`ConjuntoChar` con elementos ordenados

Estructuras de Datos
Facultad de Informática - UCM

Retomemos la definición de `ConjuntoChar` que utiliza un array con los caracteres contenidos en el conjunto:

```
class ConjuntoChar {
public:
  ConjuntoChar();

  bool pertenece(char letra) const;
  void anyadir(char letra);

private:
  int num_elems;
  char elems[MAX_CHARS];
};
```

Si `c` es una instancia de `ConjuntoChar`, suponemos la siguiente función de abstracción e invariante de representación:

$$f(\texttt{c}) = \{\texttt{c.elems}[i] \mid 0 \leq i < \texttt{c.num\_elems}\}$$

$$\begin{aligned} I(\texttt{c}) = \quad & 0 \leq \texttt{c.num\_elems} \leq \texttt{MAX\_CHARS} \\ & \wedge\, \forall i : 0 \leq i < \texttt{c.num\_elems} \Rightarrow \texttt{c.elems}[i] \in \{A..Z\} \\ & \wedge\, \forall i, j : 0 \leq i < j < \texttt{c.num\_elems} \Rightarrow \texttt{c.elems}[i] < \texttt{c.elems}[j] \end{aligned}$$

1. Explica, con tus propias palabras, el significado del invariante de representación.
2. Implementa el método `anyadir` para que inserte el elemento dado en el array `elems` de manera ordenada, preservando así el invariante. Si el elemento ya existía en el array, no hace nada.
3. Implementa el método `pertenece` para que realice una búsqueda eficiente del elemento pasado como parámetro.
4. Indica el coste en tiempo de ambas operaciones.

Solución

El invariante de representación dice que:

- $\forall i : 0 \leq i < \texttt{c.num\_elems} \Rightarrow \texttt{c.elems}[i] \in \{A..Z\}$

  Todos los elementos del segmento `c.elems[0..num_elems)` son caracteres contenidos entre la letra A y la letra Z.

- $\forall i, j : 0 \leq i < j < \texttt{c.num\_elems} \Rightarrow \texttt{c.elems}[i] < \texttt{c.elems}[j]$

  El segmento `c.elems[0..num_elems)` está ordenado de manera ascendente y no contiene elementos duplicados.

```
class ConjuntoChar {
public:
  // Crea un ConjuntoChar vacío
  ConjuntoChar();

  // Determina si una letra pertenece al ConjuntoChar
  bool pertenece(char letra) const;
  // Añade una letra al ConjuntoChar
  void anyadir(char letra);

private:
  int num_elems;
  char elems[MAX_CHARS];

  // Métodos privados:
  // Realiza una búsqueda de la letra en el vector ‘elems‘
  int buscar_caracter(char letra) const;
  // Método recursivo auxiliar para la búsqueda binaria
  int buscar_caracter_rec(char letra, int c, int f) const;

};

// Crea un ConjuntoChar vacío (sin elementos)
ConjuntoChar::ConjuntoChar() {
  num_elems = 0;
}

// O(log N), donde N es el número de elementos del conjunto
bool ConjuntoChar::pertenece(char l) const {
  // Se hace una búsqueda binaria de ‘l‘ en el vector.
```

```
  int pos = buscar_caracter(l); // O(log num_elems)
  // Ojo: si la letra no está nos devuelve la posición en la que
  // debería insertarse. Esa posición puede ser num_elems, es decir,
  // fuera del array. Por tanto, antes de comprobar si elems[pos] == l,
  // hemos de asegurarnos de que pos < num_elems.
  return pos < num_elems && elems[pos] == l;
}

// O(N), donde N es el número de elementos del conjunto
void ConjuntoChar::anyadir(char l) {
  // Buscamos la letra utilizando búsqueda binaria
  int pos = buscar_caracter(l); // O(log num_elems)

  // Si no hemos encontrado la letra, pos nos dice dónde
  // debería insertarse
  if (pos == num_elems || elems[pos] != l) {
    // Suponemos que el vector no está lleno
    assert (num_elems != MAX_CHARS);

    // Desplazamos todos los elementos del segmento elems[pos..num_elems) una
    // posición hacia la derecha.

    // Coste: el bucle hace (num_elems - pos) iteraciones. Cada una de ellas
    // tiene coste O(1). Por tanto, O(num_elems - pos). Sin embargo, en el
    // caso peor, pos == 0, por lo que es O(num_elems).
    for (int j = num_elems; j > pos; j--) {
      elems[j] = elems[j - 1];
    }
    // Colocamos el nuevo elemento en la posición ‘pos‘.
    elems[pos] = l;
    num_elems++;
  }

  // Coste: O(log num_elems + num_elems) = O(num_elems)
}

/*
  Método privado auxiliar.

  Busca la ‘letra‘ pasada como parámetro en el vector ‘elems‘. Devuelve la
  posición del vector en la que se encuentra. Si no la encuentra, devuelve la
  posición en la que debería insertarse.
```

```
  Precondición:
    - El vector elems no tiene elementos duplicados

  Postcondición:
    - elems[result] = letra, o bien
    - Todos los elementos del segmento v[0..result) son menores que letra y
      todos los elementos del segmento v[result..num_elems) son mayores que
    letra
*/
int ConjuntoChar::buscar_caracter(char letra) const {
  // Hace uso de una búsqueda binaria implementada recursivamente.
  return buscar_caracter_rec(letra, 0, num_elems);
}

/*
  Método privado auxiliar.

  Busca la ‘letra‘ pasada como parámetro en el segmento elems[c..f). Devuelve
  la posición del vector en la que se encuentra. Si no la encuentra, devuelve
  la posición en la que debería insertarse.

  Precondición:
    - 0 <= c <= f <= num_elems
    - Todos los elementos del segmento elems[0..c) son estrictamente menores
    que ‘letra‘
    - Todos los elementos del segmento elems[f..num_elems) son estrictamente
    mayores que ‘letra‘
    - El vector elems no tiene elementos duplicados

  Postcondición:
    - elems[result] == letra, o bien
    - Todos los elementos del segmento v[0..result) son menores que letra y
      todos los elementos del segmento v[result..num_elems) son mayores que
    letra
*/
int ConjuntoChar::buscar_caracter_rec(char letra, int c, int f) const {
  if (c == f) {
    // La precondición dice que elems[0..c) contiene elementos estrictamente
    // menores que letra y que elems[f..num_elems) contiene elementos
    // estrictamente mayores. Si c == f, entonces todos los elementos del
```

```
    // vector son distintos de ‘letra‘. Además, si insertásemos la ‘letra‘,
    // debería ser en la posición ‘c‘.
    return c;
  } else {
    // Si el segmento no es vacío, calculamos el punto medio m
    int m = (c + f) / 2;
    if (elems[m] == letra) {
      // Si encontramos la letra en el punto medio, la devolvemos directamente
   .
      // Se cumple la postcondición elems[result] == letra
      return m;
    } else if (elems[m] < letra) {
      // Si en el punto medio hay un elemento estrictamente menor que ‘letra‘,
      // entonces todo el segmento ‘elems[c..m)‘ también es menor que ‘letra‘,
      // por lo que podemos descartarlo. Nos centramos en el ‘elems[m+1..f)‘.
      return buscar_caracter_rec(letra, m + 1, f);
    } else {
      // Si en el punto medio hay un elemento estrictamente mayor que ‘letra‘,
      // entonces todo el segmento ‘elems[m..f)‘ puede descartarse. Nos queda
      // el segmento elems[c..m) para buscar la letra.
      return buscar_caracter_rec(letra, c, m);
    }
  }
}
```