CS 310 Lecture

Connor Baker

January 29, 2019

Worst, Average, or Best Case

- Plan for the worst, hope for the best
- Best case isn't usually helpful (best case is almost always $O(1)$)
- Average case can be helpful but it is typically very hard to prove, or can only be shown probabilistic-ally
- Worst case is the most important (usually)

String Concatenation

- Efficiency analysis needs to consider everything that the computer does
  - Even the stuff that's not obvious!

```java
void method8(String[] arr) {
    String result = "[";
    for (String s : arr) {
        result += s + " ";
    }
    result += "]";
    System.out.println(result);
}
```

In the code above, we might think that `result += s + " ";` is constant, but it is in fact linear. Put this inside of a `for` loop and we've got $O(n^2)$ already.

New Topic: List

- Wikipedia: a list or sequence is an abstract data type that represents a countable number of ordered values, where teh same value may occur more than once
  - Dynamic array: like `ArrayList` from Java's `Collections` framework
  - Linked lists (either singly or doubly linked)

Outline Today

- Goals today: check how to implement an expandable array similar to `ArrayList`
  - Generic type with type parameters
  - Implementation highlights
    * Size can grow if needed
    * Lacking nice `[]` syntax: use `get()` and `set()`
    * Other convenient operations supported
  - Analyze the complexity

Our Expandable Array

- Using an underlying array to keep data

  ```
  public class MyArrayList<T> {
      T[] data;
      int size;
      ...
  }
  ```

- Generic class (an array that can hold any type `T`)
- `data` is a standard fixed sized array
  - Use consecutive locations, no "holes" allowed
- If/when `data` runs out of space, expand it: when? how?
- What is the use of size?

Create MyArrayList

```
public class MyArrayList<T> {
    T[] data; // Holds elements
    int size; // Virtual size
    public MyArrayList(); // Initialize fields
    public int size(); // Virtual size of ArrayList
    public void add(T x); // Add an element to the end
    public T get(int i); // Accessing an element
    public void set(int i, T x);

    // Only replaces an existing element
    public void insert(int i, T x);

    /// Insert x at position i, shift elements if necessary
    public T remove(int i);

    // Remove element at position i, shift elements to remove the gap
    public int indexOf(T x);
}
```

Expanding Array

- Which methods need to expand?
  - cats
- When to expand?
  - If/when data runs out of space
- How to expand array `data`?
  1. Allocate a new larger array `data2`
  2. Copy from `data` to `data2`
  3. Add new elements(s) to `data2`
  4. Update reference: set `data` to `data2`
  5. GC gets the old array

Implementation

- Demo in Java

Key Observations

- Can't do `new T[10]`, instead use `Object[] + Casting`
  - Recall that arrays are covariant and generics are invariant
  - Considered as unsafe operators
    - We can use `@SuppressWarnings("unchecked")` but only when we absolutely must
- Magic numbers: standard Java `ArrayList` increases the new size to `3/2*oldSize+1`
  - Chose based on engineering experience rather than theory, can use bit shifts to compute the size quickly
  - Similarly, default size = 10

Complexity

- `ArrayList` of size N

Method	Big-O
`.size()`	$O(1)$
`.get(i)`	$O(1)$
`.set(i, x)`	$O(1)$
`.add(x)`	$O(n)$
`.insert(i, x)`	$O(n)$
`.remove(i)`	$O(n)$
`.indexOf(x)`	$O(n)$

Compare with a Static Array

- Array of size N
- Worst case

Implementation	get/set	add/del at end	add/del at start	add/del in mid	search	can grow?
Static Array	1	1	N	N	N	no
Dynamic Array	1	**N**	N	N	N	yes

- Wait... we are only occasionally expanding the array, so do we care about all these other things?

Amortized Analysis for `add(x)`

i-th call	1	2	3	4	5	6	7	8	9	10	11	12
size	1	2	4	4	8	8	8	8	16	16	16	16
Cost of doubling/copying		1	2		4				8			
Cost of putting `x`	1	1	1	1	1	1	1	1	1	1	1	1
Total cost	1	2	3	1	5	1	1	1	9	1	1	1

- Assume that we start with `capacity = 1`
- Assume that we always double the capacity
- Worst case: keep adding, no removing

Dynamic Array Add: Algebraic Approach

- If we always double the array...
- c_i is the cost of the i-th call
  - If $i-1$ is an exact power of 2, we need to expand and $c_i = 1$
- Total time for N operations is $O(n)$ as shown below

$$\begin{aligned}\sum_{i=1}^{n} &\leq n + \sum_{j=0}^{\lfloor \log(n) \rfloor} 2^j \\ &< n + 2^{\lfloor \log(n) \rfloor + 1} \\ &= n + 2 \cdot 2^{\lfloor \log(n) \rfloor} \\ &\leq n + 2n \\ &= 3n\end{aligned}$$

  - Amortized analysis shows that `add(x)` is $O(1)$

Amortized Analysis

- Consider a sequence of M operations

$$\text{amortized efficiency} = \frac{\text{worst-case sequence efficiency}}{M}$$

- Looks at the time performance a sequence of operations averaged over the number of operations: $T(n)/n$
- shows that the average cost over time isn't as bad as the worst case for a single operation
- This is **NOT** the same as average case analysis
  - **Average case:** the expected cost of each operation (innately probabilistic)
  - **Amortized:** the average cost of each operation is the worst case

Complexity

- `add()` and `remove()` is amortized as a constant
- Now competitive with a static array for linear operations

Implementation	get/set	add/del at end	add/del start	add/del mid	search	can grow?
Static Array	1	1	N	N	N	no
Dynamic Array	1	**1***	N	N	N	no

*Amortized analysis

Take-Home

- Today: expandable array
  - Practice by completing the code
  - Time/space best/worst case analysis
- Next time: linked lists!
  - Reading: Chapter 17 of Weiss