Amortized Analysis

Dr. Bibhudatta Sahoo

Communication & Computing Group

 # CS215, Department of CSE, NIT Rourkela

Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

Amortized analysis

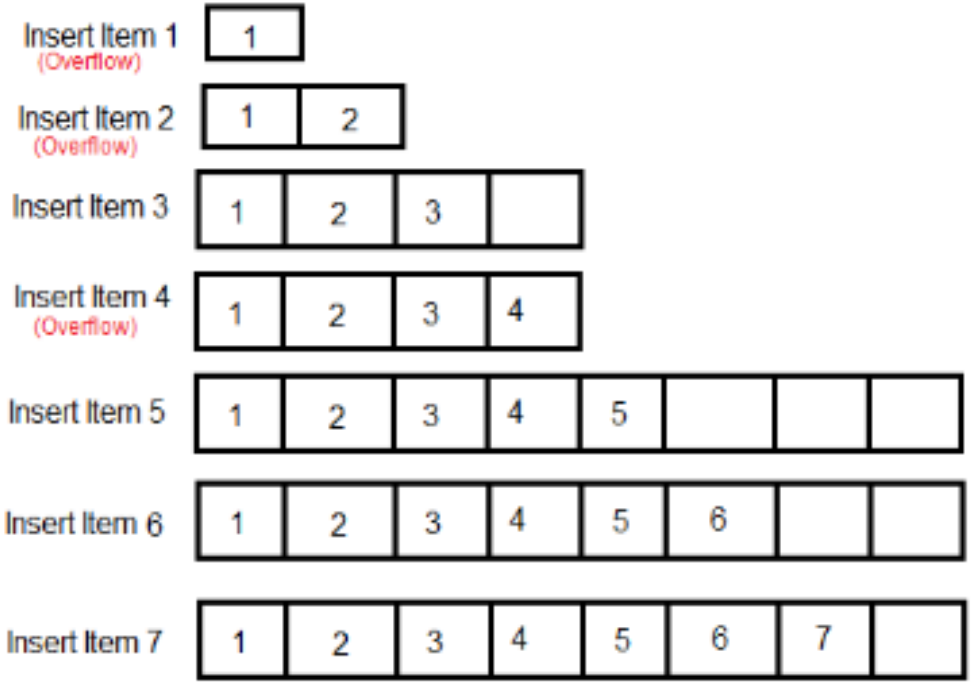
- Amortized analysis is a technique used in computer science to determine the **average time complexity** of a **sequence of operations over a data structure**.
- Unlike standard worst-case analysis, which looks at the time taken by a single operation, amortized analysis provides a way to guarantee that the **average performance** of each operation is efficient, even if a single operation might be expensive.
- Amortized analysis is useful because it provides a more accurate and realistic measure of the performance of an algorithm over a sequence of operations.

Amortized analysis

- Amortized analysis is a technique used in computer science to analyze the average-case time complexity of algorithms that perform a sequence of operations, where some operations may be more expensive than others.
- The idea is to spread the cost of these expensive operations over multiple operations, so that the average cost of each operation is constant or less.
- In an ***amortized analysis***, the time required to perform a sequence of data-structure operations is averaged over all the operations performed.
- Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation might be expensive.
- Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the *average performance of each operation in the worst case*.

The hash table insertions

Initially table is empty and size is 0



Next overflow would happen when we insert 9, table size would become 16

Item No.	1	2	3	4	5	6	7	8	9	10	.....
Table Size	1	2	4	4	8	8	8	8	16	16	.....
Cost	1	2	3	1	5	1	1	1	9	1	.....

$$\text{Amortized Cost} = \frac{(1 + 2 + 3 + 5 + 1 + 1 + 9 + 1 \ldots)}{n}$$

We can simplify the above series by breaking terms 2, 3, 5, 9.. into two as (1+1), (1+2), (1+4), (1+8)

$$\text{Amortized Cost} = \frac{[\overbrace{(1 + 1 + 1 + 1 \ldots)}^{n \text{ terms}} + \overbrace{(1 + 2 + 4 + \ldots)}^{\lfloor \text{Log}_2(n-1) \rfloor + 1 \text{ terms}}]}{n}$$

$$<= \frac{[n + 2n]}{n}$$

$$<= 3$$

$$\text{Amortized Cost} = O(1)$$

Dynamic Table scheme has O(1) insertion time

Characteristics of Amortize Analysis

[1] Averaging over Operations: Instead of analyzing the worst-case time for a single operation, amortized analysis averages the time over a sequence of operations. This approach ensures that, although some operations might take longer, the average time per operation remains low.

[2] Types of Amortized Analysis:

- **Aggregate Method**: Calculates the total cost for n operations and divides by n to get the average cost per operation.
- **Accounting Method**: Assigns different "charges" to operations (often charging more than the actual cost to some and storing the extra as credit) so the average cost per operation reflects the overall efficiency. **Potential Method**: Uses a "potential function" to keep track of the "stored energy" in the data structure. This potential is used to cover the cost of expensive operations, ensuring the average cost remains low.

Characteristics of Amortize Analysis

[3] Focus on Sequences of Operations: Amortized analysis looks at a series of operations as a whole, which is useful for operations that occasionally require more resources but, on average, perform efficiently.

[4] Reduces Perceived Cost of Expensive Operations: Amortized analysis often applies to data structures like dynamic arrays and splay trees, where certain operations (like resizing or rebalancing) are costly but happen infrequently.

[5] Useful for Dynamic Data Structures: Amortized analysis is frequently applied to data structures that grow or shrink dynamically, as it helps demonstrate their average performance over time, regardless of infrequent costly adjustments.

Amortized Analysis: Data structures

- Data structures typically support several different types of operations, each with its own cost (e.g., time cost or space cost).
- The idea behind amortized analysis is that, even when expensive operations must be performed, it is often possible to get away with performing them rarely, so that the average cost per operation is not so high.
- It is important to realize that these "average costs" are not expected values; there needn't be any random events.
- Instead, we are considering the worst-case average cost per operation in a sequence of many operations.
  - In other words, we are interested in the asymptotic behavior of the function **C(n) = 1/n . (worst-case total cost of a sequence of n operations)**
- (possibly with some condition on how many times each type of operation may occur). "Worst-case" means that no adversary could choose a sequence of n operations that gives a worse running time.

Amortized Analysis:

- Not just consider one operation, but a sequence of operations on a given data structure.
- Average cost over a sequence of operations.

Probabilistic analysis:

- Average case running time: average over all possible inputs for one algorithm (operation).
- If using probability, called expected running time.

Amortized analysis:

- No involvement of probability
- Average performance on a sequence of operations, even some operation is expensive.
- Guarantee average performance of each operation among the sequence in worst case.

Three Methods of Amortized Analysis

[1] Aggregate analysis:

- Total cost of n operations/n,

[2] Accounting method:

- Assign each type of operation an (different) amortized cost
- overcharge some operations,
- store the overcharge as credit on specific objects,
- then use the credit for compensation for some later operations.

[3] Potential method:

- Same as accounting method
- But store the credit as "potential energy" and as a whole.

Example 1: The Aggregate Method

Example: Dynamic Array Doubling

Consider a dynamic array that starts with an initial capacity of 1. Whenever the array is full, it doubles in size to accommodate additional elements. Let's analyze the cost of inserting n elements into this dynamic array.

1. **Cost Breakdown for Each Operation:**
   - Most insertions are simple and take $O(1)$ time since they only require placing the element in an available spot.
   - However, when the array is full, it requires a resizing operation, which involves creating a new array twice the size and copying all elements from the old array to the new array. This copying operation is costly and takes $O(k)$ time, where k is the current number of elements.

Example 1: The Aggregate Method

2. **Sequence of Operations:**
   - Insert element 1 (no resizing required, cost = 1).
   - Insert element 2 (array is full, resize to capacity 2, cost = 1 (insert) + 1 (copy) = 2).
   - Insert element 3 (array is full, resize to capacity 4, cost = 1 (insert) + 2 (copy) = 3).
   - Insert element 4 (no resizing needed, cost = 1).
   - Insert element 5 (array is full, resize to capacity 8, cost = 1 (insert) + 4 (copy) = 5).

   Following this pattern, each time the array is full, the cost increases due to the copying operation.

3. **Total Cost Calculation:**
   - For $n = 8$ inserts, we encounter costs during each doubling (1, 2, 4, 8, ...).
   - The total cost for the sequence of n insertions includes all individual insertions plus the cumulative cost of resizing.

4. **Amortized Cost:**
   - The total cost for inserting n elements can be shown to be $O(n)$, even though resizing operations occasionally take $O(k)$ time.
   - When we divide the total cost by n, the amortized cost per insertion remains $O(1)$.

