CC4102 **Diseño y Análisis de Algoritmos** **2021A**

Jérémy Barbay, Javier Oliva

Ejercicio Formativo 7

(Fecha de entrega: [2021-05-11 Tue])

Antonia Sofía Labarca Sánchez

Supuesto: con el costo de espacio amortizado a lo más $4n$ en verdad se refiere a $4n + \Theta(1)$, ya que se debe almacenar algunos números y punteros para que la estructura de datos funcione.

La implementación está basada en la de **acá**, solo que modificando qué ocurre cuando se llena el arreglo y revisando el espacio ocupado cuando se borra algo para ver si se puede achicar.

```
// C or C++ program for insertion and deletion in Dynamic Circular Queue
#include<iostream>
using namespace std;

struct Queue
{
    // Values:
    int rear, front, occup, size;
    // Constants:
    int beta;
    double alpha;
    // Pointers:
    int *arr;
    int *arr2;

    Queue(int s)
    {
       front = rear = -1;
       occup = 0;
       size = s;
       arr = new int[s];
       beta = 4;
       alpha = (beta + 1.0)/2.0
    }

    void push(int value);
    int pull();
    void displayQueue();
};

/* Function to add an element to the queue */
void Queue::push(int value)
{
    if (front == 0 && rear == size-1)
    {
        printf("\nQueue is Full");
        arr2 = new int[(int)size*alpha];

        for (int i = front; i <= rear; ++i)
        {
            arr2[i] = arr[i];
        }

        rear++;
        arr2[rear] = value;
        size = (int)size*alpha;

        delete[] arr;
```

```
        arr = arr2;
    }

    else if (rear == front-1)
    {
        printf("\nQueue is Full");
        arr2 = new int[(int)size*alpha];

        for (int i = 0; i < size-front; ++i)
        {
            arr2[i] = arr[i+front];
        }
        for (int i = 0; i <= front; ++i)
        {
            arr2[size-front+i] = arr[i];
        }

        front = 0;
        rear = occup;
        arr2[rear] = value;
        size = (int)size*alpha;

        delete[] arr;
        arr = arr2;
    }

    else if (front == -1) /* Insert First Element */
    {
        front = rear = 0;
        arr[rear] = value;
    }

    else if (rear == size-1 && front != 0)
    {
        rear = 0;
        arr[rear] = value;
    }

    else
    {
        rear++;
        arr[rear] = value;
    }

    occup++;
}

// Function to delete element from Queue
int Queue::pull()
{
    if (front == -1)
    {
        printf("\nQueue is Empty");
        return INT_MIN;
    }

    int data = arr[front];
    arr[front] = -1;
    if (front == rear)
    {
        front = -1;
        rear = -1;
    }
    else if (front == size-1)
        front = 0;
```

```
    else
        front++;

    occup--;

    // Check if there is a lot of empty space
    if (occup*beta <= size)
    {
        arr2 = new int[occup*alpha];

        if (front <= rear)
        {
            for (int i = 0; i <= rear - front; ++i)
            {
                arr2[i] = arr[front + i];
            }
            rear = rear - front;
            front = 0;
        }
        else
        {
            for (int i = front; i < size; ++i)
            {
                arr2[i-front] = arr[i];
            }
            int start = size - front;
            for (int i = 0; i <=rear; ++i)
            {
                arr2[start + i] = arr[i];
            }
            rear = occup - 1;
            front = 0;
        }
        delete[] arr;
        arr = arr2;
        size = occup*alpha;
    }

    return data;
}

// Function displaying the elements
// of Circular Queue
void Queue::displayQueue()
{
    if (front == -1)
    {
        printf("\nQueue is Empty");
        return;
    }
    printf("\nElements in Circular Queue are: ");
    if (rear >= front)
    {
        for (int i = front; i <= rear; i++)
            printf("%d ",arr[i]);
    }
    else
    {
        for (int i = front; i < size; i++)
            printf("%d ", arr[i]);

        for (int i = 0; i <= rear; i++)
            printf("%d ", arr[i]);
    }
}
```

La idea de la estructura es la misma que la de una Circular Queue, pero al llenarse se crea un nuevo arreglo del doble de tamaño y se copia lo que ya se tenía. También, al hacer pull se revisa la cantidad de espacio ocupado realmente. Si es mucho menor al tamaño (4 veces menor, para asegurar que el costo en espacio sea $4n$, se crea un array de la mitad de tamaño y se copia lo que se tenía.

El análisis de esta estructura es similar al que se hace en el apunte para realocaciones de arreglos permitiendo contracciones.

