CSL351: Analysis and Design of Algorithms
Assignment No. 5

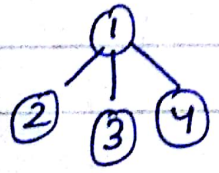
Sahil
2016UCS0008

September 30, 2018

Ans-1 In the vertex coloring problem, we have to assign the smallest no. of colors to the vertices of the given graph so that no 2 adjacent vertices are the same color.

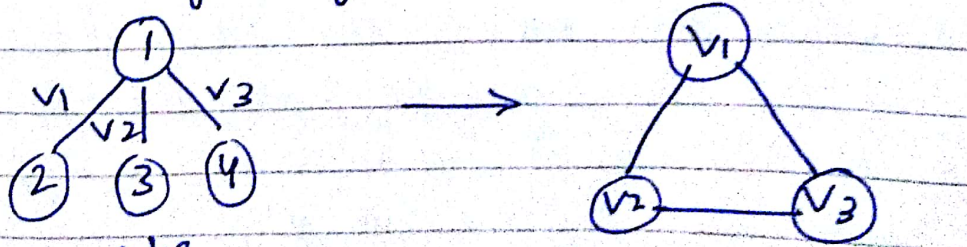
Whereas, in the edge coloring problem, we need to assign the smallest no. of colors to the edges of the graph so that no 2 edges with the same endpoint are the same color.

Consider an example



For vertex-coloring, we need to have min. 2 colors in this case and min. 3 colors for edge-coloring.

To transform it to ~~an edge~~ a vertex-coloring problem, we create a new graph whose ~~edges~~ vertices represent the edges of this graph and we connect two vertices in the new graph iff these vertices represent two edges with a common endpoint in the original graph.



~~Vertex~~ edge-coloring (3)

vertex-coloring (3)

Thus, we can always transform an edge-coloring problem to a vertex-coloring vertex since each edge can be considered as a vertex in the new graph.

But, it is not possible to transform a vertex-coloring problem to an edge-coloring problem, since vertices are connected to each other, so if we convert each vertex to an edge, it is not clear how to connect those edges to make vertices.

Edge-coloring $\xrightarrow{\checkmark}$ Vertex Coloring

Vertex-coloring $\xrightarrow{\times}$ Edge-coloring

Ans-2

The location for post office is (x, y) and the n points are $(x_1, y_1), \ldots (x_n, y_n)$.

We need to ~~min~~ minimize,

$$d = \frac{1}{n} \sum_{i=1}^{n} (|x_i - x| + |y_i - y|)$$

$$\Rightarrow d = \left(\frac{1}{n} \sum_{i=1}^{n} |x_i - x|\right) + \left(\frac{1}{n} \sum_{i=1}^{n} |y_i - y|\right)$$

We can reduce the problem to solving the 2 problems of finding x and y separately.

i.e. $d = d_1 + d_2$

We minimize d_1 and d_2 separately.

$$d_1 = \frac{1}{n} \sum_{i=1}^{n} |x_i - x|$$

To minimize

$$\frac{d}{dx} \sum_{i=1}^{n} |x_i - x| = \frac{d}{dx} \sum_{i=1}^{n} \sqrt{(x_i - x)^2}$$

$$= \sum_{i=1}^{n} \frac{d}{dx} \sqrt{(x_i - x)^2}$$

$$= \sum_{i=1}^{n} \frac{-2(x_i - x)}{2\sqrt{(x_i - x)^2}}$$

$$= \sum_{i=1}^{n} \frac{(x - x_i)}{|x - x_i|}$$

It can be zero only when x = median of x_i's.

Thus, $x = \text{median}(x_1, x_2, \ldots, x_n)$

$y = \text{median}(y_1, y_2, \ldots, y_n)$

Ans- 3 Given the system of linear eqn $Ax = B$

In Cramer's rule, we compute $x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$ using

$$x_1 = \frac{\det(A_1)}{\det(A)}, \quad \cdots \quad x_n = \frac{\det(A_n)}{\det(A)}$$

where A_j is the matrix obtained by replacing j-th column of B.

In Gaussian elimination, we convert the augmented matrix $A|B$ to an upper-triangular matrix using forward elimination and then perform backward substitution to get x_i's.

We can consider the no. of operations in Gaussian Elimination to be cn^3 since it is $\Theta(n3)$ algorithm.

→ In Cramer's rule, we have to compute $(n+1)$ determinants and perform n divisions.

