Today’s Lecture

- Amortized Analysis
  - aggregate method
  - accounting method
  - potential method
Motivations from MST:

- Expensive operations:
  - Sort edges: $O(|E| \cdot \lg |E|)$
  - $O(|V|)$: MakeSet()
  - $O(|E|)$: FindSet()
  - $O(|V|)$: Union()

- Upshot:
  - Comes down to efficiency of disjoint-set operations, particularly Union()
Unions:

- Worst-case analysis: \( O(n^2) \) time for \( n \) Union’s
  
  \[
  \begin{align*}
  \text{Union}(S_1, S_2) & \quad \text{“copy”} \quad 1 \text{ element} \\
  \text{Union}(S_2, S_3) & \quad \text{“copy”} \quad 2 \text{ elements} \\
  \vdots \\
  \text{Union}(S_{n-1}, S_n) & \quad \text{“copy”} \quad n-1 \text{ elements} \\
  \end{align*}
  \]

- Improvement: always copy smaller into larger
  
  \(-\) How long would above sequence of Union’s take?
  
  \(-\) Worst case: \( n \) Union’s take \( O(n \lg n) \) time
  
  \(-\) Proof uses amortized analysis!

Review:

Amortized Analysis of Disjoint Sets

- If elements are copied from the smaller set into the larger set, an element can be copied at most \( \lg n \) times
  
  \(-\) Worst case:
    
    \[
    \begin{align*}
    1^{\text{st}} \text{ time} & \quad \text{resulting set size} \quad \geq 2 \\
    2^{\text{nd}} \text{ time} & \quad \geq 4 \\
    \vdots \\
    (\lg n)^{\text{th}} \text{ time} & \quad \geq n
    \end{align*}
    \]
Review:
Amortized Analysis of Disjoint Sets

- Since we have $n$ elements each copied at most “$\lg n$” times, “$n$” Union( )’s takes $O(n \lg n)$ time
- Therefore we say the amortized cost of a Union() operation is $O(\lg n)$
- This is the aggregate method of amortized analysis:
  - $n$ operations take time $T(n)$
  - Average cost of an operation = $T(n)/n$

Amortized Analysis: Accounting Method

- **Accounting method**
  - Charge each operation an amortized cost
  - Amount not used stored in “bank”
  - Later operations can use stored money
  - Balance must not go negative
Accounting Method Example:
Dynamic Tables

- Implementing a table (e.g., hash table) for dynamic data, want to make it small as possible
- Problem: if too many items inserted, table may be too small
- Idea: allocate more memory as needed

Dynamic Tables

1. Init table size $m = 1$
2. Insert elements until number $n > m$
3. Generate new table of size $2m$
4. Reinsert old elements into new table
5. (back to step 2)
- What is the worst-case cost of an insert?
- One insert can be costly, but the total?
Analysis Of Dynamic Tables

- Let $c_i = \text{cost of } i^{th} \text{ insert}$
- $c_i = i$ if $i-1$ is exact power of 2, 1 otherwise
- Example:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Table Size</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert(1)</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Table Size</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert(2)</td>
<td>2</td>
<td>1 + 1</td>
</tr>
</tbody>
</table>
Analysis Of Dynamic Tables

- Let $c_i =$ cost of $i$\textsuperscript{th} insert
- $c_i = i$ if $i \cdot 1$ is exact power of 2, 1 otherwise

Example:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Table Size</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert(1)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Insert(2)</td>
<td>2</td>
<td>1 + 1</td>
</tr>
<tr>
<td>Insert(3)</td>
<td>4</td>
<td>1 + 2</td>
</tr>
</tbody>
</table>

09/25/02
Analysis Of Dynamic Tables

- Let \( c_i = \) cost of \( i^{th} \) insert
- \( c_i = i \) if \( i-1 \) is exact power of 2, 1 otherwise
- Example:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Table Size</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert (1)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Insert (2)</td>
<td>2</td>
<td>1 + 1</td>
</tr>
<tr>
<td>Insert (3)</td>
<td>4</td>
<td>1 + 2</td>
</tr>
<tr>
<td>Insert (4)</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Insert (5)</td>
<td>8</td>
<td>1 + 4</td>
</tr>
<tr>
<td>Insert (6)</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>
## Analysis Of Dynamic Tables

- Let $c_i$ = cost of $i^{th}$ insert
- $c_i = i$ if $i-1$ is exact power of 2, 1 otherwise
- Example:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Table Size</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert (1)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Insert (2)</td>
<td>2</td>
<td>$1 + 1$</td>
</tr>
<tr>
<td>Insert (3)</td>
<td>4</td>
<td>$1 + 2$</td>
</tr>
<tr>
<td>Insert (4)</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Insert (5)</td>
<td>8</td>
<td>$1 + 4$</td>
</tr>
<tr>
<td>Insert (6)</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Insert (7)</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Insert (8)</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>
Analysis Of Dynamic Tables