Example 2: The Aggregate Method

- Inserting n elements into a binary min-heap

Cost of Each Insertion

For each element we insert:

1. In the worst case, it may need to travel all the way up from the last level of the heap to the root.
2. This worst-case "bubbling up" cost for an element at height k is $O(\log n)$.

Aggregate Analysis of All Insertions

To compute the total cost of inserting n elements into the heap, let's analyze how often each level of the heap participates in the "bubbling up" process:

1. **Levels in a Heap**:
   - A binary heap of n elements has $O(\log n)$ levels (from the root at level 0 to the deepest level).
2. **Contribution of Each Level to Total Cost**:
   - Only a few elements reach the upper levels of the heap, and thus "bubbling up" costs vary depending on the level of insertion.
   - For example, approximately half of the elements are at the last level, a quarter at the second-to-last level, and so on.

Example 2: The Aggregate Method

3. **Summing Up the Costs:**
    - Let's sum the costs by counting how often each level is involved in the bubbling up for all n insertions.

The total cost can be broken down as follows:

$$\text{Total Cost} = \sum_{k=0}^{\log n} (\text{Number of nodes at level } k \times \text{Cost of bubbling up from level } k)$$

Since half of the nodes are at the last level and take only 1 comparison, a quarter of the nodes are at the second-to-last level and take 2 comparisons, and so forth, the sum simplifies to:

$$\text{Total Cost} = O(n)$$

4. **Amortized Cost per Insertion:**
    - Dividing the total cost by n, we find that the amortized cost of each insertion is $O(1)$.

Conclusion

Using the Aggregate Method, we conclude that the amortized cost of inserting each element into a binary heap is $O(1)$, even though individual insertions may take $O(\log n)$ time due to "bubbling up." This is because the overall work is spread out across all insertions, making the average cost per insertion constant in the long run.

Example 3: The Accounting Method

The **Accounting Method** for amortized analysis assigns an "amortized cost" (or "charge") to each operation, sometimes setting it higher than the actual cost. Any surplus is stored as a "credit," which can then be used to offset more expensive operations. This approach helps maintain a balance, showing that even though some operations may be costly, they are infrequent and covered by the surplus accumulated from other operations.

Example: Dynamic Array Doubling Using the Accounting Method

Consider a dynamic array that starts with an initial capacity of 1. When the array is full, it doubles in size to accommodate additional elements. Let's analyze the cost of inserting n elements into this dynamic array using the Accounting Method.

Example 3: The Accounting Method

1. **Cost of Operations:**
   - Most insertions take $O(1)$ time since they only require adding an element to an available position.
   - However, when the array is full, it must double in size, which requires $O(k)$ time for copying k elements, where k is the current number of elements in the array before resizing.
2. **Assigning Amortized Costs:**
   - Assign an **amortized cost** of 3 units for each insertion.
   - For most insertions (when resizing is not required), this 3-unit charge covers the $O(1)$ insertion cost, leaving a surplus (credit).
   - The surplus is stored as a credit and will be used to pay for the more expensive resizing operations.
3. **Tracking Credits:**
   - Every time we insert an element without resizing, we charge 3 units but only use 1 unit, leaving a credit of 2 units.
   - When we eventually need to resize (double the array size), we use these accumulated credits to cover the cost of copying existing elements to the new array.

Example 3: The Accounting Method

3. **Tracking Credits:**
   - Every time we insert an element without resizing, we charge 3 units but only use 1 unit, leaving a credit of 2 units.
   - When we eventually need to resize (double the array size), we use these accumulated credits to cover the cost of copying existing elements to the new array.
4. **Using Credits for Resizing:**
   - When resizing is triggered, say after reaching a size of k, we need to copy all k elements to the new, larger array.
   - By this point, we have accumulated enough credits: each of the k elements stored in the array contributed 2 units in credit, providing $2 \times k$ units, which exactly covers the $O(k)$ resizing cost.
5. **Amortized Cost of Each Insertion:**
   - Since each insertion operation was charged an amortized cost of 3 units, and the credits from these charges cover the occasional expensive resize operation, we can conclude that the amortized cost per insertion remains $O(1)$ despite the occasional costly resize.

Example 3: The Accounting Method

Summary

Using the Accounting Method:

- Each insertion is charged an amortized cost of 3 units.
- The credits generated by this overcharge cover the costly resize operations, ensuring that the average cost per insertion remains constant.
- This shows that, even with periodic resizing, the amortized cost per insertion operation is O(1).

