**Implementation of Amortized Analysis (Accounting 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).

### Accounting (Banker's) Method

The aggregate method directly seeks a bound on the overall running time of a sequence of operations. In contrast, the accounting method seeks to find a *payment* of a number of extra time units charged to each individual operation such that the sum of the payments is an upper bound on the total actual cost. Intuitively, one can think of maintaining a bank account. Low-cost operations are charged a little bit more than their true cost, and the surplus is deposited into the bank account for later use. High-cost operations can then be charged less than their true cost, and the deficit is paid for by the savings in the bank account. In that way we spread the cost of high-cost operations over the entire sequence. The charges to each operation must be set large enough that the balance in the bank account always remains positive, but small enough that no one operation is charged significantly more than its actual cost.

We emphasize that the extra time charged to an operation does not mean that the operation really takes that much time. It is just a method of accounting that makes the analysis easier.

If we let *c'i* be the charge for the *i*-th operation and *ci* be the true cost, then we would like

| *Σ1≤i≤n ci ≤ Σ1≤i≤n c'i* |
| --- |

for all *n*, which says that the ***amortized time*** Σ*1≤i≤n c'i* for that sequence of *n* operations is a bound on the true time Σ*1≤i≤n ci*.

Considering the example of the extensible array. Say it costs 1 unit to insert an element and 1 unit to move it when the table is doubled. Clearly a charge of 1 unit per insertion is not enough, because there is nothing left over to pay for the moving. A charge of 2 units per insertion again is not enough, but a charge of 3 appears to be:

| **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 c'i 3 3 3 3 3 3 3 3 3 3 bi 2 3 3 5 3 5 7 9 3 4** |
| --- |

where bi is the balance after the i-th insertion.

In fact, this is enough in general. Let m refer to the m-th element inserted. The three units charged to m are spent as follows:

* One unit is used to insert m immediately into the table.
* One unit is used to move m the first time the table is grown after m is inserted.
* One unit is donated to element m − 2k, where 2k is the largest power of 2 not exceeding m, and is used to move that element the first time the table is grown after m is inserted.

Now whenever an element is moved, the move is already paid for. The first time an element is moved, it is paid for by one of its own time units that was charged to it when it was inserted; and all subsequent moves are paid for by donations from elements inserted later.

In fact, we can do slightly better, by charging just 1 for the first insertion and then 3 for each insertion after that, because for the first insertion there are no elements to copy. This will yield a zero balance after the first insertion and then a positive one thereafter.

**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 account = 0;   cout<<"Initial account balance = "<<account<<endl<<endl;   while(1){  int n;  cout<<"Enter the number you wish to insert in the dynamic table: ";  cin>>n;  account += 3; //adding 3 in account   if(count<size){  \*(p+count) = n;  count+=1;  print(p, count);  account-=1;  }else{  cout<<"Double"<<endl;  int \*new\_arr = new int[size\*2];  for(int i =0; i<count; i++){  new\_arr[i] = \*(p+i);  }  account-=count;  p = new\_arr;  \*(p+count) = n;  size\*=2;  count+=1;  account-=1;  print(p, count);  }  cout<<"Account balance: "<<account<<endl;  cout<<endl;  }   return 0; } |
| --- |

Output:

| Initial account balance = 0  Enter the number you wish to insert in the dynamic table: 5 5  Account balance: 2  Enter the number you wish to insert in the dynamic table: 10 Double 5 10  Account balance: 3  Enter the number you wish to insert in the dynamic table: 2 Double 5 10 2  Account balance: 3  Enter the number you wish to insert in the dynamic table: 19 5 10 2 19  Account balance: 5  Enter the number you wish to insert in the dynamic table: 5 Double 5 10 2 19 5  Account balance: 3  Enter the number you wish to insert in the dynamic table: 23 5 10 2 19 5 23  Account balance: 5 |
| --- |

**Observations**:

The account balance did not fall below zero even once while we entered values in the dynamic table.

**Conclusion**:

Intuitively, we can consider the accounting method as “saving for a rainy day.” The idea is to allocate a fixed cost d for each step of the algorithm. Low-cost calls will accrue “money” to be able to pay for more expensive calls. the amount in “the bank” must never drop below 0, during any step of the algorithm. As long as this holds, we know that our amortized cost of ˆci per operation is a valid amortized cost of the operationand 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>

<http://www2.hawaii.edu/~nodari/teaching/s16/notes/notes01.pdf>