Solution 1: Dynamic Array

The *amortized cost* is the average time per operation, evaluated over a sequence of operations, when individual operations may have varying costs. There are different ways to derive the amortized cost. Using *aggregate analysis*, the amortized cost is the average cost per operation over a sequence of operations. Given a sequence of n operations with total cost $T(n)$, the amortized cost per operation is $T(n)/n$. It is usually expressed using Landau notation, so e.g. $\Theta(f(n))$.

We are going to make use of the following operations, with their respective costs:

- $A \leftarrow$ **allocate**(n): Create an empty Array of size n. Allocation costs $\Theta(1)$. The array size n can then be queried with the **length**(A) function (which itself is $\Theta(1)$).
- **copy(**A**,** B**)**: Copy the contents of Array A to Array B. Cost: $\Theta(\text{length}(A))$.
- $A[i] \leftarrow x$: Set the value of A at the ith position to x. Cost: $\Theta(1)$.
- $A \leftarrow B$: Replace the array that is referenced by A with the array that is referenced by B, and deallocate the original array. This operation does not involve any moving of data, it basically just renames the array. The cost is therefore $\Theta(1)$.

Consider the following algorithms for a dynamically growing array, i.e. an array that provides an `append` method that can add an unlimited number of elements. For each of these, determine the costs of the `append` operation as $\Theta(f(n))$ in terms of the current number of elements n in the array. You should find the asymptotic *best-case* and *worst-case* cost of adding an element to an array of size n, as well as the asymptotic amortized cost of adding elements up to size n.

(a)

Algorithm 1 GrowingArray1

```
$A \leftarrow$ ALLOCATE(0), $n \leftarrow 0$
function APPEND($x$)
    $n \leftarrow n + 1$
    $A' \leftarrow$ ALLOCATE($n$)
    COPY($A$, $A'$)
    $A \leftarrow A'$
    $A[n] \leftarrow x$
end function
```

(b)

Algorithm 2 GrowingArray2

```
$A \leftarrow$ ALLOCATE(100), $n \leftarrow 0$
function APPEND($x$)
    $n \leftarrow n + 1$
    if $n >$ LENGTH($A$) then
        $A' \leftarrow$ ALLOCATE($n + 100$)
        COPY($A$, $A'$)
        $A \leftarrow A'$
    end if
    $A[n] \leftarrow x$
end function
```

Algorithm 3 GrowingArray3

(c)
```
$A \leftarrow$ ALLOCATE(1), $n \leftarrow 0$
function APPEND($x$)
    $n \leftarrow n + 1$
    if $n >$ LENGTH($A$) then
        $l \leftarrow$ LENGTH($A$)
        $A' \leftarrow$ ALLOCATE($2 \cdot l$)
        COPY($A$, $A'$)
        $A \leftarrow A'$
    end if
    $A[n] \leftarrow x$
end function
```

Algorithm 4 GrowingArray4

(d)
```
$A \leftarrow$ ALLOCATE(100), $n \leftarrow 0$
function APPEND($x$)
    $n \leftarrow n + 1$
    if $n >$ LENGTH($A$) then
        $l \leftarrow$ LENGTH($A$)
        $A' \leftarrow$ ALLOCATE($\lceil 1.1 \cdot l \rceil$)
        COPY($A$, $A'$)
        $A \leftarrow A'$
    end if
    $A[n] \leftarrow x$
end function
```

Here, $\lceil x \rceil$ denotes rounding up x to the next integer.

We analyze the costs of the `append` operation for different dynamic array strategies in terms of n, where n is the number of elements in the array after the `n <- n + 1` step within the `append` procedure. Thus, the element being appended is the n^{th} element, and the array contained $n-1$ elements before this specific operation began to modify the array structure.

We now analyze each algorithm:

(a) **Algorithm 1**: Appending causes allocation of a new array of size n and copies all $n-1$ existing elements.

- **Worst-case cost**: Each `append` operation involves allocating a new array of size n (cost $\Theta(1)$), copying the previous $n-1$ elements (cost $\Theta(n)$), reassigning A (cost $\Theta(1)$), and assigning the new element (cost $\Theta(1)$). Thus, the total cost is $\Theta(1) + \Theta(n) + \Theta(1) + \Theta(1) = \Theta(n)$.
- **Best-case cost**: Same as the worst-case, as all steps are performed for every append. Cost: $\Theta(n)$.
- **Amortized cost**: Consider a sequence of M `append` operations. The k^{th} operation costs $\Theta(k)$. The total cost for M operations is bounded (above and below) by $\sum_{k=1}^{M}(c_0 + c_1 \cdot k)$ for some constants c_0, c_1; $c_2 > 0$. This sum is $c_0 M + c_1 \frac{M(M+1)}{2} = \Theta(M^2)$. The amortized cost per operation is $T(M)/M = \Theta(M^2)/M = \Theta(M)$. In terms of n (the number of elements after M operations, so $n = M$): Amortized cost is $\Theta(n)$.

