Algoritmos sobre
Búsqueda y Ordenamiento
(Análisis y Diseño)

Dra. María de Luz Gasca Soto
Departamento de Matemáticas,
Facultad de Ciencias, UNAM.

17 de marzo de 2020

Capítulo 5

Algoritmos de Ordenamiento con desempeño $O(n \log n)$

En este capítulo presentamos cuatro algoritmos de ordenamiento cuyo desempeño computacional, en el peor de los casos, es $O(n \log n)$: Merge, Quick, Heap y Tree Sort.

5.1. Merge Sort

Este método es también conocido como ordenamiento por mezcla, en esta técnica se aprecia muy claramente el uso de la estrategia divide y vencerás. Este algoritmo es considerado estable y en ocasiones requiere de memoria adicional dependiendo de la manera en que sea implementado.

Estrategia

Usando la estrategia divide y vencerás es claro observar los pasos por medio de los cuales se resuelve el problema:

Divide.- Particiona el ejemplar original en ejemplares más pequeños, de manera recursiva, hasta reducirlo a ejemplares de tamaño uno.
Vence.- Resuelve para todos los ejemplares, iniciando con los de tamaño uno, continuando con los de tamaño 2, en cada paso va incrementando (duplicando) el tamaño del ejemplar hasta llegar al tamaño original.
Mezcla.- Combina las soluciones de los problemas, aplicadas a ejemplares pequeños para resolver problemas con ejemplares más grandes, de manera recursiva, hasta obtener la solución del problema original.

La idea de esta estrategia se basa en los siguientes argumentos:

a) Toda secuencia de tamaño uno está ordenada.
b) Mezclar dos secuencias ordenadas para obtener una tercera secuencia ordenada no es complicado y no requiere muchas comparaciones.

Como ya mencionamos, en el paso divide partimos al ejemplar en otros más pequeños, pero no indicamos cómo hacer tales particiones. La estrategia consisten en ir dividiendo siempre a la mitad, de tal forma que cada vez que se realiza este paso se obtienen dos subsecuencias cuyo tamaños difieren a lo más en 1.

Para ilustrar el método consideremos el ejemplo de la Figura 5.1. Se desea ordenar la secuencia $S = \{1, 38, 27, 8, 43, 12, 3, 9, 82, 10\}$.

