ESTRUCTURAS DE DATOS Y ALGORITMOS

CURSO 2009 – PRÁCTICO 6

SOLUCIÓN

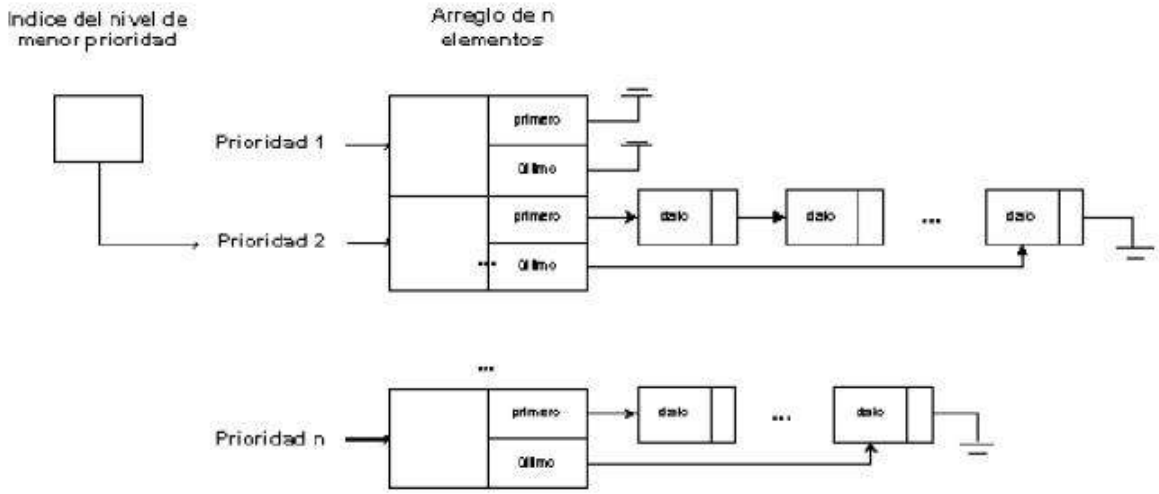
Ejercicio 8.a)

El TAD Cola de Prioridad con n niveles dispone de las siguientes operaciones:

```
/* Se crea una nueva Cola de Prioridad con un número de
 * niveles = cantNiveles */
CPrioridad* CreateCP(int cantNiveles);

/* Se encola el elemento dato, con prioridad en la Cola de
 * Prioridad cp.
 * En caso de que ya exista un dato con la prioridad
 * cp en la cola, el criterio de desempate utilizado es FIFO.
 * Pre: DarNiveles(cp) >= prioridad */
void Insert(CPrioridad *& cp, int dato, int prioridad);

/* Retorna el número de niveles asociados a la Cola de Prioridad cp */
int DarNiveles(CPrioridad *cp);

/* Retorna TRUE si la Cola de Prioridad cp es vacía, FALSE en
 * cualquier otro caso */
bool IsEmpty(CPrioridad *cp);

/* Retorna el elemento de menor prioridad almacenado en la Cola de
 * Prioridad cp y asigna el valor de su prioridad en el parámetro
 * prioridad.
 * Pre: !IsEmpty (cp) */
int FindMin(CPrioridad *& cp, int & prioridad);

/* Elimina de la cola de prioridad el elemento de menor prioridad.
 * Pre: !IsEmpty (cp) */
void DeleteMin(CPrioridad *& cp);
```

Ejercicio 8.b)

Una implementación para la definición del TAD anterior consiste en utilizar un arreglo. Los índices del arreglo representan los niveles de prioridad y cada casilla del arreglo contiene un puntero al primer y último elemento de una lista encadenada. Además se cuenta con un registro, encargado de indicar cual es el índice de menor prioridad disponible hasta el momento, es decir un número natural encargado de almacenar la menor prioridad insertada. A continuación se presenta un diagrama con éstos conceptos:

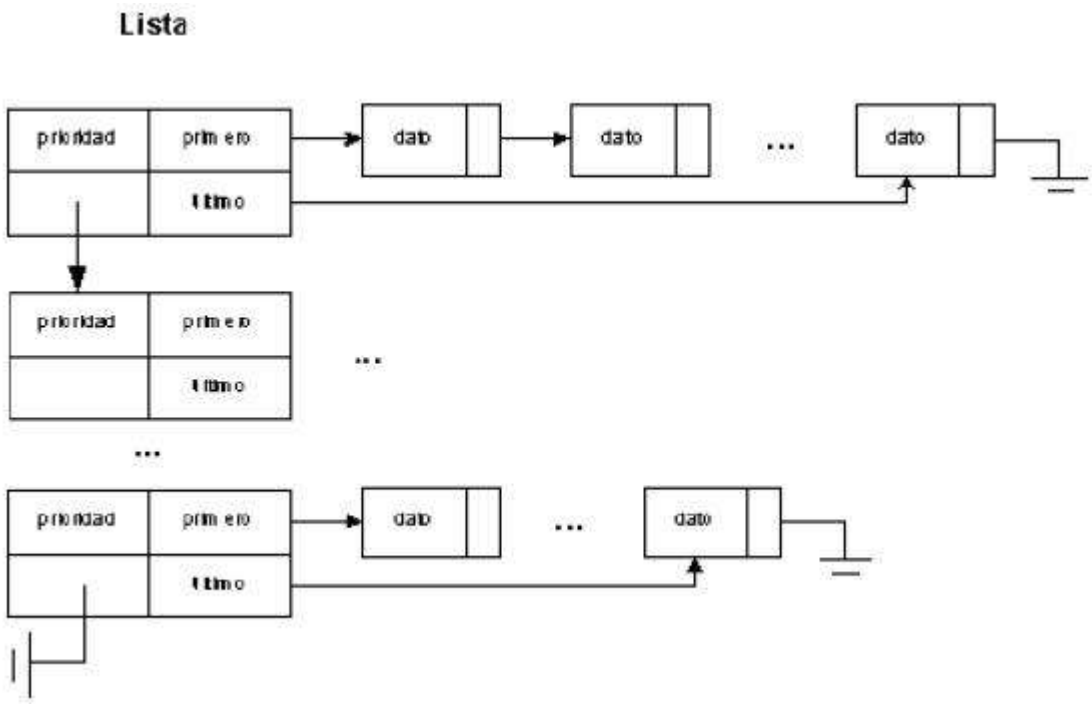


Cada vez que se agrega un elemento de prioridad n, se accede a la casilla n del arreglo y se inserta el elemento al final de la lista. En caso de que la prioridad del elemento insertado sea menor al índice que almacena la menor prioridad, entonces se actualiza el valor de dicho índice con la prioridad del elemento insertado. Para obtener el valor del elemento con la menor prioridad (operación FindMin), se accede directamente a la posición de la casilla determinada por el índice que almacena la menor prioridad de la cola. En caso de borrar un elemento de prioridad mínima (operación DeleteMin), se accede a la lista de igual forma que en la operación FindMin y se elimina la primer celda de la lista. Si luego de la eliminación la lista es vacía, es decir no quedan más elementos con la misma prioridad, entonces es necesario recorrer el arreglo y actualizar el valor del índice con el siguiente mínimo disponible en la estructura.

Intuitivamente la implementación anterior es eficiente para realizar inserciones, ya que dicha estructura permite acceder directamente a los niveles de prioridad. Por ejemplo en cada inserción se accede directamente a la casilla del arreglo y se agrega una nueva celda (incluyendo el dato) a la lista usando el puntero al último elemento. Notar que en este caso no fue necesario recorrer ninguna estructura. Acceder al elemento de prioridad mínima también es eficiente, ya que utilizando el índice de mínima prioridad se puede acceder directamente a la casilla del arreglo conteniendo el dato con menor prioridad.

