Ciencias computacionales

Propedeutico: Programación

INAOE

Contents

1 Métodos de ordenación

Material multimedia recomendado:

- Ordenamiento por el método de burbuja: `https://www.youtube.com/watch?v=EQMGabLO_M0`
- Ordenamiento shell: `https://www.youtube.com/watch?v=ATUKgp3R87E`
- Quick Sort: `https://www.youtube.com/watch?v=u4kUXTKFebk`
- Heap Sort: `https://www.youtube.com/watch?v=32jdzOmLsYQ`
- Vista gráfica de los métodos de ordenación: `https://www.youtube.com/watch?v=nwkNIb2fHQM`

1.1 Introducción

En esta sección se presentan distintas técnicas para ordenar un conjunto de datos. El tener datos ordenados es una ventaja para diversas aplicaciones. Por ejemplo: el algoritmo de búsqueda binaria requiere que los datos se encuentren ordenados.

Ordenar un conjunto de elementos se requiere en tareas cotidianas como ordenar los ficheros de un directorio, ya sea por nombre, fecha, tamaño, etc. Cuando queremos mostrar los registros de una base de datos es común que se muestren ordenados, esto nos permite visualizarlos y procesarlos de una manera más eficiente.

Los datos que se quieren ordenar pueden ser simples o compuestos. Dentro de los datos simples se encuentran los números enteros, números de punto flotante, los caracteres del alfabeto, etc. Por datos compuestos nos referimos a estructuras que constan de más de un campo. Por ejemplo: registros de una base de datos que almacenan registros de alumnos, donde se tiene los campos fecha de nacimiento, nombre, semestre, dirección, teléfono, etc. En este caso los elementos se ordenan de acuerdo a un campo clave. Por ejemplo, se puede seleccionar el nombre y ordenar los elementos por nombre en orden alfabético, se puede seleccionar la fecha de nacimiento, y ordenar los elementos por edad, etc.

Dentro de los algoritmos de ordenación básicos se encuentran el algoritmo de Burbuja, Shell, Merge Sort, Quick Sort, entre otros. En esta sección se presenta una breve descripción de estos algoritmos. Se utilizará un arreglo de números enteros para realizar los ejemplos de ordenamiento.

1.2 Burbuja

El algoritmo de burbuja es uno de los algoritmos más fáciles de implementar. Y aunque no es de los más eficiente es un algoritmo que sirve muy bien para ejemplificar el problema de ordenar un conjunto de elementos.

En cada paso del algoritmo de burbuja los elementos se recorren por rango creciente, esto es desde el rango 0 hasta el final de la secuencia desordenada. En cada paso se compara un elemento con su vecino, y si se observa que esos elementos consecutivos están en orden incorrecto, esto es, que el elemento anterior es mayor que el que le sigue, se intercambian los dos elementos. La secuencia se ordena al terminar n pasos.

El algoritmo de burbuja tiene las siguientes propiedades:

- En el primer recorrido el elemento mayor se coloca en la última posición. Podemos considerar que este elemento ya se encuentra ordenado y que falta por ordenar los $n-1$ elementos restantes.
- En el segundo recorrido, solo se recorre la secuencia desordenada, esto es, los primeros $n-1$ elementos. Al terminare el segundo recorrido se tendrá el siguiente elemento mayor en la penúltima posición.
- Y así sucesivamente.

El algoritmo de burbuja tiende a tener un buen rendimiento cuando los elementos del vector están casi ordenados u ordenados. En este caso el algoritmo podría continuar con su ejecución, pero ya no tendría elementos que ordenar. Por lo cual, es común utilizar una bandera que nos indique si hubo

algún intercambio (swap) de los elementos. En caso de que no se realicé intercambio alguno, podemos considerar que el vector ya se tiene ordenado y terminar la ejecución del algoritmo.

En la figura 1 se muestran los pasos que sigue el algoritmo de burbuja para ordenar un vector de 6 elementos. El algoritmo realiza comparaciones de 2 elementos contiguos. En un primer recorrido, el algoritmo recorre todo el vector, y al finalizar este primer recorrido el elemento mayor estará en la última posición del arreglo, este elemento ya se encuentra ordenado. Esto se muestra en la figura 1 con los elementos en color verde, en una primera iteración es el elemento con valor 7 el que se encuentra ordenado. En la segunda iteración el penúltimo elemento se encontrará ordenado, esto es el elemento con valor 5. El algoritmo continua con este comportamiento hasta que no queden elementos por ordenar.