(b) **Algorithm 2**: Appending adds to the current array if space is available. If $n >$ length(A) (current capacity), a new array of size $n + 100$ is allocated, elements are copied, and then the new element is added. Capacity increases by a fixed amount (100) upon resize.

Worst-case cost: Occurs when a resize is needed ($n >$ length(A)). The cost includes allocating a new array (size $n + 100$, cost $\Theta(1)$), copying $n-1$ elements from an array of capacity $L_{old} = n - 1$ (cost $\Theta(n)$), reassigning A (cost $\Theta(1)$), and assigning the new element (cost $\Theta(1)$). Total cost: $\Theta(n)$.

Best-case cost: Occurs when no resize is needed ($n \leq$ length(A)). This involves incrementing n and assigning $A[n] \leftarrow x$. Cost: $\Theta(1)$.

Amortized cost: Let $C = 100$ be the fixed increment to capacity (and also the initial capacity). A resize occurs approximately every C operations. The k^{th} resize occurs when roughly kC elements have been inserted. The cost of this k^{th} resize (copy part) is $\Theta(kC)$ because the capacity being copied is $\approx kC$. Consider M append operations. There are M base operations costing $\Theta(1)$ each (total $\Theta(M)$). The number of resizes is roughly M/C. The total cost of copying due to resizes is $\sum_{j=0}^{(M/C)-1} \Theta(jC) = \Theta(C \sum_{j=0}^{(M/C)-1} j) = \Theta(C \cdot \frac{((M/C)-1)(M/C)}{2}) = \Theta(C \cdot (M/C)^2) = \Theta(M^2/C)$. The total cost $T(M) = \Theta(M) + \Theta(M^2/C)$. Since C is a constant (100), $T(M) = \Theta(M^2)$. The amortized cost is $T(M)/M = \Theta(M)$. In terms of n (the number of elements after M operations, so $n = M$): Amortized cost is $\Theta(n)$.

(c) **Algorithm 3**: Capacity is doubled when a resize is needed.

Worst-case cost: Occurs when a resize is needed ($n >$ length(A)). Let $L_{old} =$ length(A) be the old capacity. The number of elements $n-1$ is equal to L_{old}. A new array of size $2L_{old}$ is allocated (cost $\Theta(1)$). Copying $n-1$ elements from capacity $L_{old} = n-1$ costs $\Theta(n-1)$. Reassigning A and assigning $A[n]$ costs $\Theta(1)$. Total cost: $\Theta(n)$.

Best-case cost: Occurs when no resize is needed. Cost: $\Theta(1)$.

Amortized cost: Consider M append operations. The total cost of the M basic insertion steps (increment n, $A[n] \leftarrow x$) is $\Theta(M)$. Resizes occur when $n-1$ (number of elements before append) is a power of 2: $1, 2, 4, 8, \ldots, 2^k$. The element $n = 2^k + 1$ triggers the resize. The capacity copied at these points is $1, 2, 4, \ldots, 2^k$. So copy costs are $\Theta(1), \Theta(2), \Theta(4), \ldots, \Theta(2^k)$. If M elements are appended, the largest power of 2 less than or equal to $M-1$ is $2^{\lfloor \log_2(M-1) \rfloor}$. The total cost of copying is $\sum_{j=0}^{\lfloor \log_2(M-1) \rfloor} \Theta(2^j) = \Theta(2^{\lfloor \log_2(M-1) \rfloor + 1} - 1) = \Theta(M)$. The total cost for M operations $T(M) = \Theta(M)$ (base steps) $+ \Theta(M)$ (copying) $= \Theta(M)$. The amortized cost is $T(M)/M = \Theta(1)$.

(d) **Algorithm 4**: Capacity is multiplied by 1.1 (and rounded up) when a resize is needed.

Worst-case cost: Occurs when a resize is needed. Let $L_{old} =$ length(A). Cost of copy is $\Theta(L_{old})$. Since $n - 1 = L_{old}$, this is $\Theta(n-1) = \Theta(n)$. Total cost for append is $\Theta(n)$.

Best-case cost: Occurs when no resize is needed. Cost: $\Theta(1)$.