→ Now, determinant can either be computed in $o(n!)$ time using recursive approach or $\Theta(n3)$ time using ~~Gaussian elimination~~ approach similar to the Gaussian elimination, converting matrix to an upper-triangular matrix. Then, determinant value is simply some constant times the product of elements on main-diagonal of upper Δ matrix.

We assume that it also takes cn^3 multiplications.

So, $T_1(n) = cn^3$ for Gaussian elimination

$T_2(n) = (n+1)cn^3 + (4c)(n)$

since for Cramer's rule

Assuming time for multiplication $= c$, division $= 4c$

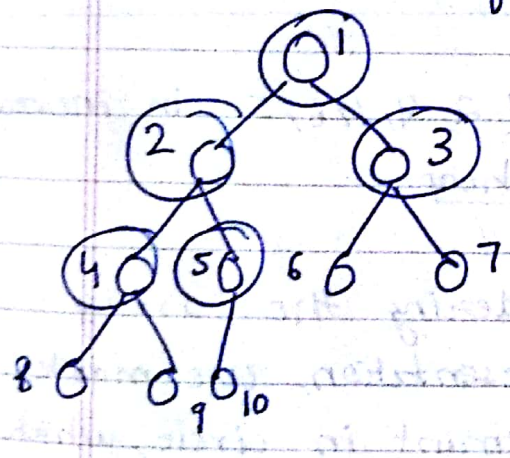
$$\Rightarrow \frac{T_2(n)}{T_1(n)} = \frac{(n+1)cn^3 + 4nc}{cn^3}$$

$$= (n+1) + \left(\frac{4}{n^2}\right)$$

Thus, Gaussian elimination is n times faster than Cramer's rule.

Ans-4 We consider heap as the max-heap where each node of heap has a value larger than both of its children nodes.

So, to determine whether H[1---n] is a heap, we simply check all nodes that have children whether there value is larger than the value of children nodes.



n=10

for eg. when n=10
only H[1], H[2], --- H[5] need to be checked with their children.

So, we check
for i = 1, 2, --- ⌊n/2⌋
whether

$$H[i] \geq \max\{H(2i), H(2i+1)\}$$

If 2i+1 > n then we just check whether H[i] ≥ H[2i].

for eg. in case of n=10, at i=5, we just check H[5] ≥ H[10].

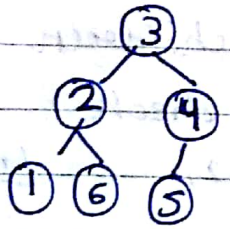
So, if this inequality resulted true for all elements, we say that H[1---n] is a heap, otherwise it not.

Since we perform a constant no. of comparisons in each of the ⌊n/2⌋ iterations, the time complexity is O(n), i.e. linear.

Algorithm / Pseudo-code

```
is-heap = ~~false~~ true
for i = 1 to ⌊n/2⌋
    if (H[i] < H[2*i])
        is-heap = false
        break
    if (2*i+1 <= n and H[i] < H[2*i+1])
        is-heap = false
        break
return is-heap
```

Ans-5 (a) To sort the elements 3, 2, 4, 1, 6, 5 in increasing order, we use a max-heap.



Considering the array representation, we mark an element in circle, which is being heapified.
→ Underlined elements are swapped.
→ Arrow points to children where comparisons are made.

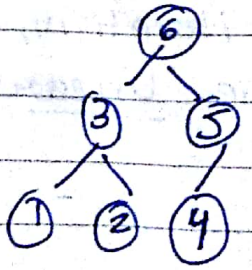
(i) → 3, 2, (4), 1, 6, 5
~~3, 2, 5, 1, 6, 4~~

(ii) → 3, (2), 5, 1, 6, 4

(iii) → (3), 6, 5, 1, 2, 4

(iv) → 6, (3), 5, 1, 2, 4

So, after heapifying the entire array -



Now, we keep on deleting elements one-by-one from the heap so that finally, we get sorted array.

start: 6, 3, 5, 1, 2, 4

(i) 4, 3, 5, 1, 2 | 6

→ 5, 3, 4, 1, 2 | 6

(ii) 2, 3, 4, 1 | 5, 6

→ 4, 3, 2, 1 | 5, 6

(iii) 1, 3, 2, | 4, 5, 6

3, 1, 2 | 4, 5, 6

(iv) 2, 1 | 3, 4, 5, 6

2, 1 | 3, 4, 5, 6 (no change after heapify)

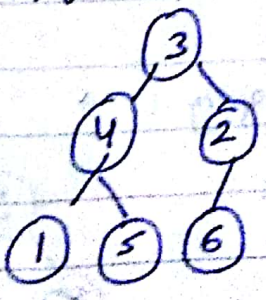
(v) 1 | 2, 3, 4, 5, 6

(vi) | 1, 2, 3, 4, 5, 6

sorted array.

(1) vertical bar represents partition blw elements present in heap and deleted elements.

(b) To sort 3, 4, 2, 1, 5, 6 in decreasing order, we can use a Min-heap.



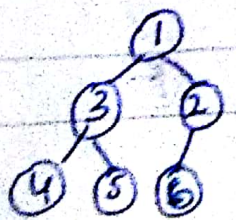
(i) 3, 4, 2, 1, 5, 6

(ii) 3, 4, 2, 1, 5, 6

3, 1, 2, 4, 5, 6

(iii) 3, 1, 2, 4, 5, 6

← 1, 3, 2, 4, 5, 6



Min-Heap formed.

Now, using Min-key deletions

1, 3, 2, 4, 5, 6

i) 6, 3, 2, 4, 5 | 1

2, 3, 6, 4, 5 | 1

ii) 5, 3, 6, 4 | 2, 1

3, 5, 6, 4 | 2, 1

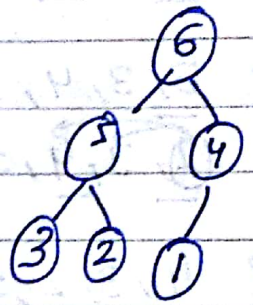
iii) 4, 5, 6 | 3, 2, 1

iv) 5, 6 | 4, 3, 2, 1

v) 6 | 5, 4, 3, 2, 1

vi) | 6, 5, 4, 3, 2, 1

Final Heap / Array



(c) H, E, A, P, S, O, R, T sorting in alphabetical order using Max-Heap.

Heapify

(i) H, E, A, P, S, O, R, T

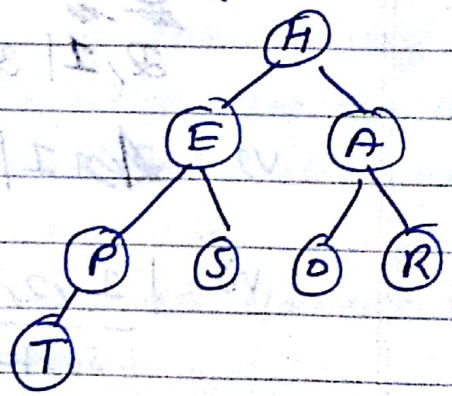
ii) H, E, A, T, S, O, R, P

(iii) H, E, R, T, S, O, A, P

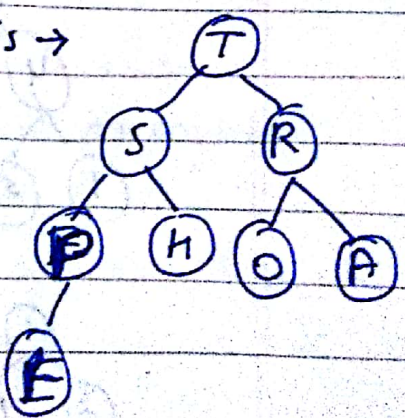
iv) H, T, R, E, S, O, A, P

→ T, H, R, E, S, O, A, P

T, S, R, E, H, O, A, P



so, final heap after heaping is →



Now, performing max-key deletions:

T, S, R, P, H, O, A, E

i) E, S, R, P, H, O, A | T

S, P, R, E, H, O, A | T

→ S, P, R, E, H, O, A | T

ii) A P, R, E, H, O | S, T

→ R, P, A, E, H, O | S, T

~~iii)~~ → R, P, O, E, H, A | S, T

iii) A, P, O, E, H | R, S, T

→ P, A, O, E, H | R, S, T

~~iv)~~ → P, H, O, E, A | R, S, T

iv) A, H, O, E | P, R, S, T

→ O, H, A, E | P, R, S, T

v) E, H, A | O, P, R, S, T

H, E, A | O, P, R, S, T

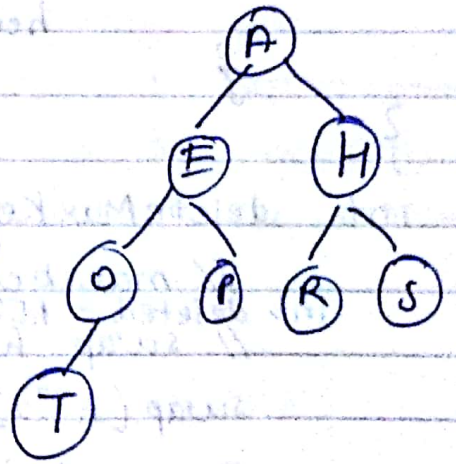
vi) ~~H~~, E | H, O, P, R, S, T

E, A | H, O, P, R, S, T

vii) A | E, H, O, P, R, S, T

viii) | A, E, H, O, P, R, S, T

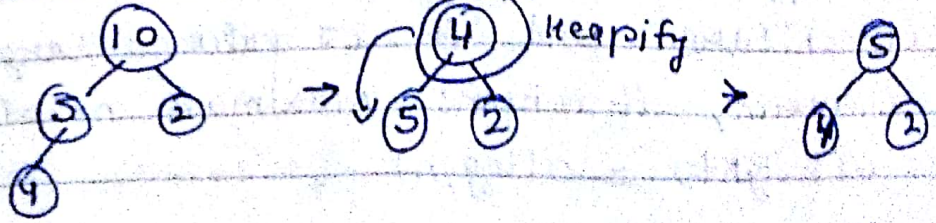
Answer:



Ans-6(a) Maximum key deletion from the given heap (assuming max-heap)

since maximum key is the root in the heap, we first swap it with the last element of heap and then heapify the root so that resultant ~~tree~~ is a heap.

eg -



Pseudo code program :-

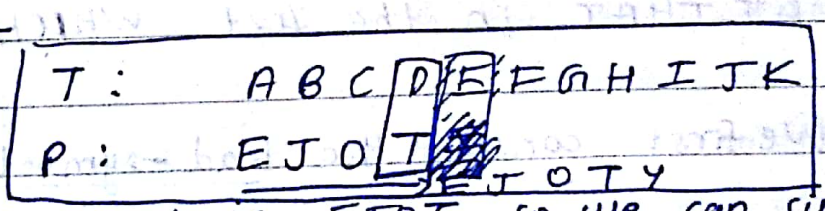
```
void heapify (int h[], int i, int n) {
        int max_index = i;
        if (2*i <= n && h[2*i] > h[i])
                max_index = 2*i;
        if (2*i + 1 <= n && h[2*i+1] > h[max_index])
                max_index = 2*i + 1;
        if (max_index != i){
                swap(h[i], h[max_index]);
                heapify(h, max_index, n);
        }
}
int deleteMaxKey (int h[], int n){
        // max. key is at h[1]
    int deleted = h[1];
        // swap h[1] with h[n]
        swap(h[1], h[n]);
        n = n-1; // decrement heap size
        // call heapify on the root
        heapify(h, 1, n);
        return deleted;
}
```

The complexity of the deleteMaxKey function is $O(\log_2 n)$ since swap takes $O(1)$ and heapify takes $O(\log_2 n)$ because at each level, it compares the swapped element / initial element to heapify, with its children and if its value is larger than the children, it returns. Maximum no. of levels = $O(\text{height}) = O(\log_2 n)$

(b) Since the input to the heapSort algorithm is an array of integers, if we consider the heap also as an array representation, then heapSort can be consider an instance-simplification algorithm since ~~it def~~ the heap array defines an ordering between h[i], h[2*i] and h[2*i+1] which helps in sorting the array.
Whereas, if we consider heap as a tree implementation, then heap sort can be considered as ~~a~~ a representation change algorithm since we converted array to a tree.

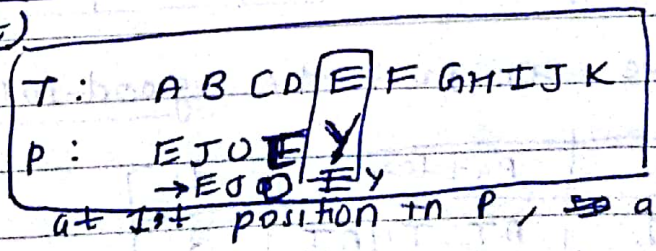
Ans-7 (a) When the character is not in the pattern, we shift by the entire length of the pattern in the Horsepool's algorithm.

eg –



Here, D is not in EJOT, so we can simply shift by length of P = 4 to find a possible match.

(b) When the character is in the pattern (but not the rightmost)



E occurs at 1st position in P, and also at 4th position, so we shift so as to align with the rightmost occurrence of E. rightmost
So, we shift by distance of occurrence of character from end of pattern. (not including ending character as rightmost occurrence)

(c) When the rightmost characters do match-

T: A B C D E F G H I J K

P: E J D E

length m → E J D E

We check till the preceding characters keep matching here C doesn't match with J.

