Tree sort

Estructuras de Datos - Facultad de Informática (UCM)

Dada una lista de elementos, el método de ordenación *Tree sort* consiste en construir un árbol binario de búsqueda con estos elementos, y luego recorrerlo en inorden para obtener los elementos en orden ascendente.

En principio, podría implementarse este algoritmo utilizando la clase `SetTree`. Dado un vector de entrada `v`, insertamos todos los elementos de `v` en un `SetTree` vacío, y luego recorremos este último en inorden, reescribiendo `v` con los valores que vayamos visitando:

```
template <typename T>
void tree_sort(vector<T> &v) {
  SetTree<T> conjunto;

  // Insertamos los elementos del vector v en el conjunto
  for (const T &elem : v) {
    conjunto.insert(elem);
  }

  // Iteramos sobre el SetTree rellenando las posiciones de v
  int pos = 0;
  for (const T &elem : conjunto) {
    v[pos] = elem;
    pos++;
  }
}
```

Sin embargo, si el vector `v` pasado como parámetro contiene elementos duplicados (por ejemplo `v = [1, 5, 2, 5, 3, 2]`), el algoritmo no funciona.

1. Modifica la función `tree_sort` para que ordene correctamente el vector `v` de entrada, incluso cuando `v` contenga elementos duplicados.
2. ¿Qué coste tiene, en el caso peor, la función `tree_sort`? Justifica la respuesta.

Solución

Guardamos los elementos del vector `v` como claves en el diccionario, cada una asociada con un entero que indica su multiplicidad, es decir, el número de veces que aparece en `v`.

Por ejemplo, si `v = [5, 1, 6, 8, 9, 9, 5, 1, 5, 5]`, se construiría el siguiente diccionario: `{ (1, 2), (5, 4), (6, 1), (8, 1), (9, 2) }`. A la hora de reconstruir el vector ordenado, tenemos que tener en cuenta la multiplicidad de cada elemento, y colocar ese elemento en el vector tantas veces como indique su multiplicidad.

```
template <typename T>
void tree_sort(vector<T> &v) {
  MapTree<T, int> m;

  // Parte 1: Construcción del diccionario
  for (const T &elem : v) {
    // El operador corchete crea una nueva entrada asociada al
    // valor por defecto del tipo. En el caso de los enteros,
    // ese valor es 0, por lo que el incremento establecería
    // el valor inicial a 1, en el caso de que la clave no existiese
    // en el diccionario.
    m[elem]++;
    // Esto sería equivalente a:
    // if (m.contains(elem)) {
    //    m.insert({elem, 1});
    // } else {
    //    m[elem]++;
    // }
  }

  // Parte 2: Reconstrucción del vector
  int cont = 0; // cont es la siguiente posición del vector que será
                // reescrita.
  // Recorremos las entradas del diccionario
  for (auto [k, v] : m) {
    // Para cada una de ellas, colocamos la clave k
    // tantas veces como indique su multiplicidad.
    for (int i = 0; i < v; i++) {
      v[cont] = k;
      cont++;
    }
  }
}
```

El acceso o inserción a un `MapTree` tiene coste $O(m)$ en el caso peor, si los árboles no se equilibran automáticamente, o coste $O(\log m)$ si se reequilibran automáticamente, donde m es el tamaño del diccionario. Por tanto, el bucle correspondiente a la parte 1 del algoritmo tiene coste $O(n^2)$ en el caso no equilibrado, o coste $O(n \log n)$ en el caso equilibrado.

El bucle correspondiente a la parte 2 tiene coste lineal con respecto al tamaño del vector de entrada, en cualquier caso.

Por tanto, el coste total en el caso no equilibrado es $O(n^2)$ y en el caso equilibrado $O(n \log n)$.