Advanced Data Structures

Lecture by Dr. Shay Mozes
Typeset by Steven Karas

2016-11-10

1 Administration

Office hours: Thursday 1730 - 1830 or by appointment Grade: 70% exam, 20% homework, 10% scribe notes

5 or less theoretical homeworks. No programming exercises, yet it can sometimes be helpful to implement to help understand. Less than 20 hours per homework. More or less around the same level of effort as Advanced Algorithms. 5 bonus points for submitting homework typeset in LaTeX.

Collaboration policy: OK to study and brainstorm together. Do not take written notes, and wait a while before writing your solution. Disclose who you worked with, cite sources used, etc.

Scribe notes - 1 lecture summarized in Hebrew LaTeX by students. Can be done in pairs. This is an experiment, and if it fails, then the grading changes to 80% exam, 20% homework.

Somewhat based on Advanced Data Structures MIT and from University of Haifa taught by Dr. Oren Weimann (use password 111111111).

2 Syllabus

Study various advanced Some simple evaluation models (WORD RAM model, Cache oblivious, Integers)

3 Introduction

Data structures store and organize data for querying and updating. We evaluate data structures based on time and space complexity.

3.1 Classical data structures

1. Arrays

2. Stack, Queue, Heap
   Note: Dijkstra does extract-min n times, and decrease-key m times.
3. Hash tables
4. Linked list
5. Binary trees:
   AVL Trees
   Red-Black Trees
   2-3 Trees
6. Graphs
7. Strings

3.2 Advanced data structures

1. Binomial heaps
   Dijkstra in $O((m+n)logn)$
2. Fibonacci heaps
   Dijkstra in $O(m+nlogn)$
3. Perfect & Universal hashing
4. Self-balancing lists
   Move frequently accessed items earlier in the list
   Can be precomputed
5. Splay trees
   Amortized $O(mlogn)$ time
6. Link-cut trees
   Support addition, deletion of subtrees
7. Dynamic graphs
   Support adding/dropping edges
8. Suffix trees
   Traditionally used for text search

4 Amortized Analysis

When actions are batched together, we can trade off worst case performance for aggregate throughput.

For example, take a tree implementation where each operation takes $O(\log n)$, but one operation takes $O(n^2)$. We want to see the complexity of $m = n^3$ operations: Classical worst case is $O(m \cdot n^2) = O(n^5)$. When we amortize that expensive operation, we can tighten this up to: $O((m-1)\log n + n^2) = O(n^3 \log n)$

Colloquially, amortized analysis is taking the aggregate worst case of all possible sequences of operations.

Growing array example We'll be using this for the rest of this lesson. A linked list that supports one operation: Insert an element (at the first empty spot in the underlying array). We start with an empty (size=1) array. When we need to insert a new element, we reallocate the array, and copy over all the existing elements. If we increase the size of the array by 1 each time, this gives us a cost of $O(n)$ to insert a new element, which gives an amortized complexity of $O(n^2)$.

Fixing the growth factor If we reallocate the array to double in size each time: Assume that $n = 2^k$ for convenience. So $n + \sum_{i=0}^{k-1} 2^i = 2m - 1$[1]. If we take the worst possible case of $n = 2^k + 1$, then we get that the overall time is $3m - 1$. Note that the worst case for an individual operation is $O(m)$, and yet the worst case for m operations is also $O(m)$!

4.1 Formal definition

Given that $worst(op)$ = the max time of all possible cases of an operation op. We say that the amortized time complexity is $amort(op)$ if for some m, every sequence of m operations takes at most $m \cdot amort(op)$. In other words: $amort(op) = \max_{\text{all possible sequences}} \frac{\text{time for m ops}}{m}$

In other words, amortized bounds is a bound on the amortized time for each operation.

4.2 Aggregate method

Direct computation of the amortized complexity using the above definitions.

4.3 Accounting method

For each operation, we are given a budget that for any given operation x, we need not utilize the entire budget.

Growing array example Every operation gives us a budget of 2 shekels (amortized cost). Every primitive action costs us 1 shekel. We need to show that at any given moment, we have at least 0 shekels. If we can prove this, we are proving that a sequence of m operations takes at most $m \cdot amort(op)$. However, this is insufficient, because we run out of money after the first reallocation. However, if we increase the per-operation budget to 3 shekels, we never run out of money[2]. As such, we've proven that the amortized cost of m bounds is $3m$[3].

[1] Note that $\sum_{i=0}^{k-1} 2^i = 2^k - 1$. We will use this equivalence a lot in this course

[2] Visual proof done on whiteboard

[3] For a formal proof, you would want to prove by induction for the cost of groups of 2^n operations

Binary counter example A counter that counts in binary. For example: 0, 1, 10, 11, 100, 101, etc. Rather than assuming that all operations are equal, we grant a budget of 2 shekels to flipping a bit from 0 to 1, and a budget of -1 shekels to flipping a 1 to 0. We can see that at most for any given operation we flip exactly one bit from 0 to 1[4].

4.4 Potential method

Rather than associating a cost to each operation, we assign a budget/cost to each state of the data structure. For every state D_i of the data structure after the ith operation, there is a potential $\phi(D_i)$. The amortized cost of the ith operation is $amort(op_i) = time(op_i) + \phi(D_i) - \phi(D_{i-1})$.

$$\sum_{i=1}^{m} amort(op_i) = \sum_{i=1}^{m} time(op_i) + \sum_{i=1}^{m} \phi(D_i) - \sum_{i=1}^{m} \phi(D_{i-1})$$

$$= \sum_{i=1}^{m} time(op_i) + \phi(D_m) - \phi(D_0)$$

$$\sum_{i=1}^{m} amort(op_i) \geq \sum_{i=1}^{m} time(op_i)$$

$$\phi(D_i) \geq 0$$

$$\phi(D_0) = 0$$

Binary counter example Using a binary counter, we define $\phi(D)$ = the number of 1s in the counter[5]. For some sequence t_i of subsequent 1s at the beginning of the counter, then $amort(op_i) = t_i + 1 + 1 - t_i = 2$ where $time(op_i) = t_i + 1$ and $\Delta\phi = 1 - t_i$.

Growing array example Define the potential function $\phi(D_i) = full(D_i) - empty(D_i) = 2 \cdot full(D_i) - size(D_i)$. For example, an array of size 4 that is completely full has $\phi(D_i) = 4$. After we add one more element, the array is now has 8 cells, of which 5 are full, giving us a potential of $\phi(D_i) = 2$.

$$amort(op_i) = \begin{cases} \text{regular operation} & 1 + 2 = 3 \\ \text{growing the array} & 2^k + 1 + 2^k + 1 - (2^k - 1) - 2^k = 3 \end{cases}$$

Supporting deletion Trivially, we can show that deletion takes $O(1)$ time. However, we need to show that this doesn't interfere with the original proof. We can do this by binding $\phi(D_i)$ to 0 in cases where the array is more than half empty. This changes the amortized cost of a regular addition to $\phi(D_i) \leq 3$.

$$amort(delete_i) = 1$$

[4]Formal proof elided for brevity

[5]It's important to define the potential function carefully for proofs using this method

5 Next week

Next week will likely be either heaps or hashing.