Si el número de niveles de prioridad es no acotado, basta con modificar el arreglo anterior por una lista, donde el primer elemento es el de mínima prioridad. Este cambio tiene dos impactos: el primero es que se debe de almacenar en cada celda de la lista las prioridades,

el segundo, es que ahora debe de recorrerse una estructura de lista para poder insertar un nuevo elemento. A continuación se presentan estas ideas:

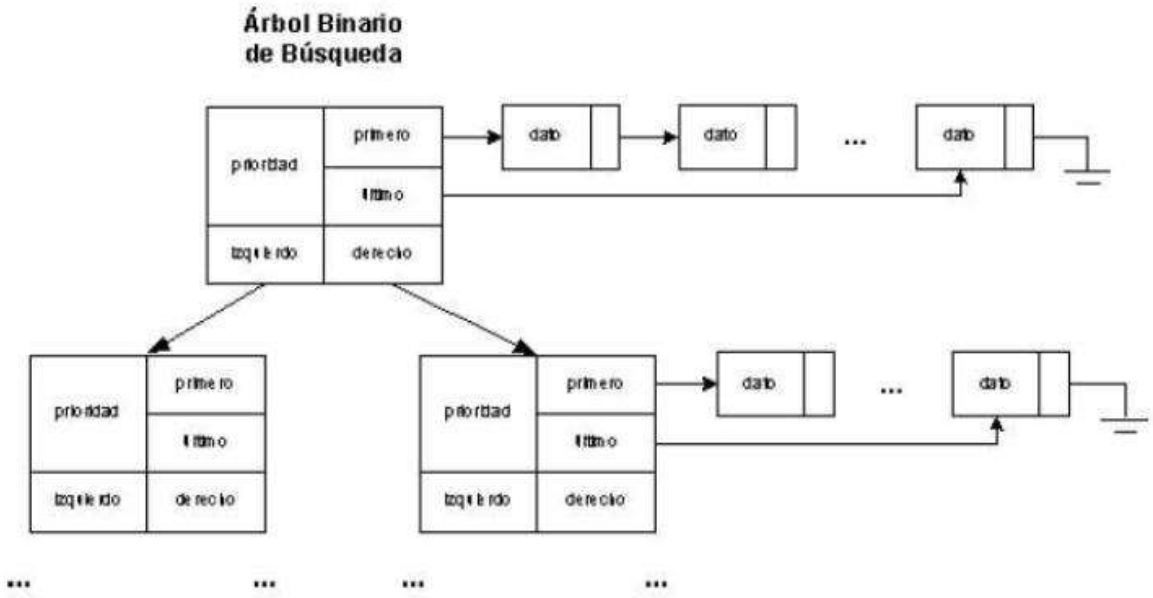


En caso de que el número de niveles de prioridad sea grande, el ejemplo anterior tiene la ventaja (sobre el modelo basado en un arreglo de prioridades) de que permite una mejor gestión de la memoria, ya que solo se crean casillas para prioridades que tienen datos asociados.

Otra implementación dinámica consiste en utilizar una lista ordenada por prioridades, donde la mínima prioridad corresponde al primer elemento. Este enfoque es recomendable solo si la cantidad de elementos es limitada ya que en caso de realizar una inserción se debe de recorrer en promedio n elementos de dicha lista.

