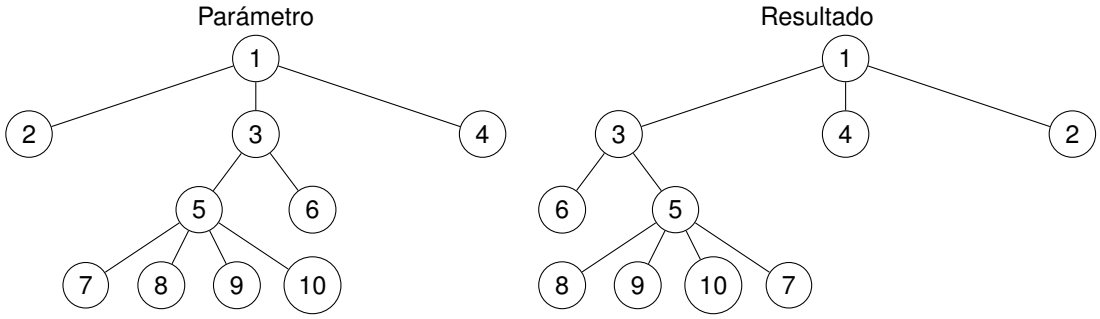
Examen de Programación 2

8 de diciembre de 2021

Problema 1 (50 puntos)

Se debe implementar una función que, dado un árbol general de elementos de un tipo genérico, `G`, para cada nodo que tiene hijos rota sus subárboles dejando al final el que era primero, y manteniendo el orden relativo de los restantes.

En el siguiente ejemplo los nodos que tienen hijos son los que están etiquetados con 1, 3, y 5. Los que eran sus primeros hijos, 2, 5 y 7 respectivamente, quedan como últimos hijos. El 3 y el 4 mantienen el mismo orden que tenían como hijos de 1. Lo mismo sucede con 8, 9 y 10 como hijos de 5.



La representación de los árboles es mediante árboles binarios con semántica primer hijo—siguiente hermano. La especificación de los tipos de datos y funciones es la siguiente:

```
typedef int G;

struct repT {
  G dato;
  struct repT * ph;
  struct repT * sh;
};

typedef struct repT * Tree;

/*
  Rota los subárboles de cada nodo de T y devuelve el resultado de la transformación.
 */
Tree rotar(Tree T);
```

(a) Implemente una función **iterativa** que aplica la rotación descrita solo al nodo pasado por parámetro (no a sus subárboles).

```
/*
  Devuelve el árbol resultado de mover el primer subárbol de 'T'
  dejándolo como su último subárbol.
  Precondición: T != NULL.
 */
Tree moverPrimero(repT * T);
```

(b) Implemente la función `rotar`.

Solución:

(a)
```
Tree moverPrimero(repT * T) {
  if ((T->ph != NULL) && (T->ph->sh != NULL)) {
    repT * aMover = T->ph;
    T->ph = T->ph->sh;
    aMover->sh = NULL;
    repT * cursor = T->ph;
    while (cursor->sh != NULL) {
```

```
            cursor = cursor->sh;
          }
          cursor->sh = aMover;
        }
        return T;
      }
```

(b)
```
Tree rotar(Tree T) {
  if (T != NULL) {
    T = moverPrimero(T);
    T->sh = rotar(T->sh);
    T->ph = rotar(T->ph);
  }
  return T;
}
```

Problema 2 (50 puntos)

Se debe determinar si una lista de enteros tiene elementos repetidos. No se puede acceder a la representación de la lista, sino que se dispone de una versión del TAD `Lista` que incluye las operaciones `longitud` y `pertenece` además de las operaciones habituales. La siguiente es la especificación del TAD y de la función a implementar:

```
/* Devuelve una lista sