Practical Session No. 13 –
Amortized Analysis, Union/Find

**Amortized Analysis**

Refers to finding the average running time per operation, over a worst-case sequence of operations. Amortized analysis differs from average-case performance in that probability is not involved; amortized analysis guarantees the time per operation over worst-case performance. In an amortized analysis, we consider the total time of a sequence of operations. Even if some single operations might be very expensive.

- Aggregate analysis determines the upper bound $T(n)$ on the total cost of a sequence of $n$ operations, then calculates the amortized cost of an operation to be $T(n) / n$.
- Accounting method determines the individual cost of each operation, combining its immediate execution time and its influence on the running time of future operations. Usually, many short-running operations accumulate a "debt" of unfavorable state in small increments, while rare long-running operations decrease it drastically. The costs of actions are represented by tokens. One token represents an operation of one action in the computer. The cheap operations actually account for more token in the accounting method than they actually are (usually still $O(1)$ but with higher constant), and when executing cheap immediate operations, the tokens are accumulated to credit for later “paying” for actual expensive operations.

**Question 1: Incrementing a Bit String**

A is a bit string. What is the time complexity of increment operation using Amortized analysis? Analyze number of bit flips (Show with both methods: aggregate and accounting).

```plaintext
Increment(A[0…m-1])
  i ← 0
  while i < m and A[i] = 1
    A[i] ← 0
    i ← i + 1
  if i < m  A[i] ← 1
```

**Solution:**

**Aggregate Method:**

Assume $n$ increments starting from all 0s.
- $A[0]$ flips every increment for $n$ flips.
- $A[1]$ flips every 2nd time for $\leq n/2$ flips.
- $A[i]$ flips every $2^i$th time for $\leq n/2^i$ flips.
- Number of flips $\leq n + n/2 + n/4 + ... = 2n$ $O(n)$
- Therefore amortized cost of each operation is $2 = O(1)$
Accounting Method:
Assume n increments starting from all 0s.
Increment flips exactly one bit from 0 to 1.
Assign an amortized cost of 2 tokens per increment.
Both tokens are assigned to the bit that is flipped from 0 to 1.
Use one token immediately for flip from 0 to 1.
Save other token for when it is flipped back to 0.
All bit flips are accounted for, so the total cost of 2n is <= number of bit flips.

Question 2: Clearable Table Data Structure
We would like to create an ADT, called clearable table, of a table which support the following operations.

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(e)</td>
<td>Insert a new element to the next unoccupied cell.</td>
</tr>
<tr>
<td>clear()</td>
<td>Empty the table (delete all the elements from the table).</td>
</tr>
</tbody>
</table>

Clearable table is implemented with a table of size N.
Prove that the amortized cost of add(e), clear() is O(1) in each of the two methods (aggregate and accounting)

Solution:
Accounting Method:
Let S be a table of size N.
Assume that n operations of clear and add have been performed on S (at the beginning S was empty).
Let us assume we pay one token for an action that is performed in constant time (basic operation).
We will define a cost to each of the operations:
add - 2 tokens.
clear - 0 tokens.
We set more tokens to add than it actually cost, and we set fewer tokens to clear.
After an element is inserted into the table, it has a spare token. Therefore, meanwhile the clear operation, each element pays the extra token for its deletion from the table.
We get a cost of O(1) for each operation

Aggregate Method:
Assume that n operations of clear and add have been performed on S (In the beginning S was empty).
For each operation of inserting an element to table, we could delete the element at most one time.
Therefore the number of elements that was deleted from the table is at most n.
The total cost of n operations is O(n), and the cost of single operation is O(n)/n=O(1).
Question 3
You must implement a queue using only (a constant number of) stacks, you may only use O(1) extra space besides the stacks. All operations work in O(1) amortized time.

Solution:
We use 2 stacks, A and B.

<table>
<thead>
<tr>
<th>Enqueue(x)</th>
<th>Dequeue()</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(B,x)</td>
<td>if (A is empty)</td>
</tr>
<tr>
<td></td>
<td>while(B is not empty)</td>
</tr>
<tr>
<td></td>
<td>push(A,pop(B))</td>
</tr>
<tr>
<td></td>
<td>if (A is not empty)</td>
</tr>
<tr>
<td></td>
<td>return pop(A)</td>
</tr>
</tbody>
</table>

**Accounting Method:**
Let us define the cost of each operation:
Enqueue(x) - 3 tokens.
Dequeue() - 1 token.
Each element enqueued has 2 spare tokens to later "pay" for moving it from stack B to stack A when needed.
An element is moved from stack B to A at most once, and hence when executing Enqueue(x), the moving of the elements between stacks is already "paid" for.
Thus all operations cost O(1) amortized time.

**Aggregate Method:**
Assume that n operations Enqueue(x) and Dequeue() have been performed on the queue (In the beginning S was empty).
Enqueue(x) always costs O(1), Dequeue() costs (2k+1) where k is the number of Enqueue( ) operations performed right before it. (k=0 if it comes after another Dequeue() or is the first operation executed on the empty stack).
So if we group every Dequeue() operation with all k Enqueue(x) operations right before it we get that the sum of the costs of these (k+1) operations is (1+1+1…+1) + (2k+1) = 3k+1 =< 4k.
Thus grouping all operations in a sequence of blocks of 1 Dequeue() operation preceded with all Enqueue( ) operations right before it we have the total cost of n operations =<4n = O(n), and the cost of single operation is O(n)/n=O(1).
הב�אות זרה – בשיטת

נוטו שלם של איברים \( n \geq 1 \).

urile המחלקה \( U \) \( = \{1,2,\ldots,n\} \).

après הב�אות זרה: \( (\text{Makeset}, \text{Union}, \text{Find}) \).

 opciones הב�אות זרה — \( \text{Makeset}(i) \) — \( \text{Union}(p,q) \) — \( \text{Find}(i) \) — \( \text{Find}(p) \) — \( \text{Find}(q) \) — \( \text{Union}(p,q) \).