En el código 1 se muestra una posible implementación del algoritmo de burbuja en c++.

Code 1: Algo

```
#include<iostream>
using namespace std;

#define N 6

void printArray(int array[]){
        for (int i = 0; i < N; i++) {
                cout << array[i] << ", ";
        }
        cout << endl;
}

void bubbleSort(int v[]){
        bool swapped = true;
        int i, j = 0;
        double tmp;

        while (swapped){
                swapped = false;
                j++;
                for (i = 0; i < N - j; i++)
                        if (v[i] > v[i + 1]) {
                                tmp = v[i];
                                v[i] = v[i + 1];
                                v[i + 1] = tmp;
                                swapped = true;
                        }
        }
}

int main(int argc, const char *argv[]){

        int array[] = { 7, 5, 3, 4, 2, 1};

        printArray(array);
        bubbleSort(array);
        printArray(array);
}
```

1.3 Shell

Este algoritmo de ordenamiento es llamado así en honor a su inventor Donald Shell. Aunque, también es conocido como método de inserción con incrementos decrecientes. En general, es una mejora del

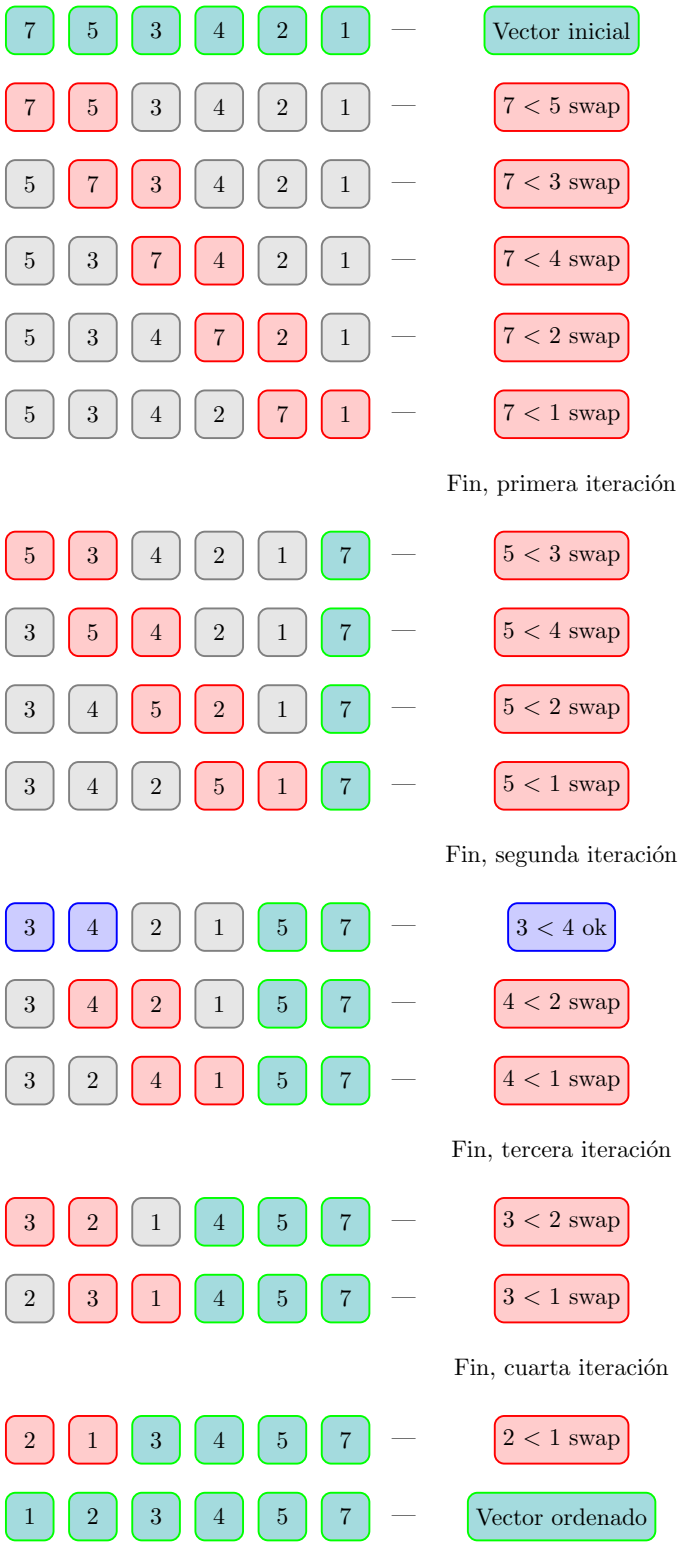

Figure 1: Ordenamiento de Burbuja

algoritmo de ordenamiento de inserción directa que se utiliza cuando se quiere ordenar una cantidad grande de elementos.