(i) Now, if the character is not present in the other (m-1) characters of pattern, we shift by m.

(ii) If there are other occurrences of character in the pattern, we shift by the distance of its rightmost ~~occurs~~ occurrence from the end in the first (m-1) characters of the pattern.

Ans-8 (a) Applying Boyers-Moore algorithm to search for AT_THAT in the text WHICH_FINALLY_HALTS.--

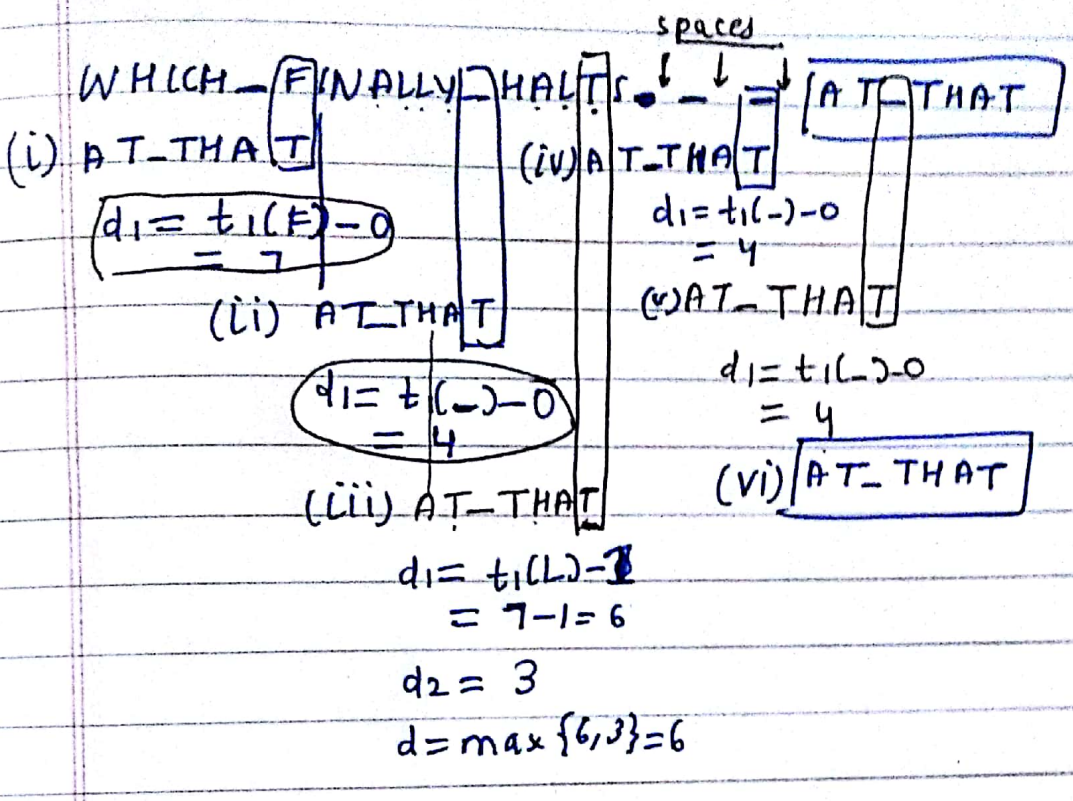
AT_THAT

We first construct the bad-symbol shift table:

c	A	B	C	---	H	---	T	---	Z	.	-	space
$t_1(c)$	1	7	7	7	2	7	3	7	7	7	4	7

Now, we construct the good-suffix table:

k	Pattern	d_2
1	AT_THAT	3
2	AT_THAT	5
3	AT_THAT	5
4	AT_THAT	5
5	AT_THAT	5
6	AT_THAT	5
7	AT_THAT	7



We get a successful match in step (vi) as shown above.

(b) Good-~~Sha~~ Suffix Shift Rule

This rule is guided by a successful match of the last $k > 0$ characters of the pattern.

We refer to the ending portion of the pattern as its suffix of size k and denote it suff(k).

- If there is another occurrence of suff(k) not preceded by same character which caused mismatch, then we can shift the pattern by distance d_2 between such a second rightmost occurrence of suff(k) and its rightmost occurrence.
- When there is no other such occurrence of suff(k), we find the longest prefix of size $l < k$ that matches the suffix of same size l. If such a prefix exists, the shift size d_2 is computed as the distance between the prefix and the corresponding suffix; otherwise d_2 = length of pattern.

Ans-9 17.1-2 Show that if a DECREMENT operation were included in the k-bit counter example, n operations could cost as much as $\Theta(nk)$ time.

