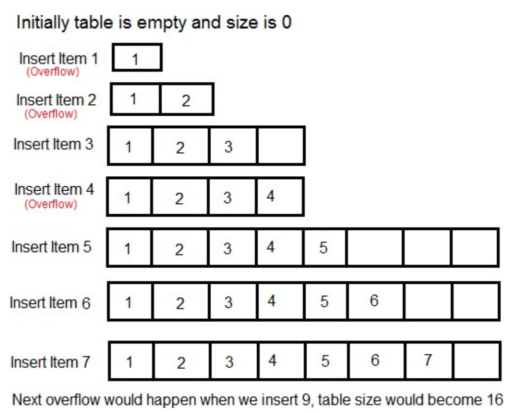
ADVANCE ALGORITHM
Experiment 1

Ayush Jain 60004200132 B3

Aim : To implement Amortized Analysis.

Theory:
Amortized Analysis.

Amortized Analysis is used for algorithms where an occasional operation is very slow, but most of the other operations are faster. In Amortized Analysis, we analyze a sequence of operations and guarantee a worst-case average time which is lower than the worst-case time of a particular expensive operation. The example data structures whose operations are analyzed using Amortized Analysis are Hash Tables, Disjoint Sets and Splay Trees.
Let us consider an example of a simple hash table insertions. How do we decide table size? There is a trade-off between space and time, if we make hash-table size big, search time becomes low, but space required becomes high.



The solution to this trade-off problem is to use Dynamic Table (or Arrays). The idea is to increase size of table whenever it becomes full. Following are the steps to follow when table becomes full.

1. Allocate memory for a larger table of size, typically twice the old table.
2. Copy the contents of old table to new table.
3. Free the old table.

Aggregate Method

The aggregate method is used to find the total cost. If we want to add a bunch of data, then we need to find the amortized cost by this formula.

For a sequence of n operations, the cost is −

$$\frac{Cost(n\ operations)}{n} = \frac{Cost(normal\ operations) + Cost(Expensive\ operations)}{n}$$

The Accounting Method

The accounting method is aptly named because it borrows ideas and terms from accounting. Here, each operation is assigned a charge, called the amortized cost. Some operations can be charged more or less than they actually cost. If an operation's amortized cost exceeds its actual cost, we assign the difference, called a credit, to specific objects in the data structure. Credit can be used later to help pay for other operations whose amortized cost is less than their actual cost. Credit can never be negative in any sequence of operations.

The Potential Method

The potential method is similar to the accounting method. However, instead of thinking about the analysis in terms of cost and credit, the potential method thinks of work already done as potential energy that can pay for later operations. This is similar to how rolling a rock up a hill creates potential energy that then can bring it back down the hill with no effort. Unlike the accounting method, however, potential energy is associated with the data structure as a whole, not with individual operations.

Code:

Aggregate Table:

```
class DynamicTable:
    def __init__(self, capacity=1):
        self.table = [0] * capacity
        self.size = 0
        self.capacity = capacity

    def add(self, element):
        if self.size == self.capacity:
            # Double the capacity of the table if it is full
            new_table = [0] * (self.capacity * 2)
            for i in range(self.size):
                new_table[i] = self.table[i]
            self.table = new_table
            self.capacity *= 2
            # print(f"Table doubled, new size: {self.capacity}")
        self.table.append(element)
        self.size += 1

    def size(self):
        return self.size

    def capacity(self):
        return self.capacity

    def numDoublings(self):
        # Calculate the number of times the table was doubled
        return int(math.log2(self.capacity))

    def numCopyings(self):
        # Calculate the number of times elements were copied during resizing
        num_copyings = 0
        for i in range(1, self.numDoublings() + 1):
            num_copyings += 2**(i-1)
        return num_copyings

table = DynamicTable()
cost = 0
f = 0
operation_cost = 1

print("Item No\tTable Size\tTable Cost\tCost of Operation")
print("=====================================================")

for i in range(1, 18):
    table.add(i)
    if f == 1:
```

```python
        cost = table.size
        f = 0
    else:
        cost = 1
    if table.size == table.capacity:
        f = 1
    print(f"{i}\t{table.capacity}\t\t{cost}\t\t\t{operation_cost}")
```

Output:

```
Item No Table Size      Table Cost      Cost of Operation
=====================================================
1       1               1                       1
2       2               2                       1
3       4               3                       1
4       4               1                       1
5       8               5                       1
6       8               1                       1
7       8               1                       1
8       8               1                       1
9       16              9                       1
10      16              1                       1
11      16              1                       1
12      16              1                       1
13      16              1                       1
14      16              1                       1
15      16              1                       1
16      16              1                       1
17      32              17                      1


...Program finished with exit code 0
Press ENTER to exit console.
```