En el algoritmo de inserción, cada elemento se compara con los elementos contiguos a su izquierda, uno tras otro, hasta encontrar el lugar que le corresponde. Cuando el algoritmo inicia con el primer elemento, no hay otros elementos a su izquierda, este se encuentra en el lugar que le corresponde, esta ordenado. Posteriormente se continua con el elemento en la segunda posición. Cuando se ordena el elemento en la posición i todo los elementos con posición menores que i ya se encuentran ordenados. En la figura 2 se muestra un ejemplo del algoritmo de inserción.

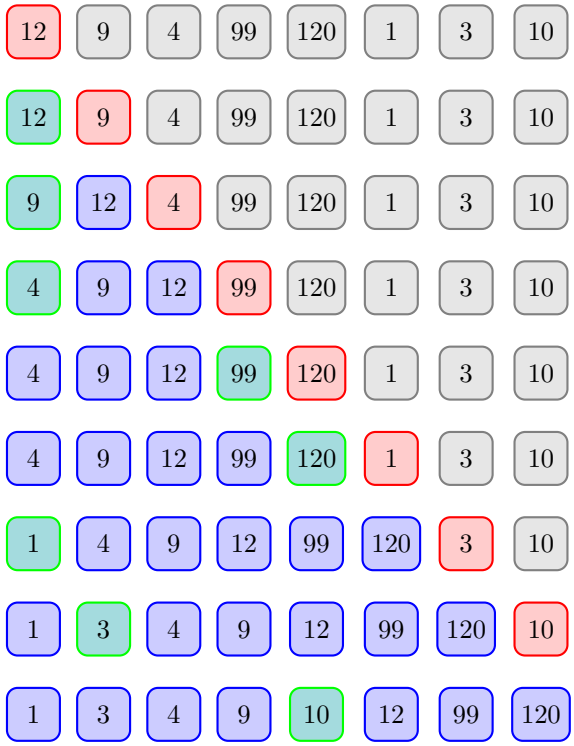

Figure 2: Algoritmo de inserción

El elemento en rojo es el valor a ordenar, y el elemento en verde es el lugar que le corresponde. Los elementos en gris son elementos que aún no han sido procesados. Los elementos en azul son elementos que ya han sido procesados y se encuentran ordenados entre ellos.

La principal desventaja del algoritmo de inserción es que si es el elemento a insertar es mas pequeño que la mayoría de elementos ya ordenados hay que realizar muchas comparaciones antes de colocarlo en el lugar que le corresponde. En el algoritmo de ordenación shell se modifican los saltos contiguos por saltos de mayor tamaño, y con eso se consigue la ordenación de forma más rápido. El método se basa en fijar el tamaño de los saltos constantes, pero más de una posición.

Supongamos un vector de elementos: 4, 12, 16, 24, 36, 3.

En el método de inserción directa, los saltos se hacen de una posición en una posición y se necesitaran cinco comparaciones para ordenar el elemento 3. En el método de Shell, si los saltos son de dos posiciones, se realizaran dos comparaciones. Y se tendría el 3 en la segunda posición. Al realizar el siguiente recorrido al vector con saltos de 1, solo se requeriría intercambiar el 3 y el 4.