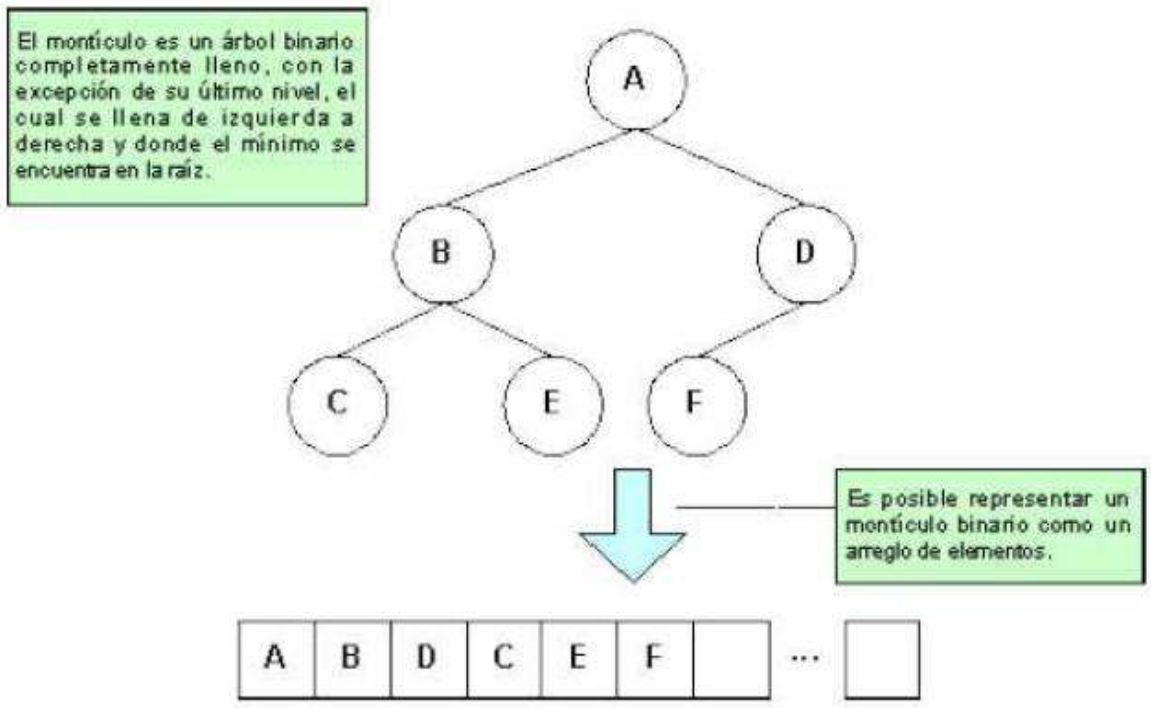
Por último se presentan dos implementaciones posibles basadas en árboles.

La primera consiste en implementar un árbol binario de búsqueda, la prioridad en dicho árbol corresponde a la clave de búsqueda. Es posible en este caso insertar todos los elementos en el árbol, manejando con cuidado que los primeros elementos insertados sean también los primeros retornados (para los que se están asociados a una misma prioridad). Otra forma más eficiente consiste en asignar a cada elemento del árbol una lista de datos con la misma prioridad. En este caso el dato prioridad no se repite por cada nodo y basta encontrar el elemento del árbol con la prioridad buscada para poder insertar y eliminar directamente. Esta implementación se presenta en el siguiente diagrama:



La última implementación basada en árboles que presentaremos utiliza la estructura de montículo (en inglés heap). Un montículo es un árbol binario completamente lleno, con la posible excepción del último nivel, el cual se llena de izquierda a derecha y donde la raíz es el elemento de valor mínimo. Un árbol con éstas características se llama árbol binario completo[1]. Dos características hacen que esta estructura pueda pensarse como una implementación del TAD cola de prioridad. Primero, que es posible insertar y eliminar elementos en orden de ejecución logarítmico, y segundo que se puede representar dicho árbol como un arreglo de elementos. A continuación se presenta un montículo y su representación vectorial:

[1] Referencia del libro *Estructuras de Datos y Algoritmos*, Mark Allen Weiss, ISBN 0-201-62571-7



A continuación se presenta una tabla resumiendo las distintas implementaciones presentadas anteriormente y la cantidad de comparaciones necesarias para una entrada de n elementos.

	Insert (promedio)	Insert (peor caso)	FindMin (promedio)	FindMin (peor caso)	DeleteMin (promedio)	DeleteMin (peor caso)
Arreglo de Listas	1	1	1	1	n	n
Listas de Listas	n	n	n	n	n	n
Lista ordenada	n	n	1	1	1	1
ABB con Listas	log(n)	n	log(n)	n	log(n)	n
Montículo Binario	log(n)	log(n)	log(n)	log(n)	log(n)	log(n)

Considerar que las tasas de crecimiento en cantidad de comparaciones para una entrada de tamaño n se ordenan de la siguiente manera (orden creciente en tiempos de ejecución): **1, log(n), n, n log(n), n*n.**