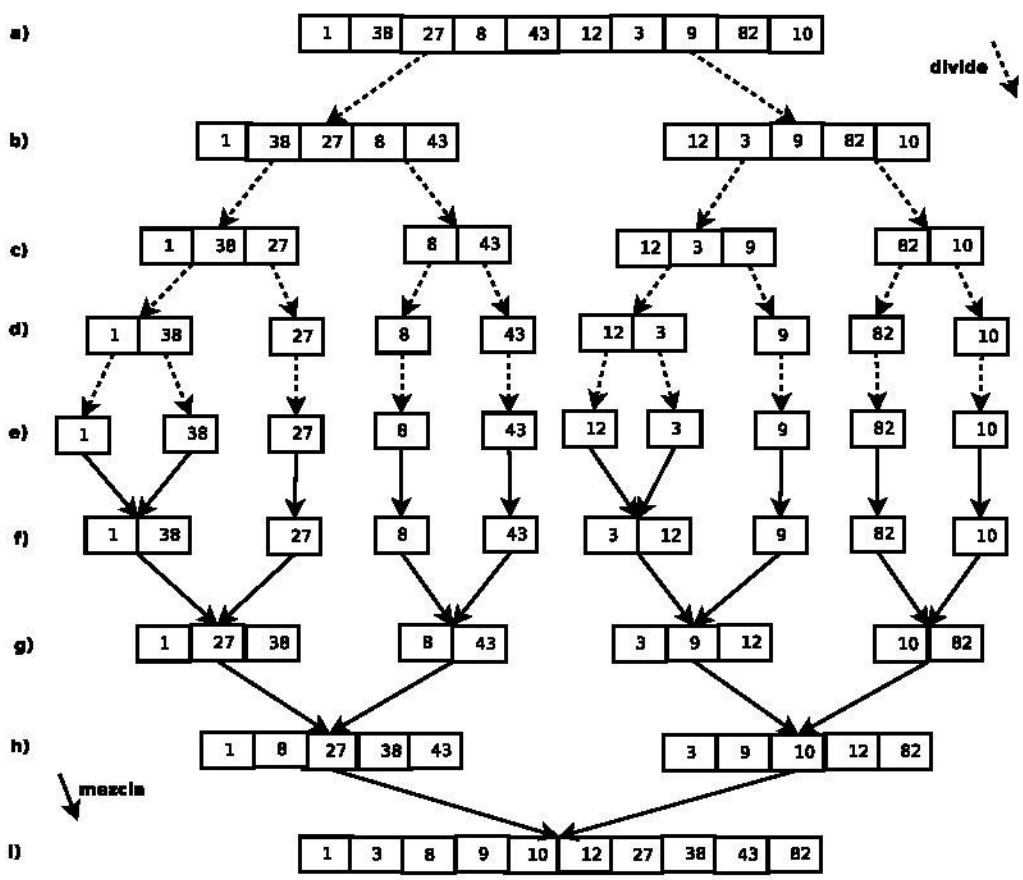

Figura 5.1: Ejemplo de Merge Sort

En el ejemplo podemos ver de forma clara como se lleva acabo el proceso divide, de manera recursiva, esto lo apreciamos de los incisos (a) al (e) en donde se obtiene subsecuencias de tamaño uno y a partir de inciso (f) se puede observar el proceso mezcla en el cual es donde se lleva acabo el ordenamiento.

Análisis de Complejidad

Para establecer el desempeño computacional es necesario primero conocer el desempeño del proceso mezcla, ya que en esta parte se efectúan las comparaciones. La cantidad de comparaciones realizadas aquí dependerá del número de elementos en las secuencias a mezclar. Revisaremos el mejor y peor caso de este proceso.

Una vez determinada la cantidad de comparaciones realizadas por mezcla se regresará al análisis de merge sort. Cabe aclarar que en el análisis se supone que la secuencia a ordenar es de tamaño n.

Sean S_A y S_B dos secuencias ordenadas con N_A y N_B elementos, respectivamente, en las cuales se aplicará el proceso mezcla.

En el mejor caso todos los elementos de una secuencia son menores al primero de la otra, supongamos, sin pérdida de generalidad, que tal secuencia es S_A, entonces el número de comparaciones realizadas es de N_A ya que sólo se compara al primer elemento de S_B con todos los elementos de S_A. La secuencia resultante, en este caso, se obtiene al agregar al final de S_A a S_B.

Para el peor caso, se tiene que $a_i < b_j, \forall a_i \in S_A$, $b_j \in S_B$, $1 \leq i \leq N_A$, $1 \leq j \leq N_B$ y $j = i$; es decir, los elementos de las secuencias S_A y S_B están intercalados, esto nos indica que para este caso se realizan $N_A + N_B - 1$ comparaciones. Como ambas secuencias están ordenadas, las comparaciones son directas. Por lo que podemos decir que el rango de comparaciones realizadas está entre N_A, el mejor caso, y $(N_A + N_B - 1)$, en el peor caso. Por lo tanto, el desempeño computacional, en el peor de los casos, para el proceso mezcla es lineal, ya que $(N_A + N_B - 1)$ es $O(n)$.

Peor caso

Para calcular el peor de los caso, tomaremos como referencia el desempeño computacional, para el peor caso del procedimiento mezcla y supondremos que en cada llamada recursiva siempre se tiene éste.

Como se está considerando a esta técnica de forma recursiva, el proceso divide se realiza hasta que las secuencias obtenidas son de tamaño uno y ya no es posible dividirlas. Ahora si a estas secuencias les aplicamos el proceso mezcla para obtener secuencias de tamaño uno es fácil ver que no hubo ninguna comparación, por ello en el peor caso de mezcla para obtener secuencias de tamaño uno se realizan cero comparaciones, lo que denotaremos como $W(1) = 0$.

Sea $w(n)$ el número de comparaciones realizadas en el peor caso para secuencias de tamaño n, entonces podemos establecer de forma recursiva el número de comparaciones realizadas por merge sort para su peor caso si consideramos lo siguiente: por un lado, sabemos que el proceso divide parte a la secuencia siempre en dos subsecuencias cuyo tamaño difiere en a lo más un elemento. Entonces, sin pérdida de generalidad y por simplicidad de cálculo, podemos decir que el tamaño de las subsecuencias al aplicar divide es de $n/2$. Por otro lado, en el peor caso, el proceso mezcla realiza $N_A + N_B - 1$ comparaciones, esto es: $(\frac{n}{2} + \frac{n}{2} - 1)$ comparaciones. Sin embargo, éstas representan sólo las últimas realizadas debido a que el proceso mezcla se lleva acabo de manera inversa al proceso divide, por ello hay que agregar las comparaciones realizadas para obtener las secuencias de tamaño $n/2$ y dado que consideramos el peor caso esta cantidad es:

$$w(n) = 2W(n/2) + \frac{n}{2} + \frac{n}{2} - 1 \; = 2W(n/2) + n - 1$$

Dado que la técnica se está considerando desde un punto de vista recursivo no basta con considerar únicamente a las comparaciones realizadas para obtener las dos últimas secuencias ordenadas a mezclar sino que también hay que tomar en cuenta todas aquellas realizadas desde las secuencias de tamaño uno. Por ello el número de comparaciones total realizadas se puede escribir de manera recursiva teniendo en cuenta a divide de la siguiente forma:

$$\begin{aligned}
w(n) &= 2W(n/2) + (n-1) \\
&= 2(2W(n/4) + (n/2) - 1) + (n-1) \\
&= 4W(n/4) + (n-2) + (n-1) \\
&= 4(2W(n/8) + (n/4) - 1) + (n-2) + (n-1) \\
&= 8W(n/8) + (n-4) + (n-2) + (n-1) \\
&= 8(2W(n/16) + (n/8) - 1) + (n-4) + (n-2) + (n-1) \\
&= 16W(n/16) + (n-8) + (n-4) + (n-2) + (n-1) \\
&= 16(2W(n/32) + (n/16) - 1) + (n-8) + (n-4) + (n-2) + (n-1) \\
&= 32W(n/32) + (n-16) + (n-8) + (n-4) + (n-2) + (n-1) \\
&\vdots
\end{aligned}$$

Se continúa de forma análoga, hasta que llegar a $W(1)$ el cual tiene valor de cero, por lo que el primer término desaparece. Para obtener la cantidad de comparaciones realizadas de manera clara conviene reescribir el resultado anterior de la siguiente forma:

$$\begin{aligned}
w(n) &= 2^1 \cdot W(n/2^1) + n \cdot (\log_2 2^1) - 2^0 \\
&= 2^2 \cdot W(n/2^2) + n \cdot (\log_2 2^2) - \textstyle\sum_{i=0}^{1} 2^i \\
&= 2^3 \cdot W(n/2^3) + n \cdot (\log_2 2^3) - \textstyle\sum_{i=0}^{2} 2^i \\
&= 2^4 \cdot W(n/2^4) + n \cdot (\log_2 2^4) - \textstyle\sum_{i=0}^{3} 2^i \\
&= 2^5 \cdot W(n/2^5) + n \cdot (\log_2 2^5) - \textstyle\sum_{i=0}^{4} 2^i \\
&\vdots \\
&= n \cdot (W(1)) + n \cdot (\log_2 n) - \textstyle\sum_{i=0}^{(\log_2 n)-1} 2^i
\end{aligned}$$

Este último término se obtiene si consideramos que el proceso divide se detiene cuando las secuencias obtenidas tienen tamaño uno, por lo que tenemos n secuencias de tamaño uno y los dos términos siguientes al observar el patrón de los casos anteriores. Dado que el primer término se hace cero, pues $W(1) = 0$ tenemos que:

$$\begin{aligned}
w(n) &= n \cdot W(1) + n \cdot (\log_2 n) - \textstyle\sum_{i=0}^{(\log n)-1} 2^i \\
&= n \cdot (0) + n \cdot (\log_2 n) - \textstyle\sum_{i=0}^{(\log n)-1} 2^i \\
&= n \cdot (\log_2 n) - \textstyle\sum_{i=0}^{(\log n)-1} 2^i
\end{aligned}$$

Sabemos que $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$, entonces tenemos que:

$$n \cdot (\log_2 n) - \sum_{i=0}^{(\log_2 n)-1} 2^i \;=\; n \cdot (\log_2 n) - 2^{\log n} + 1 \;=\; n \cdot (\log_2 n) - n + 1.$$

Lo que implica un desempeño $w(n) = O(n \log n)$ para el peor caso de Merge sort.

Mejor caso

En este escenario nuevamente tomamos como punto de partida el número de comparaciones realizadas por mezcla para el mejor caso y supondremos que en las llamadas recursivas siempre obtenemos éste. Usaremos la notación $B(n)$ para denotar el mejor caso de mezcla y $b(n)$ para el mejor caso de merge sort.