El método general se basa en tomar como salto $N/2$ (siendo N el número de elementos) y luego reduciendo a la mitad en cada repetición hasta que el salto o distancia sea de 1. Aunque existen varias propuestas sobre como realizar los saltos. La última iteración que utiliza saltos de 1 es un recorrido del algoritmo de inserción original en el cual se espera que los elementos no tengas que compararse con muchos elementos ya que estarán cercanos al lugar que les corresponde.

Ejemplo:

Supongaos que queremos ordenar con el algoritmo de ordenación shell el vector x que consta de los siguientes elementos: 25, 57, ,48 ,37, 12, 92, 86, 33.

Y que se elige una secuencia de incrementos de (5, 3, 1), los elementos a ordenar para esta secuencia son:

- Con incremento 5
  - $(x[0], x[5])$
  - $(x[1], x[6])$
  - $(x[2], x[7])$
  - $(x[3])$
  - $(x[4])$
- Con incremento 3
  - $(x[0], x[3], [6])$
  - $(x[1], x[4], [7])$
  - $(x[2], x[5])$
- Con incremento 1
  - $(x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7])$

La secuencia de pasoso para ordenar el vector x se muestran el la figura 3.

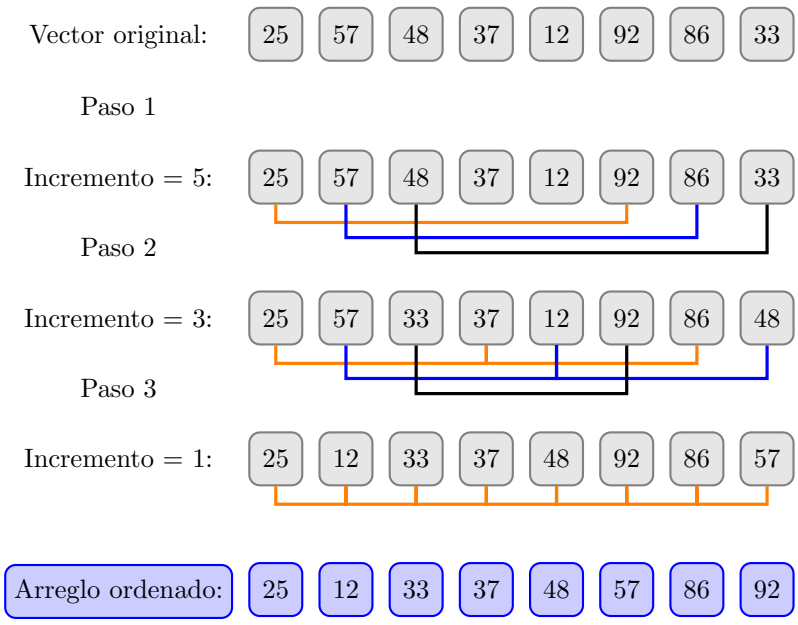

Figure 3: Ordenamiento Shell

1.4 Quick Sort

El siguiente algoritmo de ordenamiento que se describe es el llamado *Quick Sort*, también conocido como ordenamiento rápido. Este algoritmo esta basado en el paradigma de divide y vencerás. Sin embargo, utiliza esta técnica en forma particular ya que primero realiza todo el trabajo **duro** y después divide el problema y hace uso de llamadas recursivas. Algunos autores mencionan a este algoritmo como vence y divide.

El algoritmo Quick Sort ordena un vector mediante un sistema recursivo. El concepto principal es aplicar la técnica de divide y vencerás, en la que el vector a ordenar se divide en dos subvectores, se hace recursión para ordenar cada una de estos subvectores y a continuación se combinan las secuencias ordenadas con una simple concatenación. En particular, el algoritmo Quick Sort consiste en los tres pasos siguientes:

Dividir: Si el vector S tiene al menos dos elementos, seleccionar un elemento especifico de x de S, que se llama **pivote**. Es común utilizar como pivote el primer elemento de S, el último elemento de S o hacer una selección aleatoria del pivote. Se quitan todos los elementos de S se ponen en tres secuencias:

- L, que guarda los elementos de S menores que x.
- E, que guarda los elementos de S iguales a X.
- G, que guarda los elementos de S mayores que x.

Recursión: Ordenar las secuencias L y G recursivamente.

Conquistar: Regresar los elementos a S, en orden, insertando primero los elementos de L, después los de E y por último los de G.

La ejecución del algoritmo Quick Sort se puede visualizar por medio de un árbol binario de recursión, llamada árbol de ordenamiento rápido. En las figuras 4 y 5 se muestra la secuencia que seguiriá el algoritmo Quick Sort para ordenar el conjunto de elementos $84, 24, 63, 45, 17, 31, 96, 50$. La figura 4 muestra como se va dividiendo el vector oroginal de acuerdo al pivote seleccionado, en este caso el último elemento, mostrado en rojo. La figura 5 muestra la salida que se obtendria con cada llamada recursiva, resaltando en rojo el pivote seleccionado.

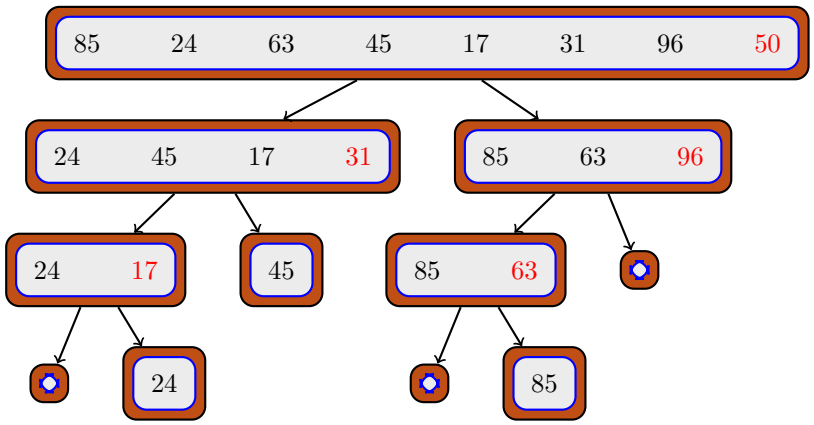

Figure 4: Árbol de orde namiento rápido para ejecutar el ordenamiento Quick Sort. Secuencia de entradas procesadas en cada nodo del árbol.

El código 2 muestra la función principal de este algoritmo. El cual hace uso de una función auxiliar para partir el vector de acuerdo al pivote. Esta función se encarga de dividir el vector original en dos subvectores, el primero con todos los elementos menores que el pivote y el segundo con todos los elementos que son mayores que el pivote. Cabe mencionar que esta función coloca el pivote en su lugar correspondiente, esto es, el pivote estará ordenado al terminar la ejecución de la función dividir. Posteriormente se ordenan cada una de estos subvectores en llamadas recursivas al método quick sort.

Code 2: Función quck sort.

```
1 void quicksort (int v[], int izda , int dcha) {
2         int pivote; // Posición del pivote
3         if (izda<dcha){
```

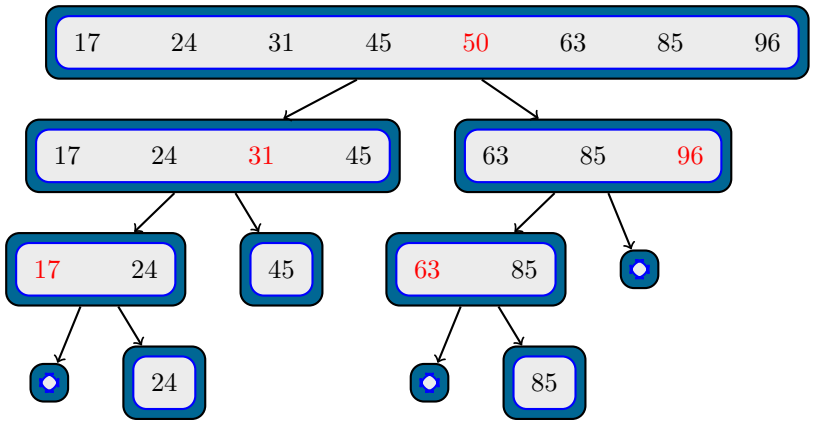

Figure 5: Árbol de orde namiento rápido para ejecutar el ordenamiento Quick Sort. Secuencia de salida generada en cada nodo del árbol.

```
                pivote = partir (v, izda , dcha);
                quicksort (v, izda, pivote -1);
                quicksort (v, pivote+1, dcha );
        }
}
```

La función partir se muestra en el código 3.