Amortized cost: Let $c = 1.1$ be the growth factor. Consider a sequence of M `append` operations. The total cost of the M basic insertion steps (increment n, $A[n] \leftarrow x$) is $\Theta(M)$. Let L_0 be the initial capacity ($L_0 = 100$, in this case). When a resize occurs, if the old capacity was L_{old}, the new capacity is $L_{new} = \lceil cL_{old} \rceil \geq cL_{old}$. The cost of copying L_{old} elements is $\Theta(L_{old})$. The capacities of the array form a geometric progression, approximately $L_0, cL_0, c^2L_0, \ldots, c^kL_0$. Suppose M operations are performed. The largest capacity reached, $L_{final} \approx c^kL_0$, will be roughly proportional to M. So, $c^kL_0 \approx M$, which means $k \approx \log_c(M/L_0)$. The total cost of all copying operations is the sum of costs at each resize: $\sum_{j=0}^{k-1} \Theta(c^jL_0)$. (Assuming k resizes occurred to reach a capacity that holds M elements). This sum is $\Theta(L_0 \sum_{j=0}^{k-1} c^j) = \Theta(L_0 \frac{c^k-1}{c-1})$. Since $c^kL_0 \approx M$, the total copy cost is $\Theta(L_0 \frac{M/L_0 - 1/L_0}{c-1}) = \Theta(\frac{M-1}{c-1})$. As $c = 1.1 > 1$, $c - 1$ is a positive constant. Thus, the total cost of copying is $\Theta(M)$. The total cost for M operations is $T(M) =$ cost of base steps$+$total cost of copies $= \Theta(M) + \Theta(M) = \Theta(M)$. Therefore, the amortized cost per operation is $T(M)/M = \Theta(M)/M = \Theta(1)$.

Solution 2: Queue

Write an R6 class that implements a queue with a maximum capacity of 10 elements. The class should have the following methods:

- `enqueue(x)`: add an element (of any type) to the queue
- `dequeue()`: remove and return the oldest element from the queue
- `head()`: return the oldest element from the queue
- `tail()`: return the newest element from the queue
- `size()`: return the number of elements in the queue

`enqueue` should throw an error if the queue is full. `dequeue`, `head`, and `tail` should throw an error if the queue is empty. Internally, the class should use a list to store the elements. It should be efficient in terms of adding and removing elements from the queue, i.e. it should not re-allocate the list for every operation.

The `Queue` R6 class implements a First-In-First-Out (FIFO) data structure with a fixed maximum capacity of 10 elements, as specified in the exercise. It provides methods for enqueueing an element, dequeueing an element, viewing the head and tail elements, and querying the current size of the queue.

```r
Queue <- R6::R6Class("Queue",
  private = list(
    # it is good practice to have these things 'private', but it is not required for this exercise
    element.store = NULL,
    max.capacity = NULL,
    head.pointer = 0L,      # Points to the first element (0-indexed)
    tail.pointer = 0L       # Points to the next available freeslot (0-indexed)
  ),
  public = list(
    initialize = function(max.capacity = 10L) {
      # the exercise wants to have max capacity 10, but it does not hurt to make it flexible
      private$max.capacity <- checkmate::assertCount(max.capacity, positive = TRUE)
      private$element.store <- vector(
        mode = "list",
        length = private$max.capacity
      )
    },

    enqueue = function(x) {
      if (private$tail.pointer - private$head.pointer >= private$max.capacity) {
        stop("Queue is full.")
      }
      # the following is necessary to avoid removing list elements when `x` is NULL.
      private$element.store[(private$tail.pointer %% private$max.capacity) + 1L] <- list(x)
      private$tail.pointer <- private$tail.pointer + 1L
      invisible(self)
    },

    dequeue = function() {
      result <- self$head()
      private$head.pointer <- private$head.pointer + 1L
      return(result)
    },

    head = function() {
      if (private$head.pointer == private$tail.pointer) {
        stop("Queue is empty.")
      }
      return(private$element.store[[(private$head.pointer %% private$max.capacity) + 1L]])
    },

    tail = function() {
      if (private$head.pointer == private$tail.pointer) {
        stop("Queue is empty.")
      }
      # the tail is the element at the slot *before* the current tail.pointer.
      # We need to subtract 1 before the modulo operation, then add it again after.
      return(private$element.store[[(private$tail.pointer - 1L) %% private$max.capacity + 1L]])
    },

    size = function() {
      return(private$tail.pointer - private$head.pointer)
    }
  )
)
```