לדוגמה, \( \{1,3\} \) \( \{2,4,6,7\} \) \( \text{Find}(6) \).

tom רבצ: \( O(\alpha(m,n)) \) \( \leq 4 \).

ופרטי \( \text{Find}(p) \) \( \text{Find}(q) \) \( \text{Find}(6) \).

ปลอด הפונקציה \( \alpha(m,n) \) \( \leq 4 \).

וכך \( O(m) \) \( O(n) \) \( O(m \cdot \alpha(m,n)) \).

דוגמה

גנתות \( m \) \( \{1,2,\ldots,n\} \) \( (\text{Makeset}(i) \) \( (\text{Union}(p,q) \) \( (\text{Find}(i) \) \( (\text{Find}(p) \) \( (\text{Find}(q) \) \( (\text{Union}(p,q) \)

אפס מחלקות המזהב

<table>
<thead>
<tr>
<th>( (1,2) )</th>
<th>( (1,2) )</th>
<th>( (2,3) )</th>
<th>( (5,7) )</th>
<th>( (4,6) )</th>
<th>( (7,8) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (1,2) )</td>
<td>( (2,3) )</td>
<td>( (4,6) )</td>
<td>( (4,6) )</td>
<td>( (5,7,8) )</td>
<td></td>
</tr>
</tbody>
</table>
1. $P = 4$

2. $O(1)$ if there is a pointer to the last element in the list.
השיטה נספת:Їב הגרירה הביאה ש yatזח האיברים הבוחו לחיצות מוספרים שלמים. ניתן
לחתימת בלשון לילית (כולל פרו לתחום חומרים). לא תוכל לך לחתמת בנבוכה.
הטבלה הממעידה בכילו של Soap כולם מוספרים שלמים, למשלי.

<table>
<thead>
<tr>
<th>פונקציה</th>
<th>Find</th>
<th>Makeset</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

נפתלUNTIFTא על הגרירה של פעולת זו והמנון המושך (וגדר
O(log n) של פעולתcession 준
ולשני פונקציה המבנה נ ogs כל המנוניםdequeי, פונקציה זו Find 몽 "כמעט לפיון" והמנון
O(1) פעולת זו Union 준.

פונקציה שלישית

니יצג כל קובץ כרשימה. כל איבר ברשימה מציב גו לראשה הרשימה, ווג ליאיבר
הבא ברשימה, והיוו מפורק מקו איברים GO ובו תצוגה צויר א. קובץ
המודח מתכתי לראשה הרשימה המ친יה.

 RuntimeException -- Exception
A[i] יוצר לע המתכתי [i]
המחרט המ وغيرها לאש
زهرימה. זמני O(1)

 RuntimeException -- Exception
A[i] יוצר לע המתכתי [i]
לראשה הרשימה המ EINA ש. זמני O(1)
דוגמא

דוגמה: לאחר (p,q) נקבל:

ל сотрудник_pagination.jpg: "מתוך התדשדות, בורח שאות 신יו המאזרוך עדף עלות בכבדה הקובה בממצאים ( Detaylı lדרר שפעלנו בדוגמא זו).

פתרון שלישית משוחרר

שידי ייסגי כאל קובאכרתימה. בנספף, בדד השימים נ ultimo את המוניקה.

יৎנץ עד qui ייקודו עם_find, יכתב עי"ו הוספת הקובא הקובה לקוד הנקובה

המוניקה. כלומר, ראי וטדה השימה الحديد ראי השימה המוניקה ינור, וכל הממצאים

ייבשו עליי.

?

? r = Union(p,q)

כיצד נבצע (p,q)?
Theorem: It can be verified that the set of all integers or floats can be represented with a disjoint-set forest using

\[ \mathcal{O}(m \cdot \alpha(m, n)) \]

where \( \alpha \) is the inverse Ackermann function, and in practical use, this is considered as:

\[ \mathcal{O}(m) \]
HEX is a board game where two players take turns placing black or white stones on an empty hexagonal board. The goal of a player is to create a line of 5 stones of the same color from one side of the board to the other.

**Problem:**

**Solution:**

The game board is a matrix $Board$ of size $K \times K$. Each square $i,j$ on the board represents a cell that can be empty, black, or white. The board is represented as a matrix where:

- $Board[i,j] = 0$ if the cell is empty.
- $Board[i,j] = 1$ if there is a black stone on the cell.
- $Board[i,j] = -1$ if there is a white stone on the cell.

During the game, the two players alternate placing stones on the board to create a path of stones of the same color. The game ends when one player manages to create a line of 5 stones of the same color from one side of the board to the other.

The game can be modeled as a network of hexagonal cells, where each cell represents a node and the edges represent the possible moves between adjacent cells. The algorithm to solve the game involves finding connected components of stones of the same color on the board.

The number of moves $makeset$ is equal to the number of squares on the board plus 4 additional white and black stones. Each move involves finding and merging the connected components of stones of the same color.

The overall complexity of the algorithm is $O(K^2)$, which is the time it takes to find and merge the connected components on the board.