Code 3: Función partir.

```
int partir (int v[], int primero , int ultimo) {
        //valor del pivote
        int pivote = v[primero];
        //variable auxiliar
        int temporal;

        int izda = primero + 1;
        int dcha = ultimo;

        do{
                while (izda <= dcha && (v[izda]) <= pivote ) izda++;
                while (izda <= dcha && (v[dcha]) >  pivote ) dcha--;
                if (izda < dcha) {
                        temporal = v[izda];
                        v[izda]  = v[dcha];
                        v[dcha]  = temporal;
                        dcha--;
                        izda++;
                }

        }while(izda <= dcha);

        //Colocar el pivote en el sitio que le corresponde
        temporal = v[primero];
        v[primero] = v[dcha];
        v[dcha] = temporal;

        return dcha;
```

```
29 }
```

Una vez teniendo estas dos funciones lista para su ejecución basta con llamar a la función quick sort con el vector a ordenar, y los indices 0 y $N-1$, siendo N la cantidad de elementos en el vector a ordenar. Un ejemplo de la llamada a la función quicksort se muestra en la función main del código 4, en el cual se ordena un vector de diez números enteros.

Code 4: Llamado de la función quicksort.

```
1 int main(int argc, const char *argv[]) {
2         int array[] = {10,80,90,60,70,30,50,40,20,100};
3
4         printArray(array);
5         quicksort(array, 0, N-1);
6         printArray(array);
7         return 0;
8 }
```

1.5 Heap-Sort

La estructura conocida como montículo (*heap)* es un tipo especial de árbol binario con raíz, que se puede implementar eficientemente en una matriz sin ningún puntero explicito. Esta estructura tiene gran cantidad de aplicaciones, como pueden ser las colas de prioridad usadas en los sistemas operativas para la planificación de tareas, o como la presentada en esta sección conocida como ordenamiento por el método del montículo (*heapsort).*

Se dice que un árbol binario es *esencialmente completo* si todo nodo interno, con la posible excepción de un nodo especial, tiene exactamente dos hijos. El nodo especial, si existe uno, está situado en el nivel 1, y posee un hijo izquierdo pero no tiene hijo derecho. Además, o bien todas las hojas de encuentran en el nivel cero, ó bien están en los niveles cero y uno, y ninguna hoja en el nivel uno esta a la izquierda de un nodo interno del mismo nivel.

Por ejemplo, en la figura 6 muestra un árbol binario esencialmente completo con diez nodos. Los cinco nodos internos ocupan los niveles tres (la raíz), el nivel dos, y el lado izquierdo del nivel. Las cinco hojas llenan el lado derecho del nivel uno y continúan después por la izquierda del nivel cero.

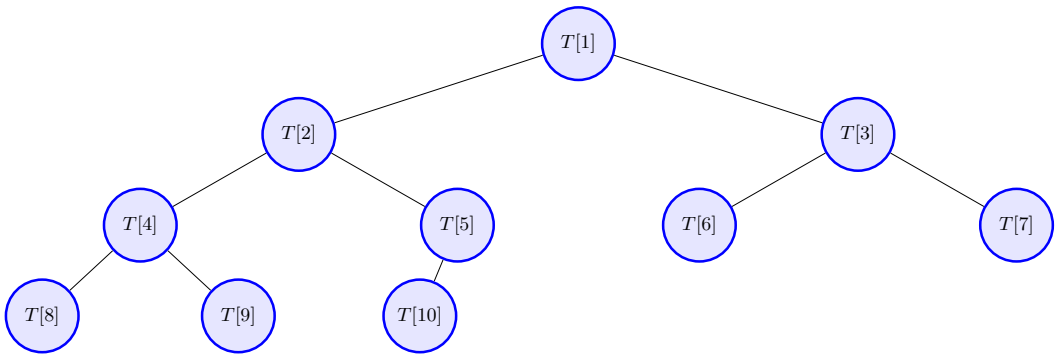

Figure 6: Árbol binario esencialmente completo.

Este tipo de estructura de árbol puede ser representada por medio de un vector T. Podemos iniciar poniendo el nodo raíz en $T[1]$, y posteriormente los hijos del nodo $T[i]$ en las posiciones $T[2i]$ y $T[2i+1]$.

