1 Vector ADT

The Vector is an Abstract Data Type corresponding to generalising the notion of the **Array** (concrete data type). The **key idea**: The *index* of an entry in an array can be thought of as the number of elements preceding it.
For examples: `http://www.cse.unsw.edu.au/~cs2011/lect/05_Stacks4b.pdf`

1.1 Performance

- In the array based implementation of a Vector
  - The space used by the data structure is $\mathbf{O(n)}$
  - `size, isEmpty, elemAtRank` and `replaceAtRank` run in $\mathbf{O(1)}$ time
  - insertAtRank and removeAtRank run in $\mathbf{O(n)}$ time (Need to shift back the rest of the list)
- If we use the array in a circular fashion (see lectures on queues)
  - `insertAtRank(0)` and `removeAtRank(0)` run in $\mathbf{O(1)}$ time
  - In an `insertAtRank` operation, when the array is full, instead of throwing an exception, we can replace the array with a **larger one**

1.2 Growable Array-based Vector

In a push (`insertAtRank(t)`) operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one. Strategies:

- *incremental strategy*: increase size by a constant c
- *doubling strategy*: double the size

1.2.1 Amortised vs average case analysis

- *Amortised*: **real sequence** of **dependent** operations
- *Average*: Set of (possibly **independent**) operations

Suppose some individual operation (such as *push*) takes time T in the **worst-case**. Suppose do a **sequence** of operations, and there are s such operations taking time T_s. Then $s * T_s$ is an *upper-bound* (small-oh) for the total time however, such an *upper-bound* might not ever occur. The time T_s might well be $\mathtt{o}(s * T_s)$ even in the worst-case.The **average time** per operation, T_s/s is sometimes the most **relevant quantity** in practice

1.3 Incremental vector capacity

If we increase array using incremental method:

- We replace array $k = n/c$ times
- The total time `T(n)` would be proportional to $n + c * k(k+1)/2$
- Because c is a constanct, runtime would be $O(n^2)$, which means that amortised time for 1 push would be $O(n^2)/n$ which is $O(n)$

1.4 Doubling vector capacity

For every push of cost $O(n)$ we will be able to do another $O(n)$ pushes of cost $O(1)$ before having to resize again. So the cost on resizing can be *amortised* over n other $O(1)$ operations and give an average of $O(1)$ per operation.

- We replace the array $k = log_2 n$ times (as we double it's 2x every thime, thus log2)
- $n + 1 + 2 + 4 + 8 + ... + 2^k - 1 = n + 2^k - 1 = 2n - 1$
- $T(n)$ is $O(n)$
- The amortized time of a push operation is $O(1)$
- That is, no worse than if all the needed memory was **pre-assigned**!

Reference section

amortized analysis

Amortized time is often used when stating algorithm complexity. Instead of giving values for worst-case performance it provides an average performance. Amortized time looks at an algorithm from the viewpoint of total running time rather than individual operations.