**Implementation of Amortized Analysis (Aggregate Method)**

**Theory**:

Amortized analysis is a method of analyzing algorithms that can help us determine an upper bound on the complexity of an algorithm. This is particularly useful when analyzing operations on data structures, when they involve slow, rarely occurring operations and fast, more common operations. With this disparity between each operations’ complexity, it is difficult to get a tight bound on the overall complexity of a sequence of operations using worst-case analysis. Amortized analysis provides us with a way of averaging the slow and fast operations together to obtain a tight upper bound on the overall algorithm runtime. Here we will consider a simplified version of the hash table problem, and show that a sequence of n insert operations has overall time O(n).

## 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 −



Let *ci* be the cost of the *i*-th insertion:

| **ci = i if i−1 is a power of 2  1 otherwise** |
| --- |

Let's consider the size of the table *si* and the cost *ci* for the first few insertions in a sequence:

| **i 1 2 3 4 5 6 7 8 9 10 si 1 2 4 4 8 8 8 8 16 16 ci 1 2 3 1 5 1 1 1 9 1** |
| --- |

Alternatively we can see that *ci*=1+*di* where *di* is the cost of doubling the table size. That is

| **di = i−1 if i−1 is a power of 2  0 otherwise** |
| --- |

Then summing over the entire sequence, all the 1's sum to *O*(*n*), and all the *di* also sum to *O*(*n*). That is,

| Σ1≤i≤n ci ≤ n + Σ0≤j≤m 2j−1 |
| --- |

where *m* = log(*n* − 1). Both terms on the right hand side of the inequality are *O*(*n*), so the total running time of *n* insertions is *O*(*n*).

Code:



| #include <iostream> #include <bits/stdc++.h> using namespace std;  void print(int arr[], int n){  for(int i = 0; i<n; i++){  cout<<arr[i]<<" ";  }  cout<<endl; }  int main(){   int size = 1;  int count = 0;  int arr[size];  int \*p = arr;  int cost = 0;    while(1){  int n;  cout<<"Enter the number you wish to insert in the dynamic table: ";  cin>>n;  if(count<size){  \*(p + count) = n;  count+=1;  cout<<p<<endl;  print(p, count);  cost+=1;  }else{  //double array  cout<<"Double"<<endl;  int \*new\_arr = new int[size\*2];  for(int i=0; i<count; i++){  // cout<<\*(p+i)<<endl;  new\_arr[i] = \*(p+i);  }  cost+=count+1;  size\*=2;  new\_arr[count] = n;  count+=1;  p = new\_arr;  cout<<p<<endl;  print(p, count);  }  cout<<"Ammortized Cost: "<<cost<<endl;  cout<<endl;  }  return 0; } |
| --- |

**Output**:



| Enter the number you wish to insert in the dynamic table: 5 0x7fff6e35bc10 5  Ammortized Cost: 1  Enter the number you wish to insert in the dynamic table: 10 Double 0x55c2abae66d0 5 10  Ammortized Cost: 3  Enter the number you wish to insert in the dynamic table: 2 Double 0x55c2abae66f0 5 10 2  Ammortized Cost: 6  Enter the number you wish to insert in the dynamic table: 19 0x55c2abae66f0 5 10 2 19  Ammortized Cost: 7  Enter the number you wish to insert in the dynamic table: 5 Double 0x55c2abae6710 5 10 2 19 5  Ammortized Cost: 12  Enter the number you wish to insert in the dynamic table: 23 0x55c2abae6710 5 10 2 19 5 23  Ammortized Cost: 13 |
| --- |

**Observations**:

The aggregate method of amortized analysis is used to find the amortized cost which it does by calculating the average of individual costs.

**Conclusion**:

The aggregate method of amortized analysis gives the amortized cost which might be higher or lower than the individual cost of some steps but is overall lesser than the total worst case scenario and the sequence of n insert operations has overall time O(n).

**References**:

<https://www.cs.cornell.edu/courses/cs3110/2012sp/lectures/lec21-amortized/lec21.html>

<https://www.tutorialspoint.com/Amortized-Analysis>