60004210210
Amartya Mishra
COMPS - C31

AA Experiment 1(A)

Aim: Amortized Analysis of (Aggregate Method of) Dynamic tables.

Theory:

Amortized Analysis is crucial technique in algorithm analysis that aims to provide a more realistic evaluation of an algorithm's performance over a sequence of operations.

One of the methods within amortized analysis is the aggregate method. Unlike the worst case analysis that focuses on the complexity in worst case scenario, Amortized analysis considers the average performance/cost over a series of operation. This Algorithm proves to be efficient in cases where certain operations take much larger complexity than others.

Aggregate Method involves calculating total cost of a sequence of operation & then averaging this cost over all operations

Observations:

In the implementation of Dynamic table, every time we ~~need~~ need to insert something in the table, which if is already full, a new table of double the size is created & all previous

Values are copied in the new table. then new incoming value is added to table.
This is an expensive operation as compared to other O(1) ~~op~~ cost operation of insertion.
Thus in normal scenario, we would worstcase time complexity assuming cost of O(n) at every step But actually only a few steps require O(N) complexity.
Thus in aggregate Method we calculate cost at each operation (O(1) for steps without ~~insertion~~/Expansion & O(N) for steps with Expansion). Then we divide it by the total No. of steps. This gives more accurate Estimation of complexity of the procedure.

Conclusion: This application sucessfully demonstrated the use of amortized analysis using aggregate method to assess efficiency of Dynamic Tables.

AA – Experiment 1(A)

Amartya Mishra

60004210210

COMPS C31

Code:

```
#include<iostream>
#include<stack>
using namespace std;
int cost = 1;
int operation = 0;
int main(){
    stack<int> sizeofarr;
    int arr[100];
    int input[10] = {1,2,3,4,5,6,7,8,9,10};
    sizeofarr.push(1);

    // int *test = incrementarr(input,10);
    for(int i =0; i<10; i++){

        cout<<"the current cost at element "<<input[i]<<" is "<<cost<<endl;
        if(i>sizeofarr.top()-1){
            cout<<"for the element "<<input[i]<<" we need to double the array "<<endl;
            cost += sizeofarr.top();
            cout<<"current size is "<<sizeofarr.top();
            sizeofarr.push(sizeofarr.top()*2);
            cout<<"after increment size is "<<sizeofarr.top()<<endl;
            cout<<"the cost now is "<<cost<<endl;
        }
        operation += 1;
        cost += 1;
    }
    cout<<"amortized cost is "<<cost/operation<<endl;


}
```

Output:

```
/tmp/KiAWDwJaOI.o
the current cost at element 1 is 1
the current cost at element 2 is 2
for the element 2 we need to double the array
current size is 1after increment size is 2
the cost now is 3
the current cost at element 3 is 4
for the element 3 we need to double the array
current size is 2after increment size is 4
the cost now is 6
the current cost at element 4 is 7
the current cost at element 5 is 8
for the element 5 we need to double the array
current size is 4after increment size is 8
the cost now is 12
the current cost at element 6 is 13
the current cost at element 7 is 14
the current cost at element 8 is 15
the current cost at element 9 is 16
for the element 9 we need to double the array
current size is 8after increment size is 16
the cost now is 24
the current cost at element 10 is 25
amortized cost is 2
```