2 Week 3: February 12

2.1 Catenatble random-access deques

A **random-access deque** stores a sequences of items, supports all four standard deque operations, and also supports the following random-access operations:

- `Peek`(k): Return the kth item in the sequence, let's say counting counting from the back (the *push/pop* end).
- `Poke`(k, x): Replace the kth item in the sequence with item x.

If we implement a deque with a dynamically resizing array-list, each of these random-access operations can be supported in $O(1)$ worst-case time. (This is one of the most common reasons to prefer array-lists over pointer-based linked lists.)

Now suppose in addition to all these operations we want to support **concatenation**. That is, we would like to efficiently combine any two deques into a single deque by connecting the front end of one deque to the back end of the other. Notice that concatenation is not commutative: compare `HERENOW` with `NOWHERE`.

If we implement each deques as a doubly-linked list, we can easily support concatenation in $O(1)$ time by pointer manipulation, but linked lists do not support fast random access. How can we efficiently concatenate array-lists?

Let's try brute force! We can easily implement concatenation by moving the items of one sequence to the other, with one deletion and one insertion. Our only nod to efficiency is that we move the items in the *shorter* sequence to the *longer* sequence.

```
# concatenate deques A and B
Concat(A,B):
    if A.num < B.num
        while A.num > 0
            B.push(A.pull())
        return B
    else
        while B.num > 9
            A.shove(B.pop())
        return A
```

In the worst case, each concatenation takes $\Theta(n)$ time, where n is the total length of both strings. (The worst case happens when the two strings have approximately the same length.)

Amortized analysis of concatenations

Theorem: *Suppose we start with several deques with total length n. Every sequence of concatenations takes at most $O(n \log n)$ time.*

> **Proof:** For any positive integer m, let $\llbracket m \rrbracket = 2^{\lceil \log_2 m \rceil}$ denote the smallest power of 2 that is greater than or equal to m. If we maintain data arrays using doubling and halving, the *capacity* of any array-list storing m items is exactly $\llbracket m \rrbracket$.
>
> Consider an arbitrary item x in one of the deques. Each time a concatenation moves x from one array-list to another, the capacity of its new array-list is at least twice the

capacity of its old array-list. Thus, if the final array-list containing x has capacity k, then x is moved at most $\log_2 k$ times. The final array-list of x has capacity at most $\lceil n \rceil$. We conclude that x is moved at most $\log_2 \lceil n \rceil = \lceil \log_2 n \rceil = O(\log n)$ times. □

Corollary: *In any sequence of concatenations, starting with n deques of size 1, each concatenation takes $O(\log n)$ amortized time.*

. . . And Insertions

Now let's consider mixed sequences of insertions (pushes and shoves) and concatenations.

Theorem: *In any mixed sequence of n insertions and any number of concatenations, starting with a collection of empty deques, each insertion takes $O(\log n)$ amortized time, and each concatenation takes **zero** amortized time.*

> **Proof:** Again, consider any item x that is ever inserted into any deque. Each time x is moved into a different array-list, the capacity of the array-list containing x grows by a factor of 2. On the other hand, the capacity of the array-list containing x is always a power of 2 less than or equal to $\lceil n \rceil$. Thus, x is moved at most $O(\log n)$ times.
>
> We conclude that the total cost of all insertion and concatenation operations is $O(n \log n)$. We can charge all of this time to the n insertions! □

This result is somewhat counterintuitive. In terms of actual *wall-clock* time, concatenations are far slower than insertions, but in terms of *amortized* time, insertions are more expensive and concatenations are absolutely free! It may help to remember that **amortized analysis is an accounting trick**. Our theorem does not claim that concatenations are *actually* faster than insertions; it only claims that *for purposes of analyzing sequences of operations*, we can *pretend* that concatenations are faster than insertions.

*. . . And Deletions! (The Potential Method)

Finally, let's consider the most general setting. Here I'm going to switch from summation/charging arguments to a more powerful (but also more difficult) method of amortized analysis that builds on the metaphor of "potential energy". Instead of describing an explicit schedule of costs or taxes associated with each operation, or each component of the data structure, we represent prepaid work as *potential* that can be used to perform later expensive operations. The potential is a function of the entire data structure.

Fix a sequence of N operations on some data structure. For each index i, let T_i denote the actual wall-clock time to perform the ith operation in the sequence, and let Φ_i denote the potential of the data structure after the $i$ht iteration. In particular, Φ_0 is the starting potential of the data structure. Then we can define the amortized time A_i for the ith operation to be the actual time plus the increase in potential:

$$\boxed{A_i := T_i + \Phi_i - \Phi_{i-1}}$$

So the *total* amortized time for the entire sequence of N operations is the *total* actual time plus the *total* increase in potential. In the following summation, most of the potential terms Φ_i appear

once positively and once negatively and therefore cancel out.

$$\sum_{i=1}^{N} A_i \;=\; \sum_{i=1}^{N} (T_i + \Phi_i - \Phi_{i-1}) \;=\; \sum_{i=1}^{N} T_i \;+\; \Phi_N - \Phi_0.$$

A potential function is *valid* if $\Phi_i - \Phi_0 \geq 0$ for all i. If the potential function is valid, then the total *actual* time to execute any sequence of operations is always less than or equal to the total *amortized* time for that sequence:

$$\sum_{i=1}^{N} T_i \;=\; \sum_{i=1}^{N} A_i - (\Phi_N - \Phi_0) \;\leq\; \sum_{i=1}^{N} A_i.$$

If we're comfortable with charging arguments, we can think of the potential Φ as the total amount of money in our bank account: charges that past operations have paid but not yet spent. In general, however, there may be no natural way to interpret change in potential as "taxes" or "charges". Taxation and charging are useful when there is a convenient way to allocate costs to specific steps in the algorithms or specific components of the data structure. Potential arguments allow us to argue more globally when a local allocation is awkward or impossible.

Theorem: *In any mixed sequence of N insertions and any number of deletions and concatenations, starting with a collection of empty deques, each insertion takes $O(\log N)$ amortized time, each deletion takes $O(1)$ amortized time, and each concatenation takes **zero** amortized time.*

> **Proof:** To simplify notation, I'll assume N is a power of 2; otherwise, we can just add $\llbracket N \rrbracket - N$ dummy operations at the end of the sequence.
>
> To make the analysis concrete, I will use the the number of *array-write* instructions of the form $q.\mathsf{data}[i] \leftarrow x$ as a proxy for running time. Hopefully it is clear that the actual running time of any operation is $O(\#\text{array-writes} + 1)$. Finally, to simplify the analysis, I will assume without loss of generality that arrays do not double during a concatenation; if necessary we can decompose any operation $\mathsf{Concat}(A, B)$ into three operations of the form $\mathsf{Concat}(A_1, B)$, $\mathsf{Resize}(B)$, and $\mathsf{Concat}(A_2, B)$.
>
> Our data structure consists of a collection of deques, each with a size num and a capacity cap. I will define the potential of each deque as the sum $\iota + \delta + \kappa$ of three terms defined as follows:
>
> - $\iota = \max\{0, 2 \cdot \mathsf{num} - \mathsf{cap}\}$ (insertion credits)
> - $\delta = 4 \cdot \max\{0, \mathsf{cap}/2 - \mathsf{num}\}$ (deletion credits)
> - $\kappa = 3 \cdot \mathsf{num} \cdot \log_2(N/\mathsf{cap})$ (concatenation credits)
>
> The potential Φ of the entire data structure is the sum of the potentials of its component deques.
>
> Let's consider how different operations changes these three functions:
>
> - insertion without doubling: num increases by 1, so
>   - ι **increases** by at most 2,
>   - δ might decrease but does not increase,
>   - κ **increases** by $3 \log_2(N/\mathsf{cap})$,
>   - the actual time is dominated by 1 array-write,

- so the amortized number of array-writes is at most $3\log_2(N/\mathsf{cap}) + 1 = O(\log N)$.

- deletion without halving: num increases by 1, so
  - ι might decrease but does not increase,
  - δ **increases** by at most 4,
  - κ decreases,
  - the actual time is dominated by 1 array-write,
  - so the amortized number of array-writes is at most $3 = O(1)$.

- concatenation without doubling: Assume the two deques have sizes $\mathsf{num} > \mathsf{num}'$. Replace num and num' with $\mathsf{num} + \mathsf{num}'$, and replace cap and cap' with $\max\{\mathsf{cap}, \mathsf{cap}'\}$.
  - ι **increases** by at most $2 \cdot \mathsf{num}'$,
  - δ might decrease but does not increase,
  - κ decreases by $3 \cdot \mathsf{num}'$,
  - the actual time is dominated by num' array-writes,
  - so the amortised number of array-writes is ***at most zero***.

- halve a quarter-full data array: num increases by 1 and cap halves, so
  - ι does not change,
  - δ decreases by $4 \cdot \mathsf{num}$,
  - κ **increases** by $3 \cdot \mathsf{num}$,
  - actual time is dominated by num array-writes,
  - so the amortized number of array-writes is ***zero***.

- double a full data array: num increases by 1 and cap doubles, so
  - ι decreases by num,
  - δ does not change,
  - κ decreases by $3 \cdot \mathsf{num}$,
  - the actual time is dominated by num array-writes,
  - so the amortized number of array-writes is ***negative!***

And this completes the proof! □

2.2 Tombstones

2.2.1 Static dictionaries (aka associative arrays)

Use sorted arrays; find by binary search.

Lazy Deletion: Tombstones

Suppose we *only* need to support deletions, not insertions.

To delete item x, mark x as "deleted". We can still use x for purposes of navigation, to guide our binary search. But if the binary search finds a marked item, we report that that item is *not* in the dictionary. Even though the item (or at least the key) is still stored in memory, it is invisible to the data structure's user.

To save space, we rebuild a clean copy of the sorted array whenever more than half of its elements are marked for death. We can charge the time to clean up and rebuild to the deletion operations

that marked the dead items, so each deletion runs in $O(1)$ amortized time.

*Lazy insertion: The Logarithmic Method

See the last question in Homework 3.