The ~~incr~~ INCREMENT operation already present is:

INCREMENT(A)

```
i=0
while i< A.length and A[i]==1
        A[i]=0
        i=i+1
if i< A.length
        A[i]=1
```

Similar to increment, ~~while~~ for decrementing, ~~if suppose we wish to decrement~~ if last bit is 1, it simply becomes 0, otherwise the carry is taken from next position. If next position is also 0, then the carry is taken from next to this. This process goes on till we encounter a 1. The 1 is flipped to 0 and rest all 0's after it flipped to 1.

eg.

```
   ← i
   100|1 0000|
       -    1
   ----------
   100|0 1111|
```

→ Notice the last 5 bits are flipped as explained above.

DECREMENT (A)

```
i = 0
While  i < A.length and A[i] == 0
       A[i] = 1
       i = i + 1
if  i < A.length
       A[i] = 0
```

If we had to consider a sequence of DECREMENT operations, we could do a similar analysis ~~for~~ as INCREMENT, saying A[0] flips each time, A[1] flips every other time so in a sequence of n operations A[1] flips $\lfloor n/2 \rfloor$ times. Similarly, A[2] flips only every 4th time, or $\lfloor n/4 \rfloor$ times in a sequence of n DECREMENT operations.

Thus, total no. of flips

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

$$= 2n.$$

~~Here, it is assumed that we~~

Thus, amortized complexity of DECREMENT ONLY

$$= \frac{2n}{n} = O(1)$$

* But, in the question we have to consider a sequence of n INCREMENT or DECREMENT, i.e. mixed type of operations.

So, consider the case when we reach to a state of $0\underbrace{1\,1\,1\,\ldots\,1}_{(k-1)\text{ ones}}$ after $(2^k - 1)$ increments provided we started from all zeroes.

Now, doing 1 increment flips all the 'k' bits. Then, if we had a decrement operation, again all 'k' bits are flipped. continuing such sequence of operations, we would have a complexity of $\theta(kn)$ for n operations.

Thus, if we have $x = n-(2^k-1) = n-2^k+1$

so, complexity $= \Theta(k(n-2^k+1))$

$= \Theta(nk)$

17.2-1 Suppose we perform a sequence of stack operations on a stack whose size never exceeds k. After every k operations, we make a copy of the entire stack for back up purposes. Show that the cost of n stack operations, including copying the stack, is O(n) by assigning suitable amortized costs to the various stack operations.

Ans- Let us give the following amortized costs to each of the operation:

Push – 2

Pop – 2

Copy – 0

Now, whenever we push an element to the stack, it costs (2). This now includes the cost of copying this element since we ~~can make at most $\frac{n}{k}$ copies as after every k-th operation, we make a copy~~ only pay 1 unit and 1 is left. Now, we can have atmost k element in the stack, ~~thus total amortized cost of n operations $\leq k * \frac{n}{k} + n \leq 4 * n$ (max pops)~~

~~$= 2n$~~

~~$= O(n)$~~

so whenever we perform a copy operation, we already have 1 unit cost left with each element which can be used to pay for copy operation. This is because k PUSH and POP operations occur before each copy, thus k extra units are always there. Since amortized cost of each operation = O(1), Total amortized cost = O(n) for n operations.

17.2-2 Redo Exercise 17.1-3 using an accounting method of analysis.

17.1-3 Suppose we perform a sequence of n operations on a data structure in which the ith operation costs i if i is an exact power of 2, and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation.

Ans — Cost of i-th operation,

$$T(i) = \begin{cases} i, & \text{if } i \text{ is a power of 2} \\ 1, & \text{otherwise} \end{cases}$$

$\Rightarrow T(1)=1,\ T(2)=2,\ T(3)=1, T(4)=4, \ldots$

Total cost of n-operations $= \sum_{i=1}^{n} T(i)$

$$= \sum_{i=0}^{k} 2^i + n - (k+1)$$

where $2^k = n \Rightarrow k = \log_2 n$

Thus, total cost $= 2^{k+1} - 1 + n - k - 1$

$= 2^{\log_2 n + 1} + n - \log_2 n - 2$

$= 2 \cdot 2^{\log_2 n} + n - \log_2 n - 2$

Total cost $= 3n - \log_2 n - 2$ → (1)

Thus, amortized cost $=$ $\frac{\text{Total cost}}{\text{no. of operations}}$

$= 3 - \frac{\log_2 n}{n} - \frac{2}{n}$

~~Thus, amortized cost = $O\left(\frac{\log_2 n}{n}\right)$~~

Since amortized cost < 3

$\Rightarrow$ Amortized cost $= O(1)$

* solving the same problem using accounting method

In this method, we assign an amortized cost ($\hat{c}_i$) which is ~~larger than~~ different from the actual cost (c_i) but at any point, the total amortized cost must be larger than the total actual cost of the operations.

