Potential function of a (data structure) 
\[ \Phi : S \rightarrow \mathbb{R} \]
Initial state is \( S_0 \)
\[ \Phi(S_0) : \Phi_0 \]
\( S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots \)
\[ \Phi_0, \Phi_1, \Phi_2, \ldots \]
Let the work done be denoted by \( W_i \) when we move from \( S_i \) to \( S_{i+1} \), which is the actual work done.

Amortized work: \[ A_i = W_i + \Phi(S_{i+1}) - \Phi(S_i) \]
Change in potential

Total work done over a sequence of \( n \) updates:
\[ W = \sum_{i=0}^{n-1} W_i \]

Total Amortized work:
\[ = W + [\Phi - \Phi_0 + \Phi_2 - \Phi_1 + \cdots + \Phi_{n-1} - \Phi_n] \]
Total amortized work = Actual work + $\phi_f - \phi_i$

$\phi_f$ : final \hspace{0.5cm} $\phi_i$ : initial

If $\phi_f - \phi_i \geq 0$ \hspace{0.5cm} $\Rightarrow$ Total work \leq Amortized work

(In general Total work = Amortized work + $\phi_i - \phi_f$)

Example: Stack operations push, pop, empty stack

$\phi(\text{Stack})$ : +1 elements in stack

$\phi_i(\text{Empty Stack}) = 0$ \hspace{0.5cm} $\phi_f > \phi_i$

**Push**: Actual work : 1 \hspace{0.5cm} Amortized work : $1 + 1$ (change in potential)

\[ = 2 \]

**Pop**: Actual work : 1 \hspace{0.5cm} Amortized work : $1 - 1$

\[ = 0 \]

**Empty stack**: Actual work : #elements \hspace{0.5cm} Amortized work : #elements - #elements

\[ = 0 \]

Cont. g m Stack operations \leq amortized cost g m operations \leq 2 m
Example 2: Coin flip Problem
Potential function: # bits equal to 1

Amortized cost of movement:

(Worst case)

\[ \# \text{bit that flip is } 1 - \# \text{bit that flip is } 0 \]

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0 & 01 + 1 \\
0 & 1 & 0 & \cdot & \Delta \phi &= 0 \\
1 & 1 & + 1 & \Delta \phi &= 1 \\
1 & 0 & 0 & \cdot & \end{array}
\]

Amortized cost \( \leq 1 \)

Application to Binomial Heaps:
Suppose we keep inserting elements into a Binomial Heap: \( O(1) \) from analogy with the coin flip
Semi-dynamic dictionary problem

supports insertions and search
(Not deletion)
We will prefer using arrays

Goal: Support search and insertion in
\[ O(\log n) \] time \( \leq \)
\[ O(\log^2 n) \]

Idea: Suppose we store the elements in multiple sorted arrays

Cost of search: \[ \leq \log(n_i) \]

\( n_i \) is the size of array \( i \)
For example with \( \log n \) arrays, we can bound by \[ O(\log n \times \log n) = O(\log^2 n) \]

Insertion: If an existing array has extra space, insert it in the array or else create a new array of twice the size
Consider arrays that size $2^i$ $i \geq 0$
like in Binomial heaps, we can represent any number in using at most $\log n$ such
arrays $\Rightarrow$ search-time $O(\log^2 n)$
(At most one array $\Rightarrow$ $\log n$ binary search)

**Insertion**: Create an array of size 1
and "merge" with the existing arrays

[Until there are no array of size $2^i$
starting with $i = 0$]
we merge two arrays of sizes $2^i$

11 elements

11

\[
\begin{array}{cccccc}
1 & 1 & 1 & 1 & 1 & 1 \\
\hline
8
\end{array}
\]
Worst case cost: \[ \leq O(2^i) \]
\[ \forall i \leq k \]
\[ \leq O(2^{k+1}) = O(2^k) \]

Amortized cost?

Given \( O(\log n) \) amortized cost

When we merge two arrays, a size \( m \)

\[ \begin{array}{c}
\text{2m} \\
\downarrow \\
\text{m} \\
\downarrow \\
\text{m}
\end{array} \]

Cost is \( O(2m) \)

\( O(\# \text{ elements involved}) \)

Let us pretend that we have a counter with each element that stores the cost of merging

\[ x_1, x_2, \ldots, x_n \]

Define \( C_i \) as the counter.

\[ \leq C_i : \text{ cost of operation } \]

Define amortized cost in the charging

and analyze the algorithm

Define potential function for each element