- Let $c_i =$ cost of $i^{th}$ insert
- $c_i = i$ if $i-1$ is exact power of 2, 1 otherwise
- Example:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Table Size</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Insert}(1)$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\text{Insert}(2)$</td>
<td>2</td>
<td>1 + 1</td>
</tr>
<tr>
<td>$\text{Insert}(3)$</td>
<td>4</td>
<td>1 + 2</td>
</tr>
<tr>
<td>$\text{Insert}(4)$</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>$\text{Insert}(5)$</td>
<td>8</td>
<td>1 + 4</td>
</tr>
<tr>
<td>$\text{Insert}(6)$</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>$\text{Insert}(7)$</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>$\text{Insert}(8)$</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>$\text{Insert}(9)$</td>
<td>16</td>
<td>1 + 8</td>
</tr>
</tbody>
</table>

Aggregate Analysis

- $n$ Insert( ) operations cost
  \[
  \sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\log n} 2^j = n + (2n - 1) < 3n
  \]
- Average cost of operation
  \[
  = (\text{total cost})/(\# \text{ operations}) < 3
  \]
- Asymptotically, then, a dynamic table costs the same as a fixed-size table
  - Both $O(1)$ per Insert operation
Accounting Analysis

- Charge each operation $3 amortized cost
  - Use $1 to perform immediate Insert()
  - Store $2
- When table doubles
  - $1 reinserts old item, $1 reinserts another old item
  - Point is, we’ve already paid these costs
  - Upshot: constant (amortized) cost per operation

Review: Accounting Analysis

- Suppose must support insert & delete, table should contract as well as expand
  - Table overflows ⇒ double it (as before)
  - Table < 1/4 full ⇒ halve it
  - Charge $3 for Insert (as before)
  - Charge $2 for Delete
    - Store extra $1 in emptied slot
    - Use later to pay to copy remaining items to new table when shrinking table
- What if we halve size when table < 1/8 full?
Stack Operations

- Goal
  - find worst case time, $T(n)$, for $n$ operations
  - the amortized cost is $T(n)/n$
- Push and Pop
  - Push($x$, $S$) has complexity $O(1)$
  - Pop($S$) returns popped object, complexity is $O(1)$
- Multi-pop operation

```plaintext
MULTIPOP(S, k)
1 while not STACK-EMPTY(S) and k ≠ 0
2 do Pop(S)
3 k ← k − 1
```

- complexity is min($|S|, k$) where $|S|$ is the stack size

A Simple Analysis

- Start with empty stack
  - Stack can be no larger than $O(n)$
  - So a multi-pop operation could have complexity $O(n)$
  - Since there are $n$ operations, an upper bound on complexity is $O(n^2)$
- Although this is a valid upper bound, it grossly overestimates the upper bound
An Amortized Analysis

- Claim - any sequence of push, pop, multi-pop has at most worst case complexity of \( O(n) \)
  - each object can be popped at most one time for each time it is pushed
  - the number of push operations is \( O(n) \) at most
  - so the number of pops, either from pop or multi-pop, is at most \( O(n) \)
  - the overall complexity is \( O(n) \)
- The amortized cost is \( O(n)/n = O(1) \)
- Question - would this still be true if we had a multi-push and a multi-pop?

Incrementing Binary Numbers

- Assume bits \( A[k-1] \) to \( A[0] \)
  \[
  x = \sum_{i=0}^{k-1} A[i] \times 2^i
  \]
- Incrementing starting from 0
  
<table>
<thead>
<tr>
<th>Increment(A)</th>
<th>Counter value</th>
<th>Total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2 while ( i &lt; \text{length}[A] ) and ( A[i] = 1 )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3 do ( A[i] \leftarrow 0 )</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4 ( i \leftarrow i + 1 )</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5 if ( i &lt; \text{length}[A] )</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>6 then ( A[i] \leftarrow 1 )</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

09/25/02
A Simple Analysis

- We measure cost as the number of bit flips
  - some operations only flip one bit
  - other operations ripple through the number and flip many bits
  - what is the average cost per operation?

- A cursory analysis
  - worst case for the increment operation is $O(k)$
  - a sequence of $n$ increment operations would have worst case behavior of $O(nk)$

- Although this is an upper bound, it is not very tight

An Amortized Analysis

- Not all bits are flipped each iteration
  - $A[0]$ is flipped every iteration; $A[1]$ every other iteration
  - for $i > \lceil \log n \rceil$, the bit $A[i]$ never flips
  - summing all the bit flips we have

\[
\sum_{i=0}^{\lceil \log n \rceil} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n
\]

- So it turns out the worst case is bounded by $O(n)$
- Therefore $O(n)/n$ is only $O(1)$!
### Stack Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Actual cost</th>
<th>Amortized cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Pop</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Multi-pop</td>
<td>min(</td>
<td>S</td>
</tr>
</tbody>
</table>

#### Rationale
- since we start with an empty stack, pushes must be done first and this builds up the amortized credit
- all pops are charged against this credit; there can never be more pops (of either type) than pushes
- therefore the total amortized cost is \( O(n) \)

---

### Incrementing a Binary Counter

#### Amortized cost
- 2 for setting a bit to 1
- 0 for setting a bit to 0
- the credits for any number are the number of 1 bits

