Pseudocode

Table-Insert(T,x){
    if (T.size == 0){allocate T with 1 slot;T.size=1}
    if (T.num == T.size){
        allocate newTable with 2*T.size slots;
        insert all items in T.table into newTable;
        T.table = newTable;
        T.size = 2*T.size
    }
    T.table[T.num] = x;
    T.num++
}
**In Class Exercise**

Recall that \( a_i = c_i + \Phi_i - \Phi_{i-1} \)

- Show that this potential function is 0 initially and always nonnegative
- Compute \( a_i \) for the case where Table-Insert does not trigger an expansion
- Compute \( a_i \) for the case where Table-Insert does trigger an expansion (note that \( num_{i-1} = num_i - 1 \), \( size_{i-1} = num_i - 1 \), \( size_i = 2 \times (num_i - 1) \))

**Table Delete**

- We’ve shown that a Table-Insert has \( O(1) \) amortized cost
- To implement Table-Delete, it is enough to remove (or zero out) the specified item from the table
- However it is also desirable to contract the table when the load factor gets too small
- Storage for old table can then be freed to the heap

**Desirable Properties**

We want to preserve two properties:

- the load factor of the dynamic table is lower bounded by some constant
- the amortized cost of a table operation is bounded above by a constant

**Naive Strategy**

- A natural strategy for expansion and contraction is to double table size when an item is inserted into a full table and halve the size when a deletion would cause the table to become less than half full
- This strategy guarantees that load factor of table never drops below \( 1/2 \)
D’Oh

- Unfortunately this strategy can cause amortized cost of an operation to be large
- Assume we perform \( n \) operations where \( n \) is a power of 2
- The first \( n/2 \) operations are insertions
- At the end of this, \( T.num = T.size = n/2 \)
- Now the remaining \( n/2 \) operations are as follows:
  
  \[ I, D, D, I, I, D, D, I, I, \ldots \]

  where \( I \) represents an insertion and \( D \) represents a deletion

Analysis

- Note that the first insertion causes an expansion
- The two following deletions cause a contraction
- The next two insertions cause an expansion again, etc., etc.
- The cost of each expansion and deletion is \( \Theta(n) \) and there are \( \Theta(n) \) of them
- Thus the total cost of \( n \) operations is \( \Theta(n^2) \) and so the amortized cost per operation is \( \Theta(n) \)

The Solution

- The Problem: After an expansion, we don’t perform enough deletions to pay for the contraction (and vice versa)
- The Solution: We allow the load factor to drop below \( 1/2 \)
- In particular, halve the table size when a deletion causes the table to be less than \( 1/4 \) full
- We can now create a potential function to show that Insertion and Deletion are fast in an amortized sense

Recall: Load Factor

- For a nonempty table \( T \), we define the “load factor” of \( T \), \( \alpha(T) \), to be the number of items stored in the table divided by the size (number of slots) of the table
- We assign an empty table (one with no items) size 0 and load factor of 1
- Note that the load factor of any table is always between 0 and 1
- Further if we can say that the load factor of a table is always at least some constant \( c \), then the unused space in the table is never more than \( 1 - c \)
The Potential

\[
\Phi(t) = \begin{cases} 
    2 \times T.\text{num} - T.\text{size} & \text{if } \alpha(T) \geq 1/2 \\
    T.\text{size}/2 - T.\text{num} & \text{if } \alpha(T) < 1/2 
\end{cases}
\]

- Note that this potential is legal since \( \Phi(0) = 0 \) and (you can prove that) \( \Phi(i) \geq 0 \) for all \( i \)

Intuition

- Note that when \( \alpha = 1/2 \), the potential is 0
- When the load factor is 1 (\( T.\text{size} = T.\text{num} \)), \( \Phi(T) = T.\text{num} \), so the potential can pay for an expansion
- When the load factor is 1/4, \( T.\text{size} = 4 \times T.\text{num} \), which means \( \Phi(T) = T.\text{num} \), so the potential can pay for a contraction if an item is deleted

Analysis

- Let’s now roll up our sleeves and show that the amortized costs of insertions and deletions are small
- We’ll do this by case analysis
- Let \( \text{num}_i \) be the number of items in the table after the \( i \)-th operation, \( \text{size}_i \) be the size of the table after the \( i \)-th operation, and \( \alpha_i \) denote the load factor after the \( i \)-th operation

Table Insert

- If \( \alpha_{i-1} \geq 1/2 \), analysis is identical to the analysis done in the In-Class Exercise - amortized cost per operation is 3
- If \( \alpha_{i-1} < 1/2 \), the table will not expand as a result of the operation
- There are two subcases when \( \alpha_{i-1} < 1/2 \): 1) \( \alpha_i < 1/2 \) 2) \( \alpha_i \geq 1/2 \)
$\alpha_i < 1/2$

- In this case, we have
  \[ a_i = c_i + \Phi_i - \Phi_{i-1} \]
  \[ = 1 + (\text{size}_i/2 - \text{num}_i) - (\text{size}_{i-1}/2 - \text{num}_{i-1}) \]
  \[ = 1 + (\text{size}_i/2 - \text{num}_i) - (\text{size}_i/2 - (\text{num}_i - 1)) \]
  \[ = 0 \]