Example 4: The Accounting Method

- **Accounting Method** for amortized analysis, using **stack operations** with an auxiliary "multi-pop" operation, which removes multiple elements from the stack at once. Goal is to determine the amortized cost per operation using the Accounting Method.

Problem Setup

Suppose we have a stack that supports the following operations:

1. **Push(x)**: Pushes an element x onto the stack.
2. **Pop()**: Removes the top element from the stack.
3. **Multi-pop(k)**: Removes the top k elements from the stack or all elements if the stack has fewer than k elements.

Each operation has the following costs:

- **Push** and **Pop** have an actual cost of $O(1)$.
- **Multi-pop** has an actual cost of $O(m)$, where m is the number of elements removed (which could be up to k).

Example 4: The Accounting Method

Solution Using the Accounting Method

1. **Assign Amortized Costs:**
   - We assign an **amortized cost of 2 units** for each **Push** operation.
   - We assign an **amortized cost of 0 units** for **Pop** and **Multi-pop** operations.
2. **Charging and Storing Credits:**
   - Each **Push** operation takes an actual cost of $O(1)$ but is charged 2 units.
   - One of these units covers the actual cost of the Push, and the other unit is stored as a **credit** on the pushed element.
   - This credit will later be used to cover the cost of a **Pop** or **Multi-pop** operation.
3. **Using Credits for Pops:**
   - When we perform a **Pop** or **Multi-pop**, we remove elements from the stack.
   - Since each element already has one unit of credit stored on it (from the Push), we can use this credit to cover the actual $O(1)$ cost of removing each element.
   - Therefore, **each Pop or Multi-pop operation is effectively free in terms of amortized cost** since they're covered by the credits from earlier Pushes.

Example 4: The Accounting Method

4. **Amortized Cost per Operation**:
    - Each **Push** operation has an amortized cost of 2 units, which includes the actual cost of the Push and the credit stored for future Pops.
    - **Pop** and **Multi-pop** operations have an amortized cost of 0 units because they are paid for by the credits accumulated during Push operations.

Conclusion

Using the Accounting Method, we conclude:

- The amortized cost of **Push** is $O(1)$ (specifically 2 units).
- The amortized cost of **Pop** and **Multi-pop** is $O(1)$, effectively 0 units due to the credits from the Push operations.

This analysis shows that, on average, each operation has a constant $O(1)$ amortized cost, even though some operations (like Multi-pop) may occasionally require removing multiple elements. The stored credits ensure that the cost of these operations is balanced across the sequence.

Example 5: The Potential method

The **Potential Method** in amortized analysis involves defining a "potential function" that tracks the "stored energy" in a data structure. This stored energy (or potential) is used to account for the cost of future expensive operations, ensuring that the average cost per operation remains low.

Example: Dynamic Array Doubling Using the Potential Method

Consider a dynamic array that starts with an initial capacity of 1 and doubles in size whenever it becomes full. We want to analyze the cost of inserting n elements into this dynamic array.

1. **Define the Potential Function**:
   - Let the potential function Φ represent the "extra capacity" available in the array. We can define Φ as:

$$\Phi = \text{Number of unused slots in the array}$$

   - When the array is exactly full, $\Phi = 0$. But after doubling, half of the array will be unused, so Φ increases, storing potential energy for future operations.

Example 5: The Potential method

2. **Compute the Amortized Cost of Each Insertion**:
   - Each insertion operation has an actual cost of $O(1)$ if no resizing is needed.
   - If resizing is needed, we must double the array size, which requires copying all existing elements to the new array. This resizing cost is $O(k)$ for k elements.

   Using the potential function, we calculate the amortized cost for each insertion as:

$$\text{Amortized Cost} = \text{Actual Cost} + \Delta\Phi$$

   where $\Delta\Phi$ is the change in the potential function.