#### Analysis of the increment operation
- the while loop resetting bits is charged against credits
- only one bit is set in line 6, so the total charge is 2
- since the number of 1’s is never negative, the amount of credit is also never negative
- the total amortized cost for \( n \) increments is \( O(n) \)
The Potential Method

- Potential is the accumulation of credits
  - it is associated with the entire data structure rather than individual objects; $\Phi(D_i)$ is a real number associated with the structure $D_i$
  - the amortized cost is $\tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$.
  - the total amortized cost is $\sum_{i=1}^{n} \tilde{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$
  - if we insure that $\Phi(D_1) \geq \Phi(D_0)$
    - then the potential never becomes negative
  - it is often convenient to let $\Phi(D_0) = 0$ then it only needs to be shown that all $\Phi(D_i)$ are nonnegative

Stack Operations - 1

- The potential function is the size of the stack
  - the total amortized cost of $n$ operations with respect to $\Phi$ is an upper bound for the actual cost
    - $\Phi(D_i) \geq 0$
    - $\Phi(D_i) = \Phi(D_0)$

- The push operation $\Phi(D_i) - \Phi(D_{i-1}) = (s + 1) - s = 1$

- The pop operation
  - the difference in potential for the pop operation is -1
  - so the amortized cost is $1 + (-1) = 0$
Stack Operations - 2

\[ \Phi(D_i) - \Phi(D_{i-1}) = -k' \quad \text{where } k' = \min(k, S) \]

so \( \tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \)
\[ = k' - k' \]
\[ = 0 \]

- Total amortized cost
  - Each operation has an amortized cost of \( O(1) \)
  - For \( n \) operations, the total amortized cost is \( O(n) \)
  - Since the potential function meets all of our requirements, the total cost is a valid upper bound

The Physicists View The Potential Function Method

- Idea: Store prepaid credit as a potential energy of the entire system. Use potential to pay for operations which cost more energy than they bring in.
- Formally: Formally let \( \Phi_i \) denote the potential at time \( i \). Clearly
  \[ \overline{c} = c_i + \Delta \Phi \]
- where \( \Delta \Phi_i = \Phi_i - \Phi_{i-1} \).
Consequently:

- thus assume $c_i = b$ the total amortized cost is

$$b \times n = \sum_{i=0}^{n} c_i = \sum_{i=0}^{n} (c_i + \Phi_i - \Phi_{i-1})$$

$$= \Phi_n - \Phi_0 + \sum_{i=0}^{n} c_i$$

---

Incrementing a Binary Counter - 1

- Potential function - the number of 1s in the count after the $i^{th}$ operation
- Amortized cost of the $i^{th}$ increment operation
  - suppose that $t_i$ bits are reset and one bit is set
  $$\Phi(D_i) - \Phi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1}$$
  $$= 1 - t_i$$
  - since the actual cost is $t_i + 1$, we have
  $$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
  $$\leq (t_i + 1) + (1 - t_i)$$
  $$= 2$$
- Therefore, for $n$ operations, the total cost is $O(n)$
Incrementing a Binary Counter - 2

- Suppose the counter is not initially zero
  - there are $b_0$ initial 1s and after $n$ increments $b_n$ 1s
    \[
    \sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \delta_i - \Phi(D_n) + \Phi(D_0)
    \]
  - But the amortized cost for $c$ are each 2, so
    \[
    \sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} 2 - b_n + b_0
    \]
    \[
    = 2n - b_n + b_0
    \]
- Total cost
  
  \[
  \text{since } b_0 < k, \text{ if we executed at least } n = \Omega(k) \text{ increment operations, then the total cost is no more than } O(n) \text{ no matter what the starting value of the counter}
  \]

Using the Potential Method for DT

- The potential function is
  \[
  \Phi(T) = 2 \cdot \text{num}[T] - \text{size}[T]
  \]
- immediately after expansion, $\text{num}[T] = \text{size}[T]/2$ so the potential is 0 (all credits used for expansion)
- immediately before expansion, $\text{num}[T] = \text{size}[T]$ so the potential is $\text{num}[T]$ which will handle the copying
- since $\text{num}[T] \geq \text{size}[T]/2$, the potential in nonnegative
**Relationships between Values**

![Graph showing relationships between values](image)

---

**Argument**

If the $i$th TABLE-INSERT operation does not trigger an expansion, then $size_i = size_{i-1}$ and the amortized cost of the operation is

$$
\bar{c}_i = c_i + \Phi_i - \Phi_{i-1}
$$

$$
= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1})
$$

$$
= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i)
$$

$$
= 3 .
$$

If the $i$th operation does trigger an expansion, then $size_i / 2 = size_{i-1} = num_i - 1$, and the amortized cost of the operation is

$$
\bar{c}_i = c_i + \Phi_i - \Phi_{i-1}
$$

$$
= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1})
$$

$$
= num_i + (2 \cdot num_i - (2 \cdot num_{i-1} - 2)) - (2(num_i - 1) - (num_i - 1))
$$

$$
= num_i + 2 - (num_i - 1)
$$

$$
= 3
$$