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

Since the ~~total~~ cost of n operations was coming out smaller than 3n, we can associate an amortized cost of 3 units to each operation.

$\Rightarrow \hat{c}_i = 3$, constant

So, if i is an exact power of 2, we pay i units by using the credit, which is $\boxed{\sum_{j=1}^{i} \hat{c}_j - \sum_{j=1}^{i} c_j}$,

otherwise we pay 1 unit and store 2 units as credit.

$$\Rightarrow \sum_{i=1}^{n} \hat{c}_i = 3n \geq \sum_{i=1}^{n} c_i = 3n - \log_2 n - 2$$

Thus, the amortized cost per operation = $O(1)$.

17-2-3 Suppose we wish not only to increment a counter but also to reset it to zero. Counting the time to examine or modify a bit as $\Theta(1)$ show how to implement a counter as an array of bits so that any sequence of n operations takes time $O(n)$ on an initially zero counter.

Ans- To implement a RESET operation, we have to flip all the set bits to zero. So, if we keep the position of last 1 from the least significant bit, we need not check the bits after this last 1 while resetting.

```
INCREMENT(A)
i=0
last=-1
while i < A.length and A[i]==1
        A[i]=0
        i=i+1
if  i < A.length
        A[i]=1
        if i > last
            last=i
else
    last=-1   // no 1's in counter
```

Now, in the RESET operation, we simply check all bits from 0 to last and ~~flip if the set~~ set to 0.

```
RESET(A)
    for i = 0 to last
        A[i] = 0
    last = -1
```

We can now use Accounting method to show that any sequence of n operations takes time O(n).
Now, since each INCREMENT operation sets a bit, we associate an amortized cost of 2 in the original scenario, 1 for setting it and another for flipping it later during INCREMENT. We pay an additional 3rd unit to be used when we ~~set~~ RESET the counter. Also, an additional 4th unit to update the last variable. It might not take place each time but it will make sure we don't undercharge.
Now, whenever we perform a RESET operation we have already paid for setting all bits which were made 1 during increment.

INCREMENT = 4 units
RESET = 0 units.

Thus, total amortized cost for n operations
= 4n = O(n)

17.3-1 Suppose we have a potential function Φ such that $\Phi(D_i) \geq \Phi(D_0)$ for all i, but $\Phi(D_0) \neq 0$. Show that there exists a potential function Φ' such that $\Phi'(D_0) = 0$, $\Phi'(D_i) \geq 0$ for all $i \geq 1$, and the amortized costs using Φ' are the same as the amortized costs using Φ.

Ans- We can define $\Phi'(D_i) = \Phi(D_i) - \Phi(D_0)$ $\forall\ i \geq 0$

$\Rightarrow \Phi'(D_0) = \Phi(D_0) - \Phi(D_0) = 0$

and $\Phi'(D_i) = \Phi(D_i) - \Phi(D_0) \geq 0 \quad \forall\ i \geq 1$

(since $\Phi(D_i) \geq \Phi(D_0)$)

Now, the amortized cost using Φ' =

$$\hat{c}_i' = c_i + \Phi'(D_i) - \Phi'(D_{i-1})$$
$$= c_i + \Phi(D_i) - \Phi(D_0) - \left(\Phi(D_{i-1}) - \Phi(D_0)\right)$$
$$= c_i + \left(\Phi(D_i) - \Phi(D_{i-1})\right)$$
~~$= c_i + \Phi$~~
$$= \hat{c}_i$$

Hence, it is same as the amortized cost using Φ.

17.3-2 Redo Exercise 17.1-3 using a potential method of analysis.

Ans- It is given that cost of i-th operation,

$$c_i = \begin{cases} i, & \text{if } i \text{ is a power of 2} \\ 1, & \text{otherwise} \end{cases}$$

~~Let Φ~~ using the potential method of analysis, we need to define a potential function Φ such that $\Phi(D_i) \geq \Phi(D_0)$ $\forall\ i$.

It is convenient to use $\Phi(D_0) = 0$.

Let $\Phi(D_i) = k+3$ if $i = 2^k$

otherwise, let k be the largest integer such that $2^k \le i$

Then $\Phi(D_i) = \Phi(D_{2^k}) + 2(i-2^k)$

Also $\Phi(D_0) = 0$

Clearly, $\Phi(D_i) \ge 0$ since both terms are +ve.