3. **Analysis of Costs and Potential Changes**:
   - **Insertion Without Resizing**:
     - The actual cost is $O(1)$.
     - The potential function Φ decreases by 1 because we used one previously unused slot.
     - Therefore, $\Delta\Phi = -1$, making the amortized cost:

$$\text{Amortized Cost} = 1 + (-1) = 1$$

Example 5: The Potential method

- **Insertion With Resizing:**
  - Suppose we currently have k elements, so the array is full, and we need to resize it to a capacity of $2k$.
  - The actual cost for resizing is $O(k)$ (copying k elements).
  - After resizing, we have k unused slots, so the potential function Φ increases by k, meaning $\Delta\Phi = k$.
  - The amortized cost in this case is:

$$\text{Amortized Cost} = k + k = 2k$$

4. **Amortized Cost Per Operation:**
   - For all subsequent operations, the potential increase accumulated during resizing helps cover future costs.
   - Since doubling only happens occasionally (after every power of 2 insertions), the amortized cost per insertion remains $O(1)$, even though resizing itself costs $O(k)$ in those rare instances.

Example 5: The Potential method

4. **Amortized Cost Per Operation**:
    - For all subsequent operations, the potential increase accumulated during resizing helps cover future costs.
    - Since doubling only happens occasionally (after every power of 2 insertions), the amortized cost per insertion remains $O(1)$, even though resizing itself costs $O(k)$ in those rare instances.

Conclusion

By using the Potential Method, we:

- Defined a potential function that tracks the unused capacity.
- Showed that most insertions cost $O(1)$ amortized, while occasional resizing costs $O(k)$ but is offset by the increase in potential.
- Concluded that the amortized cost of each insertion is $O(1)$, thanks to the potential function storing enough energy for future expensive operations.

Example 6: The Potential method

Potential Method with a **stack data structure** that includes a "double" operation.

Problem Setup

Suppose we have a stack with the following operations:

1. **Push(x)**: Pushes an element x onto the stack.
2. **Pop()**: Removes the top element from the stack.
3. **Double()**: Doubles every element in the stack (e.g., if the stack contains $[1, 2, 3]$, after the Double operation it becomes $[2, 4, 6]$).

- **Push** and **Pop** operations both have an actual cost of $O(1)$.
- The **Double** operation costs $O(k)$, where k is the number of elements in the stack, as each element needs to be accessed and doubled.

Goal

We want to determine the amortized cost of each operation using the Potential Method.

Example 6: The Potential method

1. **Define the Potential Function:**
   - Let the potential function Φ represent the "extra potential energy" in the stack based on the number of elements in the stack.
   - Define Φ as:

$$\Phi = \text{number of elements in the stack}$$

   - This potential function increases with each Push operation, storing energy for future Double operations. The potential decreases with each Pop operation.
2. **Compute the Amortized Cost of Each Operation:**
   - The amortized cost of each operation is given by:

$$\text{Amortized Cost} = \text{Actual Cost} + \Delta\Phi$$

   - Here, $\Delta\Phi$ is the change in the potential function due to the operation.
3. **Analysis of Each Operation:**
   - **Push(x):**
     - The actual cost of Push is $O(1)$.
     - This operation increases the potential by 1 because we add an element to the stack.
     - Thus, $\Delta\Phi = 1$.
     - The amortized cost is:

$$\text{Amortized Cost} = 1 + 1 = 2$$

Example 6: The Potential method

- **Pop():**
  - The actual cost of Pop is $O(1)$.
  - This operation decreases the potential by 1 because we remove an element from the stack.
  - Thus, $\Delta\Phi = -1$.
  - The amortized cost is:

$$\text{Amortized Cost} = 1 + (-1) = 0$$

  - This means the amortized cost of a Pop operation is effectively zero, as it's "paid for" by the potential accumulated during Push operations.
- **Double():**
  - The actual cost of Double is $O(k)$, where k is the number of elements in the stack.
  - This operation doesn't change the number of elements, so the potential function remains the same, and $\Delta\Phi = 0$.
  - The amortized cost is:

$$\text{Amortized Cost} = k + 0 = k$$

  - However, since the Push operations have built up enough potential by adding 1 unit of potential for each element, this stored potential will help pay for the Double operation when it happens.

Example 6: The Potential method

4. **Amortized Cost per Operation:**
    - Each Push has an amortized cost of 2, which accumulates potential in the system.
    - Each Pop has an amortized cost of 0, as it's paid for by the potential.
    - The occasional Double operation has an amortized cost of $O(k)$, but since it's infrequent, the average cost per operation remains $O(1)$ over a sequence of operations.

Conclusion

Using the Potential Method:

- We defined a potential function that grows with each Push and decreases with each Pop.
- The accumulated potential is used to offset the cost of the infrequent Double operation.
- The amortized cost of each operation (Push, Pop, and Double) remains $O(1)$ on average, demonstrating that the stack operations are efficient even with the occasional Double.