Accounting using Multipop
Code:

```
class Stack:
    def __init__(self):
        self.items = []
        self.cost = 0
        self.balance = 0

    def push(self, item):
        self.items.append(item)
        self.cost += 1
        self.balance += 1

    def pop(self):
        if not self.is_empty():
            self.cost += 1
            self.balance -= 1
            return self.items.pop()

    def multi_pop(self, k):
        if k > len(self.items):
            k = len(self.items)
        for i in range(k):
            self.pop()

        self.cost += k
        self.balance -= k

    def is_empty(self):
        return len(self.items) == 0

    def get_cost(self):
        return self.cost

    def get_balance(self):
        return self.balance

    def display_table(self, n):
        print("{:<15}{:<15}{:<15}{:<15}".format("Operation", "Total Cost",
"Amortized Cost", "Balance"))
        print("-"*60)
        for i in range(1, n+1):
            if i % 3 == 1:
                self.push(i)
                print("{:<15}{:<15}{:<15.2f}{:<15}".format(f"push({i})",
self.get_cost(), self.get_cost()/i, self.get_balance()))
            elif i % 3 == 2:
                self.pop()
```

```
                print("{:<15}{:<15}{:<15.2f}{:<15}".format("pop()",
self.get_cost(), self.get_cost()/i, self.get_balance()))
            else:
                self.multi_pop(i//3)
                print("{:<15}{:<15}{:<15.2f}{:<15}".format(f"multi_pop({i//3})
", self.get_cost(), self.get_cost()/i, self.get_balance()))
        print()


n = int(input("Enter the number of operations to perform on the stack: "))


stack = Stack()
stack.display_table(n)
```

Output:

```
Enter the number of operations to perform on the stack: 5
Operation      Total Cost     Amortized Cost Balance
------------------------------------------------------------
push(1)        1              1.00           1
pop()          2              1.00           0
multi_pop(1)   2              0.67           0
push(4)        3              0.75           1
pop()          4              0.80           0


...Program finished with exit code 0
Press ENTER to exit console.
```

Potential Method:

```
class DynamicTable:
    def __init__(self, capacity=1):
        self.table = [0] * capacity
        self.size = 0
        self.capacity = capacity

    def add(self, element):
        if self.size == self.capacity:
            # Double the capacity of the table if it is full
            new_table = [0] * (self.capacity * 2)
            for i in range(self.size):
                new_table[i] = self.table[i]
            self.table = new_table
            self.capacity *= 2
        self.table[self.size] = element
        self.size += 1

    def size(self):
        return self.size

    def capacity(self):
        return self.capacity

def potential(table):
    # Potential function, defined as the double of the number of elements
minus the size of the table
    return 2*table.size - table.capacity

def cost(table, operation):
    # Calculate the amortized cost of the operation
    if operation == "add":
        if table.size == table.capacity:
            # Double the table
            return potential(table) + 1
        else:
            return 1
    else:
        raise Exception("Invalid operation")

table = DynamicTable()
total_cost = 0
operation_cost = 1
operation_cos = 1
prev = 0
print("Item No.\tTable Size\tPotential\tOperation Cost\tTotal Cost\tAmortized
Cost")
for i in range(1, 18):
    operation_cost = cost(table, "add")
```

```
    total_cost += operation_cost
    table.add(i)
    # print("="*8)
    pot = potential(table)
    amortized_cost = operation_cost + (pot - prev)
    prev = pot
    # print(amortized_cost)
    print(f"{i}\t\t  {table.capacity}\t\t  {potential(table)}\t\t  {operation_
cos}\t\t  {operation_cost}\t\t  {amortized_cost}")
```

Output:

Item No.	Table Size	Potential	Operation Cost	Total Cost	Amortized Cost
1	1	1	1	1	2
2	2	2	1	2	3
3	4	2	1	3	3
4	4	4	1	1	3
5	8	2	1	5	3
6	8	4	1	1	3
7	8	6	1	1	3
8	8	8	1	1	3
9	16	2	1	9	3
10	16	4	1	1	3
11	16	6	1	1	3
12	16	8	1	1	3
13	16	10	1	1	3
14	16	12	1	1	3
15	16	14	1	1	3
16	16	16	1	1	3
17	32	2	1	17	3

```
...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion: In conclusion, we learned the Amortized Analysis of the Algorithm where an occasional operation is very slow, but most of the other operations are faster.