طراحی الگوریتم

۲۱ و ۲۳ مهر ۹۸

ملکی مجد

Topic	Reference
Recursion and Backtracking	Ch.1 and Ch.2 JeffE
Dynamic Programming	Ch.3 JeffE and Ch.15 CLRS
Greedy Algorithms	Ch.4 JeffE and Ch.16 CLRS
Amortized Analysis	Ch.17 CLRS
Elementary Graph algorithms	Ch.6 JeffE and Ch.22 CLRS
Minimum Spanning Trees	Ch.7 JeffE and Ch.23 CLRS
Single-Source Shortest Paths	Ch.8 JeffE and Ch.24 CLRS
All-Pairs Shortest Paths	Ch.9 JeffE and Ch.25 CLRS
Maximum Flow	Ch.10 JeffE and Ch.26 CLRS
String Matching	Ch.32 CLRS
NP-Completeness	Ch.12 JeffE and Ch.34 CLRS

Pseudocode of Huffman's algorithm

```
HUFFMAN(C)
1  n ← |C|
2  Q ← C
3  for i ← 1 to n − 1
4       do allocate a new node z
5          left[z] ← x ← EXTRACT-MIN(Q)
6          right[z] ← y ← EXTRACT-MIN(Q)
7          f[z] ← f[x] + f[y]
8          INSERT(Q, z)
9  return EXTRACT-MIN(Q)          ▷ Return the root of the tree.
```

Correctness of Huffman's algorithm

- Let C be an alphabet in which each character $c \in C$ has frequency $f[c]$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

- Let C be a given alphabet with frequency $f[c]$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let $C2$ be the alphabet C with characters x, y removed and (new) character z added, so that $C2 = C - \{x, y\} \cup \{z\}$; define f for $C2$ as for C, except that $f[z] = f[x] + f[y]$. Let $T2$ be any tree representing an optimal prefix code for the alphabet $C2$. Then the tree T, obtained from $T2$ by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

Correctness of Huffman’s algorithm

- اثبات لم ها در کلاس توضیح داده شده است.

Amortized Analysis

- In an *amortized analysis*, the time required to perform a sequence of data-structure operations is averaged over all the operations performed.
- the average cost of an operation is small
  - even though a single operation within the sequence might be expensive.
- Amortized analysis differs from average-case analysis in that probability is not involved
  - an amortized analysis guarantees the average performance of each operation in the worst case

three most common techniques used in amortized analysis.

- aggregate analysis
- the accounting method
- the potential method

- یادآوری:
  - هزینه های نسبت داده شده در فرایند تحلیل سرشکن، برای تحلیل استفاده می شود. در کُد برنامه نویسی، از این هزینه ها استفاده نمی کنیم.

Aggregate analysis

- we show that for all n, a sequence of n operations takes worst-case time T (n) in total. In the worst case, the average cost, or **amortized cost**, per operation is therefore T (n)/n.
  - all operations have the same amortized cost.

The accounting method

- When there is more than one type of operation
  - each type of operation may have a different amortized cost

- we assign differing charges to different operations, with some operations charged **more or less than they actually cost**.
  - The amount we charge an operation is called its ***amortized cost***.

- When an operation's amortized cost exceeds its actual cost, the difference is assigned to specific objects in the data structure as prepaid ***credit***.
  - Credit can be used later on to help pay for operations whose amortized cost is less than their actual cost.

The accounting method (2)

- This method is very different from aggregate analysis, in which all operations have the same amortized cost

- The total credit stored in the data structure is the difference between the total amortized cost and the total actual cost
  - Must be greater than zero
- the total amortized cost of a sequence of operations must be an upper bound on the total actual cost of the sequence

$$\sum_{i=1}^{n} \widehat{c}_i \geq \sum_{i=1}^{n} c_i$$

The potential method

- Is like the accounting method in that we determine the amortized cost of each operation and may overcharge operations early on to compensate for undercharges later
- maintains the credit as the "potential energy" of the data structure **as a whole** instead of associating the credit with individual objects within the data structure

- For each $i = 1, 2, \ldots, n$, we let c_i be the actual cost of the ith operation
- D_i be the data structure that results after applying the ith operation to data structure D_{i-1}.
- A ***potential function*** maps each data structure D_i to a real number $\Phi(D_i)$,
- ***amortized cost*** $\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$.

