Recorridos con iteradores

Estructuras de Datos
Facultad de Informática - UCM

1. Escribe una función `eliminar_pares` que, dada una lista de números enteros, elimine aquellos que sean pares. Indica su coste en tiempo.

   ```
   void eliminar_pares(list<int> &list);
   ```

2. Implementa una función que, dada una lista de números enteros, indique si está ordenada de manera ascendente. Indica su coste en tiempo.

   ```
   template <typename T>
   bool ordenada_ascendente(const list<T> &list);
   ```

3. Un palíndromo es una palabra o frase que se lee igual en un sentido o en otro. Escribe una función `es_palindromo` que determine si la cadena pasada como parámetro es un palíndromo. Utiliza iteradores para recorrer la cadena en ambos sentidos.

   ```
   bool es_palindromo(const string &cadena);
   ```

Solución

1. ```
   // El iterador 'it' recorre la lista desde el principio
   // hasta el final
   void eliminar_pares(list<int> &lista) {
     auto it = lista.begin();
     while (it != lista.end()) {
       if (*it % 2 == 0) {
         // El metodo 'erase' invalida el iterador pasado como parametro,
         // y devuelve un iterador al elemento que esta justo despues del
         // eliminado. Por tanto, reescribo 'it' con ese valor.
         it = lista.erase(it);
       } else {
         ++it;
       }
     }
   }
   ```

   El coste de la función `eliminar_pares` es lineal con respecto al número de elementos de la lista pasada como parámetro. Existe un bucle que se ejecuta tantas veces como la longitud de la lista, y dentro del cuerpo solamente se realizan operaciones de coste constante.

2. ```
   // Tenemos dos iteradores, 'it_ant' e 'it_sig', que van a estar
   // siempre en posiciones consecutivas, el primero antes que el segundo.
   // Ambos iteradores avanzan simultaneamente, y ha de cumplirse que
   // el valor apuntado por 'it_ant' sea menor o igual que el valor apuntado
   // por 'it_sig'.
   bool ordenada_ascendente(const list<int> &lista) {
     auto it_ant = lista.begin();
     if (it_ant == lista.end()) return true;
     auto it_sig = ++lista.begin();

     while (it_sig != lista.end() && *it_ant <= *it_sig) {
       it_ant++;
       it_sig++;
     }

     return it_sig == lista.end();
   }
   ```

   Si la lista de entrada tiene N elementos, la función tiene coste $\mathcal{O}(N)$. En efecto, tenemos un bucle que se ejecuta $N - 1$ veces y en cada iteración solamente se realizan operaciones de acceso a iteradores y de incremento de iteradores, todas ellas de coste constante.

3. ```
   // Tenemos dos iteradores: 'it1', que comienza al principio de la lista
   // e 'it2', que comienza al final de la lista. El primer iterador se desplaza
   // hacia la derecha, y el segundo hacia la izquierda. Comprobamos que los
   // caracteres apuntados por ambos iteradores coinciden.
   //
   // En realidad, como 'cadena.end()' no apunta a ningun caracter, siempre
   // vamos a comprobar '*it1' con '*(--it2)', es decir, el valor apuntado por
   ```
   ```

```
// la posicion que esta a la izquierda de aquella apuntada por it2.
bool es_palindromo(const string &cadena) {
  auto it1 = cadena.begin(), it2 = cadena.end();
  bool palindromo = true;
  while (it1 != it2 && it1 + 1 != it2 && palindromo) {
    it2--;
    if (*it1 != *it2) {
      palindromo = false;
    }
    it1++;
  }
  return palindromo;
}
```

En este caso, si N es el número de caracteres de la `cadena` de entrada, el coste de esta función es $\mathcal{O}(N)$. en este caso, el bucle hace aproximadamente $N/2$ iteraciones.