Sea s el tamaño del arreglo, n la cantidad actual de elementos. Notemos que se achica cuando $s \geq 4n$, y queda de tamaño $2n$ (la mitad del espacio que ocupaba originalmente). Esto nos garantiza que el máximo de memoria usado por el arreglo es $4n$ (excepto por los momentos en el que se tienen el arreglo original y la copia de distinto tamaño y se está re-escribiendo, pero esto ocurre muy poco, y en el apunte se desprecia esto).

Se considerará la función de potencial $\phi = |an - bs|$. Se tienen los siguientes casos:

1. **push de un valor sin agrandar:** el costo consiste en revisar que se puede (a lo más 2 comparaciones) cuál es la posición en la que se debe agregar (a lo más 4 comparaciones), actualizar el valor de rear, occup, y si es necesario front (a lo más 3 operaciones) y escribir el valor en la posición correcta (una escritura). Luego $c_i \leq 10$. El valor de $\Delta\phi_i$ es a lo más a, por lo que $\hat{c}_i \leq 10 + a = \Theta(1) + a$.
2. **pull de un valor sin achicar:** el costo consiste en extraer el elemento (1 operación), escribir un -1 en la posición correspondiente (1 operación), y actualizar el valor de front, occup, y si es necesario rear (a lo más 3 operaciones, calcular size-1 y 2 comparaciones). Luego $c_i \leq 8$. El valor de $\Delta\phi_i$ es a lo más a, por lo que la cota superior $\hat{c}_i \leq \Theta(1) + a$ se mantiene.
3. **agrandar el array y push:** crear un nuevo array, y copiar todos los elementos que ya se tenían y el que se quiere agregar (n escrituras). Ocurre cuando $n = s$, luego el potencial es $\phi_{i-1} = |an - bn|$ y $\phi_i = |an - \alpha bn|$. Debe tenerse que $an - \alpha bn \geq 0$, es decir, que $a \geq \alpha b$. Luego $\Delta\phi_i = -b(\alpha - 1)n$ y $\hat{c}_i \leq 0$ si $b \geq \frac{1}{\alpha-1}$.
4. **achicar el array y pull:** ocurre cuando $s \geq 2n$, y los potenciales son $\phi_{i-1} = |an - bs|$ y $\phi_i = |an - \alpha bn|$. Esto debe darse solo si $an - \alpha bn \leq 0$, es decir que $a \leq \alpha b$. Luego $\phi_{i-1} = bs - an \geq \beta bn - an$ y $\phi_i = \alpha bn - an$, luego $\Delta\phi_i \leq -b(\beta - \alpha)n$. Por lo tanto, $\hat{c}_i \leq 0$ si $b \geq \frac{1}{\beta-\gamma}$.

De lo anterior se obtienen las condiciones:

$$b\alpha \leq a \leq b\alpha \implies a = b\alpha$$
$$b \geq \frac{1}{\alpha - 1}$$
$$b \geq \frac{1}{\beta - \alpha}$$

Luego, el objetivo es minimizar a, ya que el costo amortizado está acotado por $O(1) + a$. Como $a = b\alpha$, el objetivo es minimizar b, para lo cual notemos que está acotado inferiormente por $\frac{1}{\beta-\alpha}$ y por $\frac{1}{\beta-\alpha}$.

Reemplazando por $\beta = 4$ y $\alpha = \frac{\beta+1}{2} = \frac{5}{2}$, se obtiene que $b \geq \frac{1}{\frac{5}{2}-1} = \frac{1}{\frac{3}{2}} = \frac{2}{3}$ y que $b \geq \frac{1}{4-\frac{5}{2}} = \frac{1}{\frac{3}{2}} = \frac{2}{3}$. Entonces, escogemos $b = \frac{2}{3}$ para minimizar.

Luego, el costo amortizado considerando lo anterior es de $\hat{c} = O(1) + a = O(1) + b\alpha = O(1) + \frac{2}{3}\frac{5}{2} = O(1)$, es decir, es constante.

Por otro lado, el espacio amortizado ocupado por el arreglo trivialmente está acotado por $4n$, ya que cuando se llega a ese espacio se achica a uno de tamaño $\frac{5}{2}n$.

Nota: capaz quedaba más bonito y convencional ocupando $\alpha = 2$, pero se realizan un poco más de operaciones de agrandar, ya que se agranda a algo un poco más chico. De todas formas el costo amortiguado queda constante, ya que se tendría que está acotado por $O(1)+2b = O(1)+2(\max \frac{1}{2-1}, \frac{1}{4-2} \leq O(1)+2 = O(1)$.