j+1] then
7:             exchange R[i+1][j], R[i][j]
8:             i += 1
9:         else
10:            exchange R[i][j+1], R[i][j]
11:            j += 1
12:        end if
13:        if R[i+1][j] == ∞ and R[i][j+1] == ∞ then
14:            break
15:        end if
16:    end while
17:    return min
18: end procedure
```

Time complexity is $O(O(max\{p, q\}))$.

Algorithm 9 INSERT

```
1: procedure INSERT(R, x)
2:     if R[p-1][q-1] == ∞ then
3:         R[p-1][q-1] = x
4:     else
5:         return error "Heap is full!"
6:     end if
7:     i=p-1, j=q-1
8:     while i ≥ 1 and j ≥ 1 do
9:         if x ≥ R[i-1][j] and x ≥ R[i][j-1] then
10:            break
11:        end if
12:        if (x ≥ R[i-1][j] and x < R[i][j-1]) or (x < R[i-1][j] and x < R[i][j-1] and R[i-1][j] ≤ R[i][j-1]) then
13:            exchange x, R[i][j-1]
14:            j -= 1
15:        end if
16:        if (x < R[i-1][j] and x ≥ R[i][j-1]) or (x < R[i-1][j] and x < R[i][j-1] and R[i-1][j] > R[i][j-1]) then
17:            exchange x, R[i-1][j]
18:            i -= 1
19:        end if
20:    end while
21: end procedure
```

This algorithm is starting from the last position and swapping with the biggest neighbor until a suitable position. The time complexity is $O(p + q)$.

(c) Let $n = m^2$, i.e., n is a perfect square (e.g., $n = 16100256$, etc.) Using *only* yoour rectangular heap opeartions (and no other sorting method) show how to sort an array of n values on $o(m^4)$ time.

Algorithm 10 HEAPSORT

```
1: procedure HEAPSORT(A, n)
2:     create an empty R with size m^2
3:     for i ← 0 to n-1 do
4:         INSERT(R, A[i])
5:     end for
6:     for i ← 0 to n-1 do
7:         A[i] = EXTRACTMIN(R)
8:     end for
9: end procedure
```

The time complexity is $O(n \cdot 2m + n \cdot m) = O(3nm) = O(3m^3) = O(m^3)$.

(d) Show that $\Omega(m^2 \log m)$ is a hard lower bound for the problem of sorting the contents of an $m \times m$ rectangular heap $\mathcal{R}$ in ascending order.

It's a sorting problem based on comparison. Already known the harder lower bound is $\Omega(n \log n)$. And $n = m^2$, therefore the lower bound is $\Omega(m^2 \log m^2) = \Omega(m^2 \log m)$.

(e) Given $\mathcal{R}$ of size $m \times m$, describe an optimal algorithm for sorting the contents of $\mathcal{R}$ in ascending order.

A cheating way is that see $\mathcal{R}$ as an ordinary array with size m^2, then use whatever quick sort, merge sort or heap sort, the time complexity is $O(m^2 \log m)$.

(f) One disadvantage of regular heaps is that they don't support the efficient search of an arbitrary element. Describe and analyze an efficient algorithm to search for an arbitrary value x in a rectangular heap $\mathcal{R}$.

Can't find a way better than $O(m^2)$.