So, $\Phi(D_i) - \Phi(D_{i-1}) = \Phi(D_{2^k}) + 2(i-2^k)$

If $(i \neq 2^k) \Rightarrow$ $- [\Phi(D_{2^k}) + 2(i-1-2^k)]$

$= 2i - 2^{k+1} - 2i + 2 + 2^{k+1}$

$= 2$

And if $i = 2^k$, $\Phi(D_i) - \Phi(D_{i-1}) =$

~~$\Phi(D_{2^k}) + 2(0) - (\Phi(D_{2^{k-1}}) - 2(i-1-2^{k-1}))$~~

~~$= k+3 - (2^{k-1}+3) + 2(i-1-2^{k-1})$~~

~~$= 2^k + 3 - 2^{k-1} + 2^k 2i - 2 - 2^k - 3$~~

~~$= 2 - 2^{k-1} + 2^{k+1}$~~

~~$= 2\left(2^k - \frac{2^k}{4} - 1\right) = 2\left(\frac{3 \cdot 2^k - 4}{4}\right)$~~

~~$= 3 \cdot 2^{k-1} - 2$~~

$= \Phi(D_{2^k}) + 2(i-1-2^{k-1}) - \Phi(D_{2^{k-1}})$

~~$= k+3 + 2i - 2 - 2^k - (k-1) - 3$~~

~~$= 2^{k+1} - 2^k + 1$~~

~~$= 2^k(2-1) + 1 = 2^k + 1$~~

$= (k+3)$

If $i = 2^k$, $\Phi(D_i) - \Phi(D_{i-1})$

$= \Phi(D_{2^k}) - \Phi(D_{2^k-1})$

$= k+3 - [\Phi(D_{2^{k-1}}) + 2(2^k - 1 - 2^k)]$

$= k+3 - [(k-1)+3 - 2] = k+3-k+1-1$

$= 3$

Thus, total amortized cost

$= \sum_{i=1}^{n} 3 = 3n = O(n)$

17.3-6 Show how to implement a queue with 2 ordinary stacks so that the amortized cost of each ENQUEUE and each DEQUEUE operation is O(1).

Ans- The strategy to implement a queue with 2 stacks is:

for eg. we have the foll. queue

← 1 2 3 4 5 6 ← Rear

Front

If all elements are in stack 1 suppose, in pushed same order

Stack 1 | 1 | 2 | 3 | 4 | 5 | 6 | ← Top.

Now, when dequeue comes, we need to remove 1 which is last in stack, so we pop all elements and push to stack 2.

stack 1 : Empty

stack 2 : | 6 | 5 | 4 | 3 | 2 | 1 | ← Top

Now, we can smoothly perform next 6 dequeues in O(1) by popping from stack 2.

For Enqueue, we keep pushing to 1st stack.

Algorithm

- Enqueue - Push the element to stack 1.
- Dequeue - If stack 2 is empty, then pop all elements from stack 1 and push to stack 2. Now, pop from stack 2.

Otherwise simply pop from stack 2.

We can now use accounting method for the analysis. We assign a credit of 3 units to each ENQUEUE operation. First credit is used for pushing to stack 1, 2nd and 3rd are used for popping from stack 1 and pushing to stack 2 (moving from Stack 1→2)

and the 4th credit can finally be used for a DEQUEUE operation, i.e. when popping from stack 2. enqueued

Now, the element already has with it the cost for dequeue, so dequeue operation has 0 amortized cost.

Thus, total amortized cost for n operations = 4n

$= \boxed{O(n)}$

17.3-7 Design a data structure to support the following 2 operations for a dynamic multiset S of integers, which allows duplicate values:

INSERT (S, x) : inserts x into S

DELETE-LARGER-HALF (S): delete the largest $\lceil |S|/2 \rceil$ elements from S.

Explain how to implement this data structure so that any sequence of m INSERT and DELETE-LARGER-HALF operations runs in O(m) time. Your implementation should also include a way to output the elements of S in O(|S|) time.

Ans- If we use an array to store the elements, insertion will simply take O(1) time, but to delete the larger half of elements, we first need to find these elements.

For this purpose, first we find the median element using the O(n) median-finding algorithm. Then, we scan the array and copy the elements smaller or equal to m in the array of half size.

In this way, we retain the smaller half.

Since the delete larger half operations take O(|S|)

time, we can select a linear potential function. Thus, the amortized cost will be linear for m operations.

- we can also print the elements of multiset, in $O(|s|)$ time by just iterating the array.