DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 2 | Part 1

More about Memory

Is appending an array really so slow?

▶ `malloc()` vs. `realloc()`

Is appending an array really so slow?

- ► If `realloc` doesn't copy: $\Theta(1)$.

- ► If `realloc` copies: $\Theta(n)$.

- ► Assume p is probability that `realloc` copies.

- ► **Expected time** is still[1] $\Theta(n)$.

---

[1] If p doesn't depend on n.

How is empty memory found?

► Basically: a linked list.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 2 | Part 2

Dynamic Arrays

Motivation

▶ Can we have the best of both worlds?

▶ Θ(1) time access like an array.

▶ Θ(1) time append like a linked list.

▶ **Yes!** (sort of)

The Idea

- Allocate memory for an **underlying array**.
  - say, 512 elements
  - This is the **physical size**.

- To append element, insert into first unused slot.
  - Number of elements used is the **logical size**.
  - Θ(1) time.

The Idea

▶ We'll eventually run out of unused slots.

▶ Fix: allocate a new underlying array whose physical size is γ times as large.
  - ▶ γ is the **growth factor**.
  - ▶ Commonly, γ = 2; i.e., double its size.
  - ▶ Takes $\Theta(k)$ time, where k is current size.

Example

```
» > arr = DynamicArray(initial_physical_size=4)
» > arr.append(1)
» > arr.append(2)
» > arr.append(3)
» > arr.append(4)
» > arr.append(5)
```

(notebook)

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 2 | Part 3

Amortized Analysis

Analysis

► Appending takes Θ(1) time *usually...*

► ...but takes Θ(k) time when we run out of slots.
  ► Where k is current size of sequence.

The Key

- Resizing is expensive, but rare.
  - If $\gamma = 2$, each new resize is twice as expensive, but happens half as often.

- Thus, the **cost per append** is small.

- **Amortize** the cost over all previous appends.

Amortized Time Complexity

▶ The **amortized** time for an append is:

$$T_{\mathrm{amort}}(n) = \frac{\text{total time for } n \text{ appends}}{n}$$

▶ We'll see that $T_{\mathrm{amort}}(n) = \Theta(1)$.

Amortized Analysis

total time for n appends
 =
total time for **non-growing** appends
+
total time for **growing** appends

Counting Growing Appends

- ▶ Want to calculate time taken by growing appends.

- ▶ First: how many appends caused a resize?
  - ▶ β: initial physical size
  - ▶ γ: growth factor

Counting Growing Appends

▶ Suppose initial physical size is β = 512, and γ = 2

▶ Resizes occur on append #:

$$512, 1024, 2048, 4096, \ldots$$

▶ In general, resizes occur on append #:

$$\beta\gamma^0, \beta\gamma^1, \beta\gamma^2, \beta\gamma^3, \ldots$$

Counting Growing Appends

▶ In a sequence of *n* appends, how many caused the physical size to grow?

▶ Simplification: Assume *n* is such that *n*th append caused a resize. Then, for some $x \in \{0, 1, 2, \ldots\}$:

$$n = \beta \gamma^x$$

▶ If $x = 0$ there was 1 resize; if $x = 1$ there were 2; etc.

Counting Growing Appends

▶ Solving for x:

$$x = \log_\gamma \frac{n}{\beta}$$

▶ Check: without assumption, $x = \lfloor \log_\gamma \frac{n}{\beta} \rfloor$

▶ Number of resizes is $\lfloor \log_\gamma \frac{n}{\beta} \rfloor + 1$

Counting Growing Appends

▶ Number of resizes is $\lfloor \log_\gamma \frac{n}{\beta} \rfloor + 1$

▶ Check with γ = 2, β = 512, n = 400
  ▶ Correct # of resizes: 0

▶ Check with γ = 2, β = 512, n = 1100
  ▶ Correct # of resizes: 2

Time of Growing Appends

▶ How much time was taken across all appends that caused resizes?

▶ Assumption: resizing an array with physical size k takes time $ck = \Theta(k)$.
  ▶ c is a constant that depends on γ.

Time of Growing Appends

- ▶ Time for first resize: $c\beta$.

- ▶ Time for second resize: $c\gamma\beta$.

- ▶ Time for third resize: $c\gamma^2\beta$.

- ▶ Time for jth resize: $c\gamma^{j-1}\beta$.

- ▶ This is a **geometric progression**.

Time of Growing Appends

► Time for jth resize: $c\gamma^{j-1}\beta$.

► Suppose there are r resizes.

► Total time:

$$c\beta \sum_{j=1}^{r} \gamma^{j-1} = c\beta \sum_{j=0}^{r} \gamma^{j}$$

Recall: Geometric Sum

▶ From some class you've taken:

$$\sum_{p=0}^{N} x^p = \frac{1 - x^{N+1}}{1 - x}$$

▶ Example:

$$1 + 2 + 4 + 8 + 16 = \sum_{p=0}^{4} 2^p = \frac{1 - 2^5}{1 - 2} = 31$$

Time of Growing Appends

▶ Total time:

$$c\beta \sum_{j=0}^{r} \gamma^j = c\beta \frac{1 - \gamma^{r+1}}{1 - \gamma}$$

Time of Growing Appends

▶ Remember: in n appends there are $r = \lfloor \log_\gamma \frac{n}{\beta} \rfloor + 1$ resizes.

▶ Total time:

$$c\beta \frac{1 - \gamma^{r+1}}{1 - \gamma} = c\beta \frac{1 - \gamma^{\lfloor \log_\gamma \frac{n}{\beta} \rfloor + 2}}{1 - \gamma}$$

$$= \Theta(n)$$

Amortized Analysis

total time for n appends
 =
total time for **non-growing** appends
+
$\Theta(n)$ ← total time for **growing** appends

Time of Non-Growing Appends

► In a sequence of *n* appends, how many are **non-growing**?

$$n - \left(\lfloor \log_\gamma \frac{n}{\beta} \rfloor + 1 \right) = \Theta(n)$$

► Time for one such append: $\Theta(1)$.

► Total time: $\Theta(n) \times \Theta(1) = \Theta(n)$.

Amortized Analysis

total time for *n* appends
 =
$\Theta(n)$ ← total time for **non-growing** appends
+
$\Theta(n)$ ← total time for **growing** appends

Amortized Time Complexity

▶ The **amortized** time for an append is:

$$T_{\text{amort}}(n) = \frac{\text{total time for } n \text{ appends}}{n}$$

$$= \frac{\Theta(n)}{n}$$

$$= \Theta(1)$$

Dynamic Array Time Complexities

▶ Retrieve kth element: $\Theta(1)$

▶ Append/pop element at end:
  ▶ $\Theta(1)$ best case
  ▶ $\Theta(n)$ worst case (where n = current size)
  ▶ $\Theta(1)$ amortized

▶ Insert/remove in middle: $O(n)$
  ▶ May or may not need resize, still $O(n)$!

DSC 190
DATA STRUCTURES & ALGORITHMS

Lecture 2 | Part 4

Practicalities

Advantages

▶ Great cache performance (it's an array).

▶ Fast access.

▶ Don't need to know size in advance of allocation.

Downsides

- ► Wasted memory.

- ► Expensive deletion in middle.

Implementations

- ▶ Python: `list`

- ▶ C++: `std::vector`

- ▶ Java: `ArrayList`

Exercise

Why do we need `np.array`? Python's `list` is a dynamic array, isn't that better?

In defense of `np.array`

▶ Memory savings are one reason.

▶ Bigger reason: using Python's `list` to store numbers does not have good **cache** performance.