Why Amortized analysis ?

1. Reflects Average Performance Over Time

- Standard worst-case analysis might give a misleading impression of an algorithm's performance if it focuses on a single, rare, expensive operation. Amortized analysis helps demonstrate that while some operations may be expensive, they occur infrequently, leading to an efficient average performance.
- For example, in dynamic array resizing (e.g., array doubling when inserting elements), resizing is costly, but it doesn't happen often. Amortized analysis shows that the average cost of insertion remains O(1).

2. Ensures Efficient Resource Management

- Many data structures involve operations that can vary in cost. By using amortized analysis, developers can ensure that the design of their data structure will not lead to unexpectedly high resource usage over time, even if individual operations sometimes take longer than average.

Why Amortized analysis ?

3. Useful for Optimizing Complex Data Structures

- Amortized analysis is particularly helpful when working with data structures where some operations can lead to others being faster or slower. It provides insights into optimizing data structures to maintain efficiency across all operations.
- For instance, in splay trees (a type of self-adjusting binary search tree), accessing nodes occasionally involves costly rotations. However, amortized analysis can show that the overall cost per operation remains O(log

4. Enables Smarter Algorithm Design

- Algorithms that are designed with amortized analysis in mind can handle a mix of operations more effectively, by balancing the cost of expensive operations with cheaper ones. This approach allows for clever design strategies that might not be evident from a worst-case perspective.
- For example, a garbage collection algorithm might occasionally perform a full sweep of memory, which is expensive. However, amortized analysis helps design systems where this cost is spread over many small operations, leading to smoother performance.

Why Amortized analysis ?

5. More Practical Than Worst-Case Analysis Alone

1. Worst-case analysis can overestimate the cost of algorithms, making them seem less efficient than they actually are in practice. Amortized analysis, by focusing on the average cost per operation, gives a better sense of how an algorithm performs in typical use, leading to more accurate assessments.
2. For example, if a worst-case analysis shows that an operation can take O(n) time, but this happens only once every n operations, amortized analysis can show that the cost per operation is still O(1).

Some common applications where amortized analysis

[1] Dynamic Arrays (e.g., Array List, Vector)

- **Resizing Mechanism**: When elements are added to a dynamic array, it occasionally needs to resize (e.g., doubling its size). Although resizing is expensive, amortized analysis shows that the average cost of adding an element remains O(1) because the cost is spread out over many operations.

[2] Hash Tables

- **Rehashing**: When a hash table becomes full, it needs to expand its capacity and redistribute elements (rehashing). Amortized analysis demonstrates that, despite occasional expensive rehashing, the average insertion or lookup operation remains O(1).

Some common applications where amortized analysis

[3] Binary Counter Increment Counting Bits:

- Incrementing a binary counter (e.g., counting in binary) can sometimes cause multiple bits to flip. Amortized analysis shows that the average cost of an increment operation is O(1) even though some increments are more expensive.

[4] Stack Operations (with Multipop)Push, Pop, and Multipop:

- If a stack supports a multipop operation that can pop multiple elements at once, amortized analysis can show that even though a multipop can be expensive, the average cost per operation over a sequence of push, pop, and multipop operations is still O(1).

Some common applications where amortized analysis

[5] Union-Find Data Structure (Disjoint Set)Union by Rank and Path Compression:

- Amortized analysis is used to show that the union and find operations, when combined with optimizations like path compression and union by rank, have an almost constant amortized time of $O(\alpha(n))$, where α is the inverse Ackermann function (which grows extremely slowly).

[6] Splay TreesSelf-Adjusting Binary Search Trees:

- In splay trees, elements are rotated to the root every time they are accessed. Amortized analysis shows that despite some rotations being expensive, the average access time is O(logn).

Some common applications where amortized analysis

[7] Fibonacci HeapsPriority Queues and Dijkstra's Algorithm:

- Fibonacci heaps support operations like insert, decrease-key, and merge in amortized $O(1)$O(1) time, while extract-min operates in $O(\log n)$O(logn). This makes them suitable for algorithms like Dijkstra's and Prim's, where repeated decrease-key operations are required.