Un montículo es un árbol binario esencialmente completo, cada uno de cuyos nodos incluye un elemento de información denominado *valor del nodo*, y que tiene la propiedad consistente en que el valor de cada nodo interno es mayor o igual que los valores de sus hijos. Esto se llama propiedad del montículo.

En la figura 7 se muestra un ejemplo de un montículo con diez nodos. En cual corresponde al árbol de la figura 6. Pero, ahora se muestran los valores de los nodos en vez las posiciones del arreglo. Es

fácil comprobar que este árbol cumple con la propiedad de montículo mencionada anteriormente. Este montículo es una representación del arreglo: 10, 7, 9, 4, 7, 5, 2, 2, 1, 6.

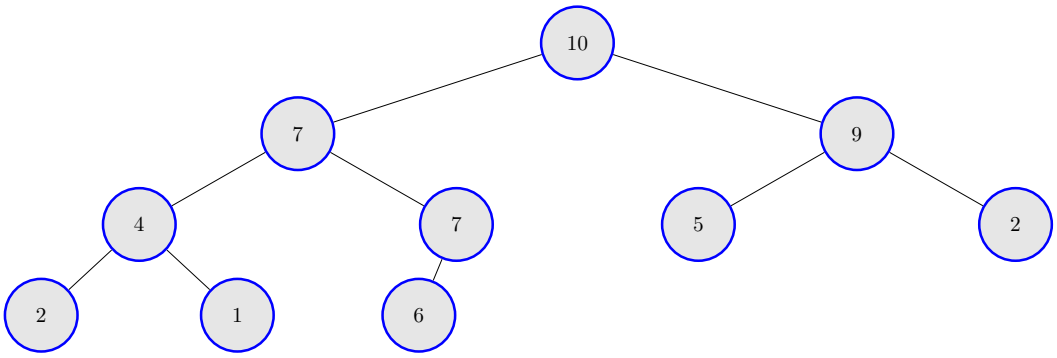

Figure 7: Árbol binario esencialmente completo.

La característica principal de un montículo es que la propiedad del montículo se puede restaurar de forma eficiente si se modifica el valor de un nodo. Si el valor de un nodo se incrementa hasta llegar a ser mayor que el de su nodo padre, es necesario intercambiar estos dos valores para que la propiedad del montículo se siga manteniendo, y continuar el mismo proceso hacia a arriba si es que es necesario hasta que se haya restaurado la propiedad del montículo. Se dice que el valor modificado ha flotado hasta llegar a su nueva posición dentro del montículo (esta operación suele denominarse *flotar)*. Por ejemplo, si se modifica el valor 1 de la figura 7 para pasar a ser 8, entonces podemos restaurar la propiedad del montículo intercambiando el 8 con su padre, que es 4, y volviendo a intercambiar con su nuevo padre, que es 7, obteniendo así el resultado que se muestra en la figura 8.

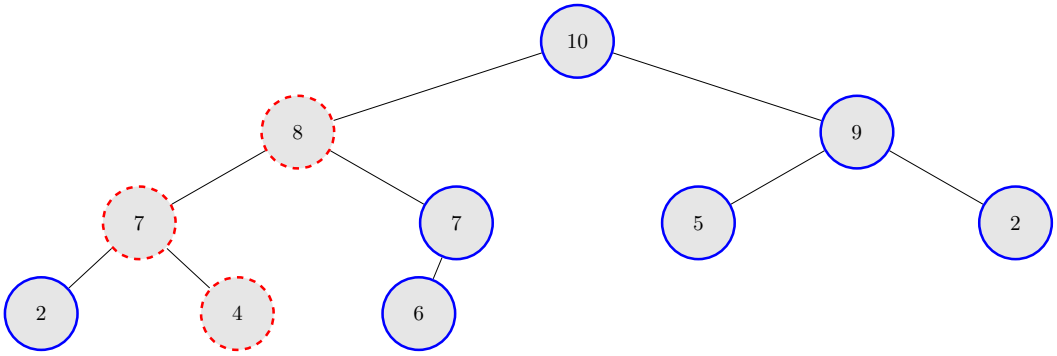

Figure 8: Operación flotar en montículo

Si por el contrario el valor del nodo decrece de tal modo que pasa a ser menor que el valor de alguno de sus hijos, entonces basta intercambiar el valor modificado con el mayor de los valores de sus nodos hijos, y continuar este proceso hacia abajo si es necesario hasta que se restaure la propiedad del montículo.

