COMPSCI 330: Design and Analysis of Algorithms Spring 2019

Lecture 21: Amortized Analysis

Lecturer: Rong Ge *Scribe: Shweta Patwa*

1 Dynamic Array Problem

Goal: Design a data structure such that adding a number takes $O(1)$ amortized running time.

What is special is that the size of the array can change dynamically during running time.
Need to reallocate new memory and more the numbers from the old array to the new array, which can hold more numbers. It is not useful to allocate too much memory if we will not need it.

Increasing the size by one every time will take $O(n^2)$ time for n operations, and the amortized running time will be much slower than we had expected.
How much larger do we make the new array?

We double the capacity when we run out of space in the existing list.

Init():

capacity = 1
length = 0
allocate an array $a[]$ of 1 element

Add(x):

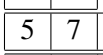
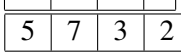
If length<capacity:

a[length] = x
length += 1

Else:

capacity = capacity $*2$
allocate a new array $a[]$ of capacity elements
copy the current array to the new array
a[length] = x
length += 1

Here is an example:

Init(): capacity=1, length= 0 
Add(5): capacity=1, length= 1 
Add(7): capacity=2, length= 2 
Add(3): capacity=4, length= 3 
Add(2), Add(4): capacity=8, length= 5

1.1 Aggregate Method

Idea: Compute the total cost of n operations, divide the total cost by n.

This is used in analyzing MergeSort and DFS.

Aggregate method for dynamic array: how many operations are needed for adding n numbers?

What is the cost of the i^{th} add operation?

Case 1: If we do not need to allocate new memory.
Algorithm takes $O(1)$ unit of time.

Case 2: If we need to allocate new memory.
For which values of i do we need to allocate new memory?
When $i = 2^k + 1$, where $k = 1, 2, ...$, we exceed the capacity and need to double the size of the array.
Thus, at this step, we need to

- allocate new memory,
- copy over the 2^k numbers from the old array to the new array, and
- and insert the new number

This algorithm takes $2^k + 1$ running time.

$t(i)$: amount of time

$$t(i) = \begin{cases} 2^k + 1 & \text{,if } i = 2^k + 1 \text{ (case 1)} \\ 1 & \text{,otherwise (case 2)} \end{cases}$$

Amortized cost = $\frac{\sum_{i=1}^{n} t_i}{n}$

$\sum_{i=1}^{n} t_i = n + \sum_{k:2^k+1 \leq n} 2^k$
In either case, we have a 1 showing up and so summation over $i = 1$ to n results in n. The remainder comes from the case 2 operations. These operations cannot happen too many times. Thus,

$$\begin{aligned} \sum_{i=1}^{n} t_i &= n + \sum_{k=0}^{\lfloor \log_2 (n-1) \rfloor} 2^k \\ &= n + 2^{\lfloor \log_2 (n-1) \rfloor + 1} - 1 \\ &\leq n + 2n \qquad (2^{\lfloor \log_2 (n-1) \rfloor} \leq n - 1 < n) \\ &= 3n \end{aligned}$$

Thus, total cost of n operations is always at most $3n$. Hence, the amortized cost is at most 3, i.e., the amortized cost for add operation is $O(1)$.

We can run into issues when the summation is more complicated than in the examples we have seen so far.

1.2 Accounting (charging) Method

Idea: small number of expensive operations, whereas most of the operations are extremely fast.
The expensive operations are the ones that cause the capacity of the array to get doubled.
If every normal operation pays a little bit extra, that can be enough to pay for the more expensive operations.
Another way to think about this is every time you perform a cheap operation, you save some amount of money or time in a bank, and when it comes time to pay for the expensive operations, you reach into the bank and use what you need.

i	1	2	3	4	5	6	7	8	9
$t(i)$	1	2	3	1	5	1	1	1	9

Recall that an expensive operation causes the capacity to get doubled. This can be see in the $t(i)$ values. For example, when the capacity goes from 4 to 5, it costs 5 units because we need to double the size of the array and copy over numbers from the old copy to this newly created array.

How many cheap operations are there between two expensive operations ($2^{k-1}+1$ to 2^k+1)?
The number of consecutive ones in $t(i)$ is $2^k+1-(2^{k-1}+1)-1$ or $2^{k-1}-1$.

Question: How much time does each operation need to pay?
We want this to roughly equal $\frac{2^k+1}{2^{k-1}+1} \approx 2$ as k becomes large.

Plan: Each cheap operation will pay one for itself, and 2 to the bank. The money in the bank is used to pay for the more expensive operations. When $i = 2^k+1$, use $2(2^{k-1}-1)$ from the bank, and pay 3 for itself. Set $f(i) = 2^k+1$. Hence, the amortized cost is at most 3 because we are paying at most 3 per operation.

1.3 Potential Method

Recall: Law of physics; swinging pendulum. At the top, the ball has zero velocity and all the energy is in the form of potential energy, whereas in the middle where the velocity is the highest, all the energy is in the form of kinetic energy. Energy can come from the change in potential.

Define a potential function $\Phi, \Phi > 0$. The value is only going to depend on the current state of the data structure.

When executing an expensive operation, the potential function should drop and this drop should pay for the expensive operation. Is the idea of the potential function similar to the bank in the case of the charging method? (Hint: There are some slight differences that we will see later.)

$$A_i = T_i - \Phi(x_i) + \Phi(x_{i+1})$$

i.e., amortized cost equals the real cost minus the potential before plus the potential after the i^{th} operation. If the i^{th} operation is costly, we expect the potential before to be a lot higher than the potential after executing the i^{th} operation. If the real cost is very small, then we can allow the potential before to be smaller than the potential after. Thus, we have saved some energy in the potential.