$\alpha_i \geq 1/2$

\[ a_i = c_i + \Phi_i - \Phi_{i-1} \]
\[ = 1 + (2 \times \text{num}_i - \text{size}_i) - (\text{size}_{i-1}/2 - \text{num}_{i-1}) \]
\[ = 1 + (2 \times (\text{num}_{i-1} + 1) - \text{size}_{i-1}) - (\text{size}_{i-1}/2 - \text{num}_{i-1}) \]
\[ = 3 \times \text{num}_{i-1} - \frac{3}{2} \times \text{size}_{i-1} + 3 \]
\[ < \frac{3}{2} \times \text{size}_{i-1} - \frac{3}{2} \times \text{size}_{i-1} + 3 \]
\[ = 3 \]

**Take Away**

- So we’ve just show that in all cases, the amortized cost of an insertion is 3
- We did this by case analysis
- What remains to be shown is that the amortized cost of deletion is small
- We’ll also do this by case analysis

**Deletions**

- For deletions, \( \text{num}_i = \text{num}_{i-1} - 1 \)
- We will look at two main cases: 1) \( \alpha_{i-1} < 1/2 \) and 2) \( \alpha_{i-1} \geq 1/2 \)
- For the case where \( \alpha_{i-1} < 1/2 \), there are two subcases: 1a) the \( i \)-th operation does not cause a contraction and 1b) the \( i \)-th operation does cause a contraction
Case 1a

- If $\alpha_{i-1} < 1/2$ and the $i$-th operation does not cause a contraction, we know $\text{size}_i = \text{size}_{i-1}$ and we have:

$$a_i = c_i + \Phi_i - \Phi_{i-1}$$

$$= 1 + (\text{size}_i/2 - \text{num}_i) - (\text{size}_{i-1}/2 - \text{num}_{i-1})$$

$$= 1 + (\text{size}_i/2 - \text{num}_i) - (\text{size}_i/2 - (\text{num}_i + 1))$$

$$= 2$$

Case 1b

- In this case, $\alpha_{i-1} < 1/2$ and the $i$-th operation causes a contraction.
- We know that: $c_i = \text{num}_i + 1$
- and $\text{size}_i/2 = \text{size}_{i-1}/4 = \text{num}_{i-1} = \text{num}_i + 1$. Thus:

$$a_i = c_i + \Phi_i - \Phi_{i-1}$$

$$= (\text{num}_i + 1) + (\text{size}_i/2 - \text{num}_i) - (\text{size}_{i-1}/2 - \text{num}_{i-1})$$

$$= (\text{num}_i + 1) + ((\text{num}_i + 1) - \text{num}_i) - ((2\text{num}_i + 2) - (\text{num}_i + 1))$$

$$= 1$$

Case 2

- In this case, $\alpha_{i-1} \geq 1/2$
- Proving that the amortized cost is constant for this case is left as an exercise to the diligent student
- Hint1: Q: In this case is it possible for the $i$-th operation to be a contraction? If so, when can this occur? Hint2: Try a case analysis on $\alpha_i$.

Take Away

- Since we've shown that the amortized cost of every operation is at most a constant, we've shown that any sequence of $n$ operations on a Dynamic table take $O(n)$ time
- Note that in our scheme, the load factor never drops below $1/4$
- This means that we also never have more than $3/4$ of the table that is just empty space
Disjoint Sets

- A disjoint set data structure maintains a collection \{S_1, S_2, \ldots, S_k\} of disjoint dynamic sets.
- Each set is identified by a representative which is a member of that set.
- Let’s call the members of the sets objects.

Operations

We want to support the following operations:

- **Make-Set(x)**: creates a new set whose only member (and representative) is \(x\).
- **Union(x, y)**: unites the sets that contain \(x\) and \(y\) (call them \(S_x\) and \(S_y\)) into a new set that is \(S_x \cup S_y\). The new set is added to the data structure while \(S_x\) and \(S_y\) are deleted. The representative of the new set is any member of the set.
- **Find-Set(x)**: Returns a pointer to the representative of the (unique) set containing \(x\).

Analysis

- We will analyze this data structure in terms of two parameters:
  1. \(n\), the number of Make-Set operations.
  2. \(m\), the total number of Make-Set, Union, and Find-Set operations.
- Since the sets are always disjoint, each Union operation reduces the number of sets by 1.
- So after \(n - 1\) Union operations, only one set remains.
- Thus the number of Union operations is at most \(n - 1\).

Analysis

- Note also that since the Make-Set operations are included in the total number of operations, we know that \(m \geq n\).
- We will in general assume that the Make-Set operations are the first \(n\) performed.
Application

- Consider a simplified version of Myspace
- Every person is an object and every set represents a social clique
- Whenever a person in the set $S_1$ forges a link to a person in the set $S_2$, then we want to create a new larger social clique $S_1 \cup S_2$ (and delete $S_1$ and $S_2$)
- We might also want to find a representative of each set, to make it easy to search through the set
- For obvious reasons, we want these operation of Union, Make-Set and Find-Set to be as fast as possible