APUNTES PARA ALGORITMOS Y ESTRUCTURAS DE DATOS II

PARTE 2: ESTRUCTURAS DE DATOS

Implementación del TAD cola de prioridades con listas enlazadas. Hemos logrado implementaciones eficientes de los TADs contador, pila y cola. En efecto, si bien algunas implementaciones daban lugar a operaciones lineales, pronto encontramos variantes que permitieron implementar el TAD con operaciones constantes.

No es el caso de la cola de prioridades. Las implementaciones más eficientes que existen requieren operaciones logarítmicas. Las veremos pronto, cuando presentemos árboles binarios.

En esta sección veremos qué se obtiene si adaptamos las implementaciones de colas para implementar colas de prioridades.

Por ejemplo, si tomamos la implementación de cola que usa listas enlazadas circulares, el problema aparece al definir la función first, que debe devolver el mayor elemento de la cola de prioridades y para ello, debe recorrerla toda, y esto la vuelve lineal. La operación dequeue, que debe eliminar dicho elemento, también es lineal. Otra posibilidad sería mantener la cola ordenada. En ese caso la función first y el procedimiento dequeue serían constantes, pero el procedimiento enqueue sería lineal.

Consideraciones similares pueden hacerse con la implementación de cola en un arreglo, con el agravante de que parece necesario desplazar los elementos del arreglo al insertar o borrar un elemento.

Tomamos como ejemplo la primer posibilidad: usamos listas enlazadas circulares y los procedimientos empty y enqueue y la función is_empty se mantienen idénticas que para cola (cambiando en todos lados queue por pqueue). Es decir que la lista enlazada circular no se mantiene ordenada. La función first debería entonces recorrer toda la lista enlazada:

```
{Pre: p ∼ Q ∧¬is_empty(q)}
fun first(p:pqueue) ret e:elem
    var r: pointer to node                {el puntero r para recorrer la lista enlazada}
    r:= p→next                                                  {⋆r es el primer nodo}
    e:= r→value                                        {por ahora éste es el máximo}
    while r ≠ p do                                  {mientras ⋆r no sea el último nodo}
        r:= r→next                            {que r pase a señalar el nodo siguiente}
        if e < r→value then e:= r→value fi          {que e sea el nuevo máximo}
    od
end
{Post: e ∼ primero(Q)}
```

El procedimiento dequeue es más complicado. Requiere eliminar el nodo donde se encuentra el máximo. Para ello, debemos recordar la dirección del nodo anterior a ese para poder modificar su campo next de modo de saltearlo. Utilizamos la misma variable

r para recorrer, pero esta vez r va a alojar todo el tiempo la dirección del nodo anterior al que se está observando. Además es necesario recordar el nodo anterior a donde se encuentra el máximo (para ello usamos el puntero pmax). También es necesario tratar separadamente el caso de la cola con un solo elemento.

```
{Pre: p ∼ Q ∧¬is_empty(p)}
proc dequeue(in/out p:pqueue)
    var r, pmax: pointer to node
    r:= p                                          {⋆(r→next) es el primer nodo}
    pmax:= p                                {por ahora pmax→next→value es el máximo}
    if p = p→next then p:= null                      {caso cola con un solo elemento}
    else while r→next ≠ p do                {mientras ⋆(r→next) no sea el último nodo}
            r:= r→next                            {que r pase a señalar el nodo siguiente}
            if pmax→next→value < r→next→value
            then pmax:= r                    {pmax→next→value es el nuevo máximo}
            fi
        od                                        {ahora hay que saltear ⋆(pmax→next)}
        r:= pmax→next
        pmax→next := r→next
        p:= pmax          {sólo es útil cuando r = p, pero se puede hacer siempre}
    fi
    free(r)
end
{Post: p ∼ decolar(Q)}
```

Como anticipamos estas operaciones resultan lineales, pero no es fácil encontrar una implementación mejor. Mantener la lista enlazada ordenada volvería constantes estas operaciones, pero lineal el procedimiento enqueue. Recién obtendremos una implementación mejor utilizando heaps, cuando veamos árboles binarios.

De todas formas, las implementaciones anteriores son ejemplos interesantes que ilustran las dificultades del uso de punteros para recorrer estructuras enlazadas. En este caso, r y pmax se utilizan para señalar nodos anteriores a los que pareciera lógico. Esto es por la necesidad de eliminar un nodo, hace falta contar con la dirección del nodo anterior para modificar su campo next. Esto complica la lógica de todo el procedimiento.

La asignación p:= pmax sólo se necesita en caso de que el máximo sea el último, es decir, el valor del nodo ⋆p. En ese caso el dequeue va a borrar ese nodo y por lo tanto el puntero p, es decir la cola, debe apuntar a otro nodo. La asignación p:= pmax elige que apunte al anterior al que se está borrando. Tratándose de listas circulares en las que el orden no importa, la asignación puede hacerse siempre.

Se puede observar que la condición del **while** es verdadera la primera vez que se evalúa, ya que es consecuencia de que no se cumpla la condición del **if** externo. Por ello, puede reemplazarse **while** r→next ≠ p **do** ... **od** por **repeat** ... **until** r→next = p.