Claim. $\sum_{i=1}^{n} T_i = \sum_{i=1}^{n} A_i + \Phi(x_1) - \Phi(x_{n+1})$.

Let's apply the potential argument to analyze the dynamic array.
Φ(dynamic array) = $2*\text{length} - \text{capacity} + 1$

For a normal operation i, say it changes the array from x_i to x_{i+1}; think of x_i as encoding the state of your data structure. Here, x has two elements, x.length and x.capacity. Thus,

x_i	x_{i+1}
length	length+1
capacity	capacity

As a result, $\Phi(x_i) + 2 = \Phi(x_{i+1})$. Hence,

$$\begin{aligned}\text{Amortized cost} &= \text{Real cost} - \Phi(x_i) + \Phi(x_{i+1}) \\ &= 1 + 2 \\ &= 3\end{aligned}$$

For an expensive operation i, we know that $i = 2^k + 1$.

	x_i	x_{i+1}
length	2^k	$2^k + 1$
capacity	2^k	$2^{k+1} + 1$
Φ	$2^k + 1$	3

Hence,

$$\begin{aligned}\text{Amortized cost} &= \text{Real cost} - \Phi(x_i) + \Phi(x_{i+1}) \\ &= 2^k + 1 - (2^k + 1) + 3 \\ &= 3\end{aligned}$$

1.4 Comparison between these 3 methods

1. Aggregate method
   - Intuitive
   - Does not work very well if there are multiple operations, and summations might not be straightforward.
     Example: For stacks (push/pop), you need to argue about any sequence of push and pop operations.
2. Charging method
   - Flexible as you can choose what operations to charge on and how much to charge.
   - Need to come up with these charging schemes.
3. Potential method

- Very flexible.
- Potential functions is not always easy to come up with.

In the potential method, the potential function reflects the current status of the data structure. It does not depend on how we got to this state. However, in the case of the charging method, the amount of money you have in the bank can depend on the history of operations.

2 Disjoint Sets

Problem: n elements, each in a separate set. We want to support the following operations:

1. Find: Given an element, find the set it is in.
2. Merge (or union): Given two elements, merge the two sets that these elements are in.

Recall: We use this data structure to implement Kruskal's algorithm.

We represent sets using trees. Each node represents an element and each subset is a tree. The "head" element is the root.
Find: find the root of the tree that this element belongs to.
Merge: merge the two corresponding trees into one tree.

Note: The trees may not be binary trees.

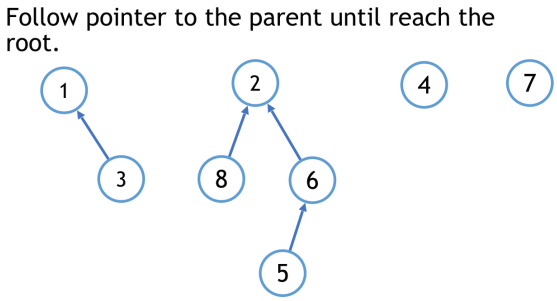


Figure 1: Find operation

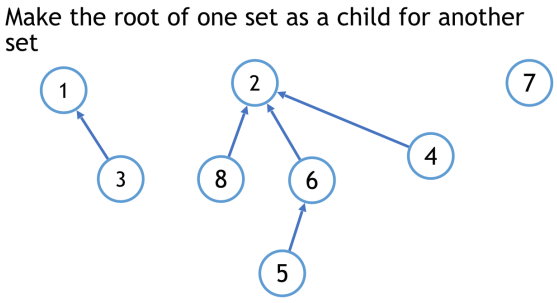


Figure 2: Merge operation

Running time:

- Find: Depth of the tree (naively). Worst-case depth could be n.
- Union: First need to perform two merge operations and then a $O(1)$ time operation to link the two trees (via their root).

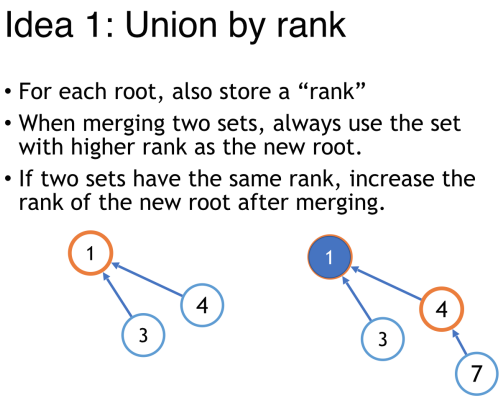


Figure 3: Union by rank

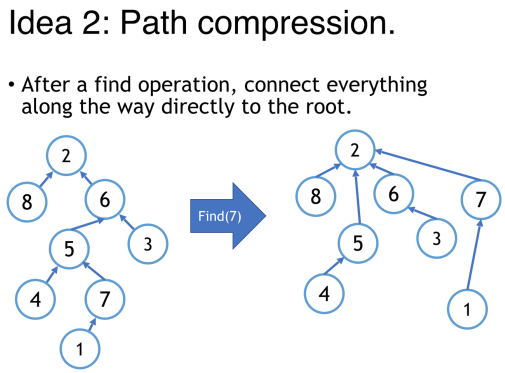


Figure 4: Path compression

Running time:

- Union by rank only:

  Depth is always bounded by $\log n$.
  $O(\log n)$ worst-case, $O(\log n)$ amortized.

- Union by rank + path compression:

  Worst case is still $O(\log n)$.
  Amortized $O(\alpha(n)) = o(\log^* n)$, where $\alpha(n)$ is the inverse-Ackerman function. $\log^* n$ is the number of times you need to take the log of n to make the output 1.