1 Amortized

To rigorously demonstrate amortized analysis, **potential function** will be important.

Definition

A potential function is a function $\Phi(t)$ that depends on the *state* of data structure at time t.

As long as a potential function is well chosen, arguing amortized runtime is quite straightforward. However, coming up with a proper potential function is a tricky business. How to choose a potential function is out of the scope of this course.

Proposition

A potential function $\Phi(t)$ needs to satisfy the following

- $\Phi(t) \geq 0$ for all t
- $\Phi(t) = 0$ for $t = 0$

These are the only restrictions on potential functions. Any functions satisfying these two properties may be a candidate for amortized analysis. Once we have a potential function, amortized runtime from t to $t + 1$ can be calculated by the following formula.

Definition

Amortized runtime of step $t \to t + 1$ is defined to be

$$\text{actual runtime} + [\Phi(t + 1) - \Phi(t)]$$

We call the difference $\Phi(t + 1) - \Phi(t)$ the potential difference.

1.1 Dynamic Array Insertion

We used to know that dynamic array insertion takes amortized runtime $O(1)$, but we never had a rigorous analysis via potential functions. We will rigorously analyze the amortized runtime for dynamic array insertion using a well-chosen potential function. To start off, we provide a nice potential function for dynamic arrays.

Definition

- Let c be a constant such that copying n items takes at most $c \cdot n$.
- Let n denote the number of items in the array.
- Let M denote the capacity of the array.

A potential function for dynamic arrays is chosen to be

$$\Phi(t) = c \cdot (2n - M + 1) + c$$

When an item is being inserted in a dynamic array, there are usually two possible stages at time $t + 1$ of that array. It is either full or not. We will be using some notations in analysis of both cases. Let M' be the capacity at time $t + 1$, n' be the number of items in the array at time $t + 1$ and a constant ϵ be the time of inserting one item in the array that has space.

1.1.1 Array Having Space

Let's consider the case where the array has space at $t + 1$. We have nothing to do with capacity, thus $M' = M$. Adding an item increases the number of items in the array by 1, that is, $n' = n + 1$. Now, we can calculate the amortized runtime in this case. The potential difference will be

$$\begin{aligned}\Phi(t+1) - \Phi(t) &= c \cdot (2n' - M' + 1) - c \cdot (2n - M + 1) + c \\ &= 2c\end{aligned}$$

Since the actual runtime and the potential difference are both order 1, the amortized runtime with the potential difference is order 1 as well. Therefore, in this case, dynamic array insertion takes amortized $O(1)$.

1.1.2 Full Array

The remaining case is where the array is full at time $t + 1$. We double the capacity, that is, $M' = 2M$. Since it is full, the number of items is exactly the capacity, that is, $M = n$. The capacity at time $t + 1$ is $2n$. Then, Adding one item increases n by one, that is, $n' = n + 1$. The potential difference will be

$$\begin{aligned}\Phi(t+1) - \Phi(t) &= c \cdot (2n' - M' + 1) - c \cdot (2n - M + 1) + c \\ &= 2c - c \cdot n\end{aligned}$$

Note that the actual runtime has an extra $c \cdot n$ compared to the first case, since we need to copy over the array to the newly allocated space. Therefore, the total amortized runtime is

$$\epsilon + c \cdot n + 2 \cdot c - c \cdot n = \epsilon + 2 \cdot c \in O(1)$$

2 Randomized Analysis

Most programming languages have a built-in pseudo-random number generator.

Example

`random(n)` returns a pseudo-random integer between 0 and $n-1$.

Randomized analysis usually provides an expected runtime. Let R be the sample space of random numbers. The expected runtime of instance $\mathcal{I}$ based on an element in R will be

$$\begin{aligned} T^{(\text{exp})}(\mathcal{I}) &= \text{expected runtime of instance } \mathcal{I} \\ &= E[\text{runtime of instance } \mathcal{I}] \\ &= \sum_{r \in R} P(r \text{ is chosen}) \cdot \text{runtime of } \mathcal{I} \text{ based on } r \end{aligned}$$

Note that $T^{(\text{exp})}(\mathcal{I})$ is the runtime of $\mathcal{I}$, not the algorithm. We define the expected runtime to be the expected runtime of the worst instance $\mathcal{I}$.

Definition

The expected runtime of an algorithm of instance size n is

$$T^{(\text{exp})}(n) = \max_{\text{size}(\mathcal{I})=n} T^{(\text{exp})}(\mathcal{I})$$

2.1 Downward Random Walk on Tree

Given a binary tree, do a random walk downward until you reach a `NIL`. Note that the worst instance of size n is a path. The worst possible runtime is $O(n)$ when the first `NIL` you visited is the end of the tree.

Theorem

The expected runtime $T(n)$ is $O(\log(n))$.

Proof. Let c be a constant such that going from a node to its randomly chosen child takes at most c. Show that $T(n) \leq c \cdot \log(n+1)$ by induction. When $n = 1$, you reach a `NIL` for the next step regardless of the random number generated. Hence, it takes at most c to finish the random walk. The RHS is $c \cdot \log(2) = 2$. Thus, base case holds. We may assume that $n > 1$. Let n_L and n_R denote the size of sub-trees. Note that $n_L + n_R = n - 1$.

$$
\begin{aligned}
T(n) &= c + P(\text{chose left}) \cdot T(n_L) + P(\text{chose right}) \cdot T(n_R) \\
&\leq c + \frac{c}{2} \cdot \log(n_L + 1) + \frac{c}{2} \cdot \log(n_R + 1) \qquad \text{by IH}
\end{aligned}
$$

Proposition

If f is a concave function, then

$$
\frac{f(x) + f(y)}{2} \leq f\left(\frac{x + y}{2}\right)
$$

Since log is a concave function, by applying this property, we get

$$
\begin{aligned}
T(n) &\leq c + c \cdot \frac{\log(n_L + 1) + \log(n_R + 1)}{2} \\
&\leq c + c \cdot \log\left[\frac{n_L + 1 + n_R + 1}{2}\right] \\
&\leq c + c(\log(n + 1) - 1) \qquad \text{since } n_L + n_R + 2 = n + 1 \\
&= c \cdot \log(n + 1) \\
&\in O(\log(n))
\end{aligned}
$$

□