[8] Balanced Trees (e.g., AVL, Red-Black Trees)Tree Rotations:

- When inserting or deleting elements, balanced trees may need to perform rotations to maintain their balance. Amortized analysis helps in understanding the average time taken to maintain the balance over a sequence of operations, showing that they remain $O(\log n)$O(logn).9. Garbage

Some common applications where amortized analysis

[9] Garbage Collection in Programming Languages

- **Memory Management**: Generational garbage collectors perform memory cleanup when needed, which can occasionally be expensive. Amortized analysis can be used to demonstrate that the average time for memory allocation and deallocation remains efficient, even though individual garbage collection cycles might take longer.

[10] Scheduling Algorithms (e.g., Task Queues)

- **Job Scheduling**: In systems where tasks are scheduled and occasionally need reordering or rebalancing (e.g., load balancing), amortized analysis can be applied to show that the average scheduling time per task remains efficient.

Some common applications where amortized analysis

[11] Persistent Data Structures

- **Functional Programming**: Data structures like persistent trees or queues, which allow access to previous versions, might involve copying or rebalancing. Amortized analysis helps in understanding the overall time efficiency for operations across different versions.

[12] Data Streams and Online Algorithms

- **Sliding Window Algorithms**: In applications where data is continuously processed (e.g., tracking running averages or medians), amortized analysis can show that the average cost of processing each element remains low, even if some steps are more computationally expensive than others.

Conclusions

- Amortized costs can provide a clean abstraction of data-structure performance.
- Any of the analysis methods can be used when an amortized analysis is called for, but each method has some situations where it is arguably the simplest or most precise.
- Different schemes may work for assigning amortized costs in the accounting method, or potentials in the potential method, sometimes yielding radically different bounds.
- Amortized analysis is particularly powerful when analyzing algorithms and data structures with fluctuating costs, offering a clearer view of average-case efficiency.


Thank you for your attention

References

- https://www.geeksforgeeks.org/introduction-to-amortized-analysis/

Exercises

Exercises

1. Explain the difference between worst-case analysis and amortized analysis. Provide an example where amortized analysis is more appropriate than worst-case analysis.
2. Describe the three techniques used in amortized analysis: aggregate method, accounting method, and potential method. Give a brief example of each.
3. Why is amortized analysis particularly useful for data structures that involve dynamic resizing or restructuring?

[1] Sample question with the answer:

Consider a dynamic array that starts with a capacity of 1. Whenever the array becomes full, it doubles its capacity. Suppose you perform a sequence of 8 insert operations on this array, starting from an empty state.

1. What is the total cost of all 8 insert operations if each insertion takes $O(1)$ time, except for when resizing happens, which costs $O(n)$, where n is the number of elements currently in the array?
2. What is the amortized cost per insertion?

[1] Sample question with the answer :

Answer:

1. **Total Cost Analysis:**
   - **Step-by-Step Cost Calculation:**
     - Insert 1st element: No resizing needed, cost = 1.
     - Insert 2nd element: Array is full, resize to capacity 2 (cost of resizing = 1, copying 1 element), then insert, cost = 1 + 1 = 2.
     - Insert 3rd element: No resizing needed, cost = 1.
     - Insert 4th element: Array is full, resize to capacity 4 (cost of resizing = 2, copying 2 elements), then insert, cost = 2 + 1 = 3.
     - Insert 5th element: No resizing needed, cost = 1.
     - Insert 6th element: No resizing needed, cost = 1.
     - Insert 7th element: No resizing needed, cost = 1.
     - Insert 8th element: Array is full, resize to capacity 8 (cost of resizing = 4, copying 4 elements), then insert, cost = 4 + 1 = 5.
   - **Total Cost:**

$$1 + 2 + 1 + 3 + 1 + 1 + 1 + 5 = 15$$

[1] Sample question with the answer :

2. **Amortized Cost Per Insertion:**
   - We performed 8 insertions, with a total cost of 15.
   - The amortized cost per insertion:
   $$\frac{15}{8} = 1.875$$
   - Using amortized analysis, we can also argue that every insertion operation has a constant cost of $O(1)$ plus the occasional resizing. Even with resizing, the cost of inserting n elements ends up being $O(n)$, leading to an average (amortized) cost of $O(1)$ per insertion.

Conclusion:

The amortized cost per insertion is $O(1)$. This shows that even though some insertions involve costly resizing operations, the overall average cost remains efficient.

Excercise

- A sequence of stack operations is performed on a stack whose size never exceeds k. After every k operations, a copy of the entire stack is made for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

Any Question?

Thanks for your attendance & attention!