Para que mezcla genere una lista de tamaño uno no se requiere ninguna comparación, por lo cual $B(1) = 0$. Realizando un procedimiento similar al efectuado para el peor caso tenemos:

$$\begin{aligned}
b(n) &= 2B(n/2) + (n/2) \\
&= 2(2B(n/4) + (n/4)) + (n/2) \\
&= 4B(n/4) + (2n/2) \\
&= 4(2B(n/8) + (n/8)) + n \\
&= 8B(n/8) + (3n/2) \\
&= 8(2B(n/16) + (n/16)) + (3n/2) \\
&= 16B(n/16) + (4n/2) \\
&\vdots
\end{aligned}$$

Para apreciar más claramente el patrón que sigue esta secuencia reescribimos como:

$$\begin{aligned}
b(n) &= 2^1 \cdot B(n/2^1) + ([n(\log_2 2^1)]/2) \\
&= 2^2 \cdot B(n/2^2) + ([n(\log_2 2^2)]/2) \\
&= 2^3 \cdot B(n/2^3) + ([n(\log_2 2^3)]/2) \\
&= 2^4 \cdot B(n/2^4) + ([n(\log_2 2^4)]/2) \\
&\vdots \\
&= nB(1) + ([n(\log n)]/2)
\end{aligned}$$

El primer término se hace cero: $B(1) = 0$, entonces tenemos que $b(n) = ([n(\log n)]/2)$, así que en el mejor caso merge sort tiene un $O(n \log n)$.

De esta manera podemos concluir que el Algoritmo Merge sort tiene un desempeño computacional de $O(n \log n)$ tanto para el peor como el mejor caso lo que lo convierte en un método de ordenamiento muy eficiente.

Listado 17 pseudocódigo Merge Sort

```
// PreC:  secuencia contenida en un arreglo S [1,N] y  S es no vacia.
//PostC:  regresa el arreglo que contiene a la secuencia ordenada

MergeSort {
   Divide_y_Mezcla (1, N)

  procedure Divide_y_Mezcla(int izq, der)
  // PreC:  izq, der  son los indices extremos del arreglo
  //PostC:  regresa el arreglo  ordenado
   int mitad;
   if ( izq < der )  {
     mitad = (izq + der) div 2;            // calcula la mitad
     Divide_y_Mezcla ( izq, mitad );       // aplica en el subarreglo izq
     Divide_y_Mezcla ( mitad, der );       // aplica en el derecho
     Mezcla ( izq, mitad, mitad+1, der ); // mezcla los subarreglos
   } //end if
  } // end Divide...

}//end MergeSort
```

Algoritmo

El Listado 17 presenta un pseudocódigo para la versión recursiva del Merge Sort, con un enfoque de arriba hacia abajo, *top-down*. El proceso Divide_y_Mezcla es el corazón del algoritmo, se encarga de aplicar la estrategia general: divide el arreglo en dos partes iguales hasta quedarse con subarreglos de tamaño 1 y luego, durante el regreso de la recursión, mezcla los subarreglos, hasta obtener el arreglo original ordenado. Cabe mencionar, que esta versión sólo trabaja con los índices, no es necesario tener arreglos auxiliares.

El sub-algoritmo Mezcla combina los dos sub-arreglos ordenados, *marcados* sólo por los índices, en un arreglo ordenado. El Listado 18 presenta un pseudo-código para este proceso. Aquí sí es necesario tener un arreglo temporal para mover los datos.

Una versión iterativa (presentada en el Listado 19) del algoritmo de Merge Sort, tomando un enfoque de abajo hacia arriba, *bottom-up*, trabaja de la siguiente manera: En el primer paso, mezclamos elementos consecutivos formando parejas ordenadas. En el segundo paso, mezclamos parejas ordenadas de elementos generando cuartetos ordenados y así sucesivamente, hasta tener todo el arreglo ordenado. La Figura 5.2 ilustra un ejemplo de esta versión.

Otro Análisis

Considerando el código recursivo del Listado 18, determinaremos de forma más directa el desempeño computacional del Merge Sort. Sin pérdida de generalidad, suponemos que n, el tamaño de la secuencia es una potencia de 2: $n = 2^k$, para k, entero positivo.

Listado 18 Proceso Mezcla

```
// PreC:   recibe dos secuencias ordenadas:  S[Izq1,Izq2] y  S[Der1,Der2];
//         Izq1 <= Izq2 = Der1-1;  Der1 <= Der2;
//PostC:   regresa el arreglo S[Izq1..Der2] ordenado

 Mezcla (int Izq1, Izq2, Der1, Der2){
  int i, j, k;                                // indices auxiliares
  array  T[Izq1,Der2];                        // arreglo temporal

  i = Izq1;          // primer indice del sub-arreglo izquierdo
  j = Der1;          // primer indice del sub-arreglo derecho
  k = 1;             // indice auxiliar

  while (i <= Izq2) && (j <= Der2) do { // mientras no excedan los extremos
     if  ( S[i] <= S[j] ) {             //   mueve los k menores elementos
        T[k] = S[i];                    //   de S[Izq1..Der2] a T[1..k]
        i++ ;  k++ ;       }            // avanza los contadores
     else{
         T[k] = S[j];    j++ ;    k++ ;  }
  }// end while

  while  (i <= Izq2) do {               // mueve el resto de los elementos,
      T[k] = S[i];    i++ ;    k++ ; }  // si hay, de S[Izq1,Izq2] a T

  while  (i <= Der2) do {               // mueve el resto de los elementos,
      T[k] = S[j];    j++ ;    k++ ; }  // si quedan, de S[Der1,Der2] a T

  For (i= 1; i=k-1; i++);               // mueve los datos del temporal T
     S[i-1+ Izq1] = T[i];               // al original S

}// end
```

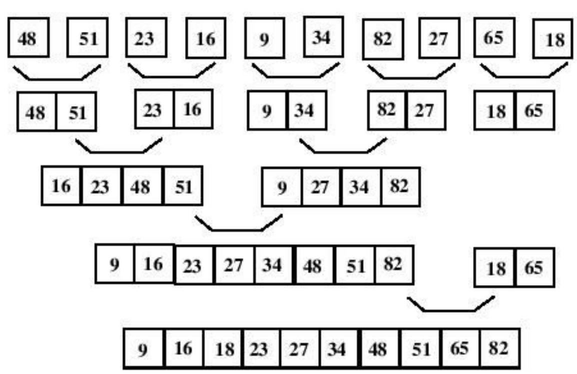

Figura 5.2: Ejemplo de Merge Sort Iterativo

Listado 19 pseudocódigo Merge Sort Iterativo

```
// PreC:  secuencia contenida en un arreglo S [1,N] y  S es no vacio.
//PostC:  el arreglo que contiene a la secuencia ordenada

MergeSort_It (array S; int n) {
  int s = 1;                           // tamanio del arreglo activo
  int Izq1, Izq2, Der1, Der2;          // indices auxiliares
   while ( s < N )  do {               // calcula la cota de las sublistas
     Izq1 = 1;
     while ( Izq1 + s <= N ) do {
       Izq2 = Izq1 + s -1;    Der1 = Izq2 + 1;
       if ( ( Der1 + s - 1) > N ) then Der2 = N;
          else    Der2 = Der1 + s - 1;
       Mezcla ( Izq1, Izq2, Der1, Der2 );    // mezcla los subarreglos
       Izq1 = Der2 + 1;                // actualiza el primer indice
       s = s * 2;                      // duplica el tamanio del arreglo
    } //end while Izq1 ...
   } //end while s ...
  } // end Divide...

}//end MergeSort
```

Sea $T_{MS}(n)$ el tiempo de ejecución del algoritmo Merge Sort aplicado a un ejemplar de tamaño n. Definimos $T_{MS}(1) = 1$. Ahora bien, $T_{MS}(n)$ es igual al tiempo requerido al aplicar el Merge Sort en dos sub-secuencias de tamaño $n/2$ más el tiempo para mezclarlas, el cual es lineal. Así, obtenemos la siguiente relación recurrente:

$T(n) = T_{MS}(1) = 1$ // tiempo sobre secuencias de tamaño 1;

$T(n) = T_{MS}(n) = 2 \cdot T_{MS}(n/2) + n.$ // tiempo sobre secuencias de tamaño n.

Para resolver esta relación de recurrencia, primero la dividiremos entre n:

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1$$

Esta ecuación es válida para cualquier n que sea potencia de 2, entonces:

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1; \quad \frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + 1; \quad \cdots$$

$$\cdots \quad \frac{T(4)}{4} = \frac{T(2)}{2} + 1 \quad \frac{T(2)}{2} = \frac{T(1)}{1} + 1.$$

Si ahora sumamos todos los términos del lado izquierdo y los igualamos con la suma de los términos del lado derecho, tenemos que $T(n/2)/(n/2)$ aparece en ambos lados, por lo cual se cancela. De hecho, virtualmente todos los términos que aparecen en ambos lados serán cancelados, pero falta por sumar los 1's y éstos son $\log n$. Después de que todo es acumulado, tenemos que:

$$\frac{T(n)}{n} = \frac{T(1)}{1} + \log n \quad \Rightarrow \quad T(n) = n \log n + n \text{ es } O(n \log n).$$

5.2. Heap Sort

Esta técnica también es conocida como ordenamiento por montículos. La estrategia que emplea consiste en aprovechar el comportamiento de una estructura de datos denominda Heaps Binarios. El Apéndice D, presenta una breve descripción de los Heaps Binarios.

Estrategia

La estrategia de esta técnica requiere de las operaciones y acciones de los heaps, a continuación describiremos dos de ellas:

Reorganización del Heap: Dado un árbol binario completo, se reorganizan los elementos del árbol para que cumplan con la relación de orden existente en el heap, a tal proceso también se le conoce como heapify o reHeap.

Eliminación del Mayor: En un heap existe una relación de orden entre sus nodos, el nodo con la mayor prioridad se encuentra en la raíz y ésto sucede para cada subárbol. Usaremos las operaciones de una cola de prioridades para eliminar el elemento de mayor prioridad, a esta función la denominaremos BorraMayor y requiere de:

1. Extraer el elemento que está en la raíz del heap.
2. Almacenar tal elemento en una lista.
3. Eliminar la raíz del heap.
4. Reorganizar el heap usando el proceso reHeap.

El bloque de pasos anteriores se lleva acabo de manera recursiva hasta que el heap queda vacío; ahora se tiene ordenada la secuencia en la lista.

Para ilustrar la estrategia se realizará un ejemplo en donde se desea ordenar la secuencia $S = \{16, 4, 9, 14, 1, 3, 10, 2, 8, 7\}$ de manera ascendente. Primero construimos un heap con los elementos de S.

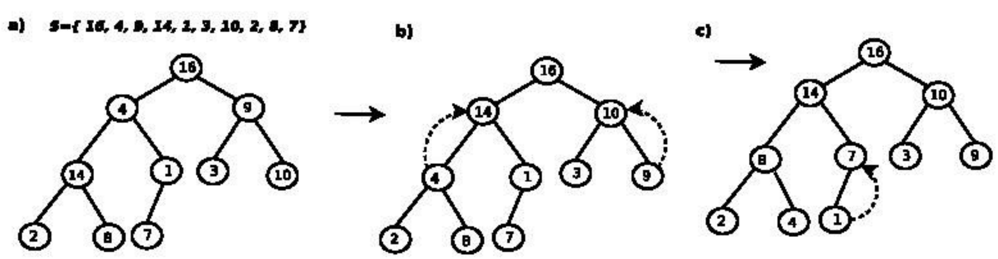

Figura 5.3: Ejemplo de construcción de un heap a partir de una secuencia

Como se observa en la Figura 5.3(a) el primer paso consiste en construir un árbol completo a la izquierda, se va insertando, cada dato, en la última posición del árbol, conforme se obtiene de la secuencia. En el inciso **(b)**, se observa el resultado de aplicar el procedimiento reHeap a los nodos intermedios del árbol, intercambiaron de posición el 14 y el 4, así como 9 y 10. En **(c)**, se tiene el resultado de aplicar reHeap a las hojas y sus padres, aquí cambio el 7 con el 1. Aquí se da por concluida la construcción de este heap maximal.