Diremos que el valor modificado se ha hundido hasta su nueva posición. Por ejemplo, si el número diez de la figura 8 pasa a ser tres, podemos restaurar la propiedad del montículo intercambiando el tres con su hijo mayor, en este caso el valor nueve, e intercambiándolo una vez más con el mayor de sus hijos nuevos, el cinco. Esto se muestra en la figura 9.

En la estructura montículo las operaciones de encontrar el mayor elemento es muy sencilla, solo verificamos el valor que se encuentra en la raíz del árbol. De igual forma las operaciones de insertar un nuevo nodo, eliminar un nodo, o modificar el valor de un nodo se pueden realizar en esta estructura a un relativamente bajo coso.

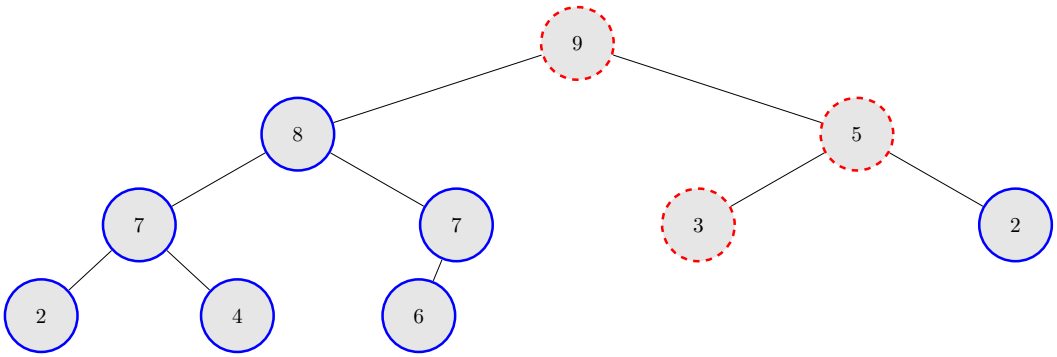

Figure 9: Operación hundir en montículo

Dado un arreglo de elementos T es posible crear un montículo con la operación flotar. Esto lo podemos observar en el algoritmo 1.

Algorithm 1 Heap

```
1: procedure CREATE HEAP(T[1 ··· n])
2:     for i ← 2 to n do
3:         FLOTAR(T[1 ··· i], i)
4:     end for
5: end procedure
```

Otra forma más eficiente de crear un montículo a partir de un arreglo T es usando la función hundir, como se muestra en el algoritmo 2.

Algorithm 2 Heap

```
1: procedure CREARMONTÍCULO(T[1 ··· n])
2:     for i ← ⌊n/2⌋ downto q do
3:         HUNDIR(T[1 ··· n], i)
4:     end for
5: end procedure
```

La estructura de montículo fue creada para servir como estructura de datos subyacente al algoritmo de ordenación mostrado en el algoritmo 3.

El algoritmo 3 require un tiempo que está en $\mathcal{O}(n \log n)$ para ordenar n elementos.

Nota: En [1] se puede encontrar código de ejemplo de los algoritmos de ordenamiento: burbuja, inserción, selección y shell.

References

[1] "Métodos de búsqueda," 2000. Available at `http://www.bufoland.cl/tc/shell.php`.

[2] G. Brassard and P. Bratley, *Fundamentos de Algoritmia (Spanish Edition)*. Prentice Hall, 1st. ed., 5 2000.

[3] M. T. Goodrich and R. Tamassia, *Estructura de Datos y Algoritmos En Java (Spanish Edition)*. Cecsa, 8 2003.

Algorithm 3 Heapsort

```
1: procedure OrdenaciónPorMontículo(T[1 ··· n])
2:     CrearMontículo(T)
3:     for i ← n downto 2 do
4:         intercambiar(T[1], T[i])
5:         hundir(T[1 ··· i − 1], 1)
6:     end for
7: end procedure
```