Data Structures - Week 2

Thursday, March 23, 2023 8:14 AM

Dynamic Arrays

Problem: static arrays are static

Semi-solution: dynamically-allocated array

Problem: might not know max size when allocating an array

All problems in computer science can be solved by another level of indirection

Solution: dynamic arrays (also know as resizable arrays)

Idea: store a pointer to a dynamically allocated array, and replace it with a newly-allocated array as needed

Dynamic Arrays:

Abstract data type with the following operations (at a minimum)

- Get(i): returns element at location i*
- Set(I, val): sets element I to val*
- PushBack(val): Adds val to the end
- Remove(i): removes element at location I
- Size(): the number of elements

Store:

- Arr: dynamically-allocated array
- Capacity: size of the dynamically-allocated array
- Size: number of elements currently in the array

Runtimes

- Get(i): O(1)

- Set(I, val): O(1)
- PushBack(val): O(n)
- Remove(i): O(n)

Dynamic Array

We only resize every so often. Many O(1) operations are followed by an O(n) operation.

What is the total cost of inserting many elements

Amortized cost: Given a sequence of n operations, the amortized cost is

Cost (n operations) / n

Aggregate Method

Dynamic array: n calls to PushBack

Let Ci = cost of I'th insertion

Ci = 1 + { i - 1 if I -1 is power of 2
{ 0

$$\frac{\sum_{i=1}^{n} c_i}{n} = \frac{n + \sum_{j=1}^{\lfloor \log_2(n-1) \rfloor} 2^j}{n} = \frac{O(n)}{n} = O(1)$$

Banker's method

- Charge extra for each cheap operation
- Save the extra charge as tokens in your data structure (conceptually)
- Use the tokens to pay for expensive operations

Like an amortizing loan

Dynamic array: n calls to PushBack

Charge 3 for each insertion: 1 token is the raw cost for insertion

- Resize needed: to pay for moving the elements, use the token that's present on each element that needs to move

- Place one token on the newly inserted element, and one token capacity / 2 elements prior

Physicist Method

- Define the potential function, Φ which maps states of the data structure to integers:
  - Φ(h0) = 0
  - Φ(ht) >= 0
- Amortized cost for operation t:
  - Ct + Φ(ht) - Φ(ht-1)
- Choose Φ so that:
  - If ct is small the potential increases
  - If ct is large, the potential decreases by the same scale

  - The cost of n operations is: $\sum_{i=1}^{n} c_i$
  - The sum of the amortized costs is:

$$\sum_{i=1}^{n}(c_i + \Phi(h_i) - \Phi(h_{i-1}))$$

$$\begin{aligned}
=& c_1 + \Phi(h_1) - \Phi(h_0) + \\
& c_2 + \Phi(h_2) - \Phi(h_1) \cdots + \\
& c_n + \Phi(h_n) - \Phi(h_{n-1})
\end{aligned}$$

$$= \Phi(h_n) - \Phi(h_0) + \sum_{i=1}^{n} c_i \geq \sum_{i=1}^{n} c_i$$

Dynamic Array: n calls PushBack

- Let Φ(h) = 2 x size - capacity
  - Φ(h0) = 2 x 0 - 0
  - Φ(hi) = 2 x size - capacity > 0
    - Since (size > capacity / 2)

Without resize when adding element I
Amortized cost of adding element i:

$C_i + \Phi(h_i) - \Phi(h_{i-1})$

$= 1 + 2 \times size_i - cap_i - (2 \times size_{i-1} - cap_{i-1})$

$= 1 + 2 \times (size_i - size_{i-1})$

$= 3$

With resize when adding element I

Let $k = size_{i-1} = cap_{i-1}$

Then

$\Phi(h_{i-1}) = 2size_{i-1} - cap_{i-1} = 2k - k = k$

$\Phi(h_i) = 2\ size_i - cap_i = 2(k+1) - 2k = 2$

Amortized cost of adding element i:

$C_i + \Phi(h_i) - \Phi(h_{i-1})$

$=(size_i) + 2 - k$