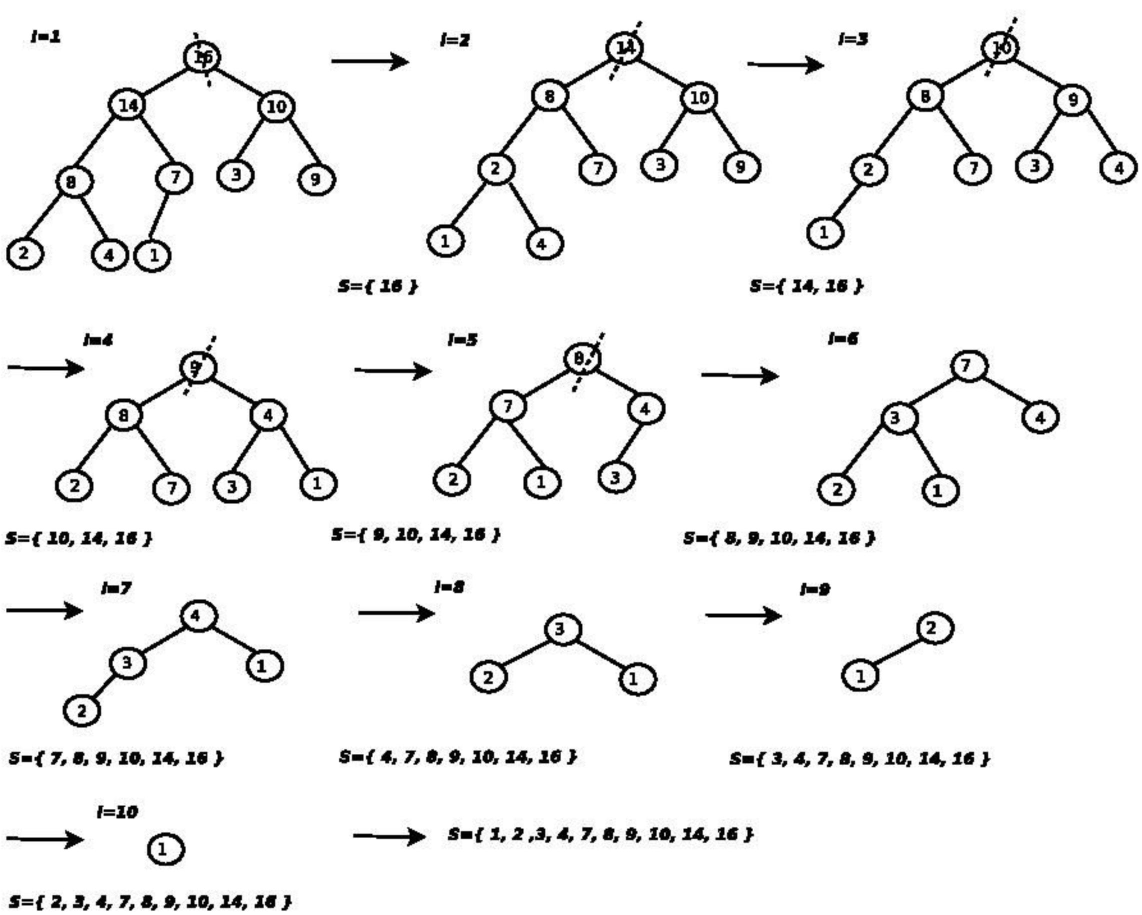

Figura 5.4: Ejemplo de implementación de una cola de prioridades

Una vez que se tiene el heap se procede a aplicar la función BorraMayor. En la Figura 5.4, se presenta en cada iteración cómo se toma la raíz del heap, la cual es almacenada en una lista, después se elimina la raíz del heap y se le aplica reHeap al árbol resultante para reorganizarlo. Se sigue con el proceso en las demás iteraciones.

Cabe mencionar, que no es necesario tener la lista, se puso aquí por motivos didácticos. En la práctica, se recomienda, ir dejando los elementos en el arreglo original.

Análisis de Complejidad

Para obtener el desempeño de este método hay que considerar la complejidad de los puntos clave de la estrategia, es decir construir el heap, reorganizarlo y eliminar al elemento de mayor prioridad.

El proceso reHeap, tiene un papel fundamental durante el desarrollo completo del método, desde convertir al árbol en un heap y mantener las propiedades de éste, así como borrar al elemento de mayor prioridad. Por lo tanto, el análisis global depende, en gran parte, del desempeño computacional del proceso reHeap.

Proceso reHeap

Sea T un árbol binario con n elementos, podemos descomponerlo T en: un nodo raíz r y sus dos subárboles: T_L y T_R. Sabemos que a un nodo se le puede considerar como un árbol con profundidad cero y también podemos considerarlo como un heap. Supongamos que los subárboles contenidos entre los niveles 1 y k son heaps. Es claro que al emplear la operación reHeap para bajar un elemento un nivel, se realizarón dos comparaciones, pues los subárboles inmediatos inferiores son heaps, entonces el número promedio de comparaciones que se espera realice reHeap en un árbol con profundidad k se puede escribir como:

$$\frac{1}{k} \cdot \sum_{i=1}^{k} 2i = \left(\frac{1}{k}\right)\left(\frac{2k\,(k+1)}{2}\right) = k+1.$$

Dado que $k = \log n$, entonces la operación reHeap tiene un desempeño de $O(\log n)$.

Construcción del Heap

Para calcular el desempeño de construir el heap emplearemos el proceso reHeap en forma ascendente, es decir empezaremos en el nivel más profundo e iremos subiendo en dirección a la raíz, de está forma podemos convertir al árbol en un heap. Si consideramos un árbol con n elementos entonces se ejecutara la misma cantidad de veces el proceso reHeap, entonces el convertir un árbol completo a la izquierda tiene una complejidad de $O(n \log n)$. Por otro lado, para calcular la complejidad de construcción del árbol completo a izquierda consideraremos que a éste lo representamos con un arreglo. Entonces el desempeño de insertar un elemento es constante y es 1, como se tienen n elementos entonces el desempeño al momento de crear el árbol a la izquierda es de $O(n)$.