The potential method (2)

- the total amortized cost of the *n* operations is

$$\begin{aligned}\sum_{i=1}^{n} \widehat{c_i} &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \ .\end{aligned}$$

- require that $\Phi(D_i) \geq \Phi(D_0)$ for all i ,
  - guarantee, as in the accounting method, that we pay in advance
  - define (D_0) to be 0 and then show that $(D_i) \geq 0$ for all i
- ($(D_i) \geq 0$ for all i) => the total amortized cost of a sequence of n INCREMENT operations is an upper bound on the total actual cost

Example1 : stack

- **Stack operations**

PUSH*(S, x)* pushes object *x* onto stack *S*. O(1)

POP*(S)* pops the top of stack *S* and returns the popped object. O(1)

MULTIPOP*(S, k) // pops several objects at once O(min(s,k))*

while not STACK-EMPTY*(S)* and *k* != 0

do POP*(S)*

$k \leftarrow k - 1$

Example1 : stack

- aggregate analysis
  - For any value of n, any sequence of n PUSH,POP, and MULTIPOP operations takes a total of O(n) time. The average cost of an operation is O(n)/n = O(1).
  - all three stack operations have an amortized cost of O(1).

- The accounting method

	actual costs:		amortized costs:
PUSH	1 ,	PUSH	2 ,
POP	1 ,	POP	0 ,
MULTIPOP	$\min(k, s)$,	MULTIPOP	0 .

Example1 : stack (The potential method)

- define the potential function on a stack to be the number of objects in the stack

$$\begin{aligned} \Phi(D_i) &\geq 0 \\ &= \Phi(D_0) \end{aligned}$$

.

$$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1$$

$$\begin{aligned} \widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 = 2\,. \end{aligned}$$

Example1 : stack (The potential method 2)

- the amortized cost of the MULTIPOP operation is

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0 .\end{aligned}$$

- The amortized cost of each of the three operations is *O(1)*

Example2: a binary counter

- a k-bit binary counter
  - counts up from 0 by means of the single operation INCREMENT.
- use an array A[0..k−1] of bits, where length[A] = k, as the counter.
  - A binary number x that is stored in the counter has its lowest-order bit in A[0] and its highest-order bit in A[k−1]
- Initially, x =0, and thus A[i] =0 for i =0,1,...,k−1.
  - To add 1 (modulo 2k) to the value in the counter, we use the following procedure.
- INCREMENT(A)

```
i ←0
while i < length[A] and A[i] =1
      do A[i] ←0
         i ←i +1
 if i < length[A]
      then A[i] ←1
```

incrementing a binary counter

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

aggregate analysis

- The total number of flips in the sequence is

$$\sum_{i=0}^{\lfloor \lg n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

- The worst-case time for a sequence of n INCREMENT operations on an initially zero counter is therefore O(n). The average cost of each operation, and therefore the amortized cost per operation, is O(n)/n = O(1).

The accounting method

- charge an amortized cost of 2 dollars to set a bit to 1
  - When a bit is set, we use 1 dollar (out of the 2 dollars charged) to pay for the actual setting of the bit, and we place the other dollar on the bit as credit to be used later when we flip the bit back to 0.
- The number of 1's in the counter is never negative, and thus the amount of credit is always nonnegative
- for *n* INCREMENT operations, the total amortized cost is *O(n)*, which bounds the total actual cost.

The potential method

- define the potential of the counter after the *i*th INCREMENT operation to be b_i , the number of 1's in the counter after the *i*th operation

- The actual cost of the operation(the *i*th INCREMENT operation resets t_i bits)
  - at most t_i+ 1
  - If b_i= 0, b_{i-1}= t_i= *k*.
  - *If* b_i> 0, then b_i= b_{i-1} −t_i+1

The potential method(2)

- In either case, $b_i \le b_{i-1} - t_i + 1$, and the potential difference is

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &\le (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i \ .\end{aligned}$$

- The amortized cost is therefore

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\le (t_i + 1) + (1 - t_i) \\ &= 2 \ .\end{aligned}$$

analyze the counter even when it does not start at zero

- There are initially b_0 1's, and after *n* INCREMENT operations there are b_n 1's, where $0 \le b_0, b_n \le k$.

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0) .$$

$$\begin{aligned} \sum_{i=1}^{n} c_i &\le \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + b_0 . \end{aligned}$$

analyze the counter even when it does not start at zero

- since $b_0 \leq k$, as long as $k = O(n)$, the total actual cost is $O(n)$
- if we execute at least $n = (k)$ INCREMENT operations, the total actual cost is $O(n)$, no matter what initial value the counter contains

$$\begin{aligned}\sum_{i=1}^{n} c_i &\leq \sum_{i=1}^{n} 2 - b_n + b_0 \\ &= 2n - b_n + b_0 \,.\end{aligned}$$