Por lo tanto, la complejidad de crear un heap es de $O(n \log n + n)$, por lo que podemos concluir que construir el heap requiere tiempo $O(n \log n)$.

Análisis del BorraMayor

Acceder al elemento raíz, agregarlo a la lista y eliminarlo el árbol, son tareas que se realizan en tiempo constante. Después de estas tareas simples, es necesario reorganizar el heap y para ello ejecutamos el proceso reHeap, cuyo desempeño es $O(\log n)$. Requerimos hacer esto para cada elemento, por lo tanto, el tiempo de ejecución de la función BorraMayor es:

$$\sum_{i=1}^{n} \log n = n \log n \quad \in \quad O(n \log n)$$

Por lo tanto, la función BorraMayor tiene desempeño computacional de $O(n \log n)$.

Desempeño Computacional de Heap Sort

En los apartados anteriores se calculó el desempeño para los dos procesos principales que componen método de ordenamiento de heap sort. Si denotamos al desempeño como T_{hs} lo podemos escribir como: $T_{hs} = O(n \log n) + O(n \log n) \ \in \ O(n \log n)$

Cabe mencionar que no importa si la secuencia esta ya ordenada o no, Heap Sort siempre realiza el mismo número de comparaciones. De hecho, si la secuencia de entrada S ya está ordenana, Heap Sort la desorganiza totalmente, para ordenarla!

Por lo tanto, podemos concluir que la complejidad de heap sort es de $\Theta(n \log n)$.

Algoritmo

Dada una Lista L de n elementos, algoritmo Heap Sort se puede resumir en los siguientes cuatro pasos:

1.- Meter en un árbol binario, los elementos de la lista L.
2.- Empezando con el *último* subárbol, re-establecer heaps binarios hasta llegar a la raíz.
3.- Reorganizar el Heap para que los elementos queden ordenados.
4.- Vaciar el Heap a la lista ordenada.

El Listado 20 presenta un pseudocódigo para heapsort, supone que S es una secuencia de números enteros, no vacía y de tamaño n, contenida en un arreglo A.

El Listado 21 presenta un pseudocódigo para el proceso reHeap. Se asume que el subárbol enraizado en la posición i es un heap, excepto (tal vez) en la posición i. Para hacer del sub-árbol un heap preguntamos si $A[i]$ es menor o igual a su hijo más grande, de ser así se realiza un intercambio.

Listado 20 pseudocódigo Heap Sort

```
// PreC: S una secuencia con n elementos, contenida en un arreglo A[1,n].
// PostC: Arreglo que contiene en orden ascendente a los elementos de S.

HeapSort(array A; int n){
  heap H;                        // arbol binario completo
  int  i;                        // contador

  H.create;                      // crea el arbol binario completo
  for  (i=1; i=n; i++)           // inserta los elementos en el arbol
     H.Insert_Last ( A[i] );

  for  (i=n/2; i=1; i--)         // convierte el arbol en un heap
     H.ReHeap (i, n);

  for  (i=n; i=2; i--) {         // pone los elementos, del mayor al menor
     H.swap( 1, i );             // en las posiciones n, n-1, ..., 2, 1.
     H.reHeap( 1, i-1 ); }

  for  (i=1; i=n; i++)           // copia los elementos del heap de
     H.retrieve ( A[i], i );     // regreso al arreglo

}// end HeapSort
```

Listado 21 pseudocódigo reHeap

```
// PreC: el subarbol enraizado en i es un heap, excepto (quiza) por i
// PostC:  el subarbol enraizado en i es un heap.

reHeap(int i,j){
  int pmg;                                   // Posicion del Mas Grande

  if  ( 2*i <= j )                           // A[i] tiene al menos un hijo
    if  (2*i = j)  pmg = j;                  // ... tiene solo un hijo
     else                                    // ... tiene dos hijo
       if  (A[2*i] >= A[2*i+1])  pmg = 2*i;
        else  pmg = 2*i+1;

  if (A[i] < A[pmg])  {
    swap (i, pmg);                           // intercambia los elementos
    if  (2*pmg <= j )
       reHeap(pmg, j) }                      // aplica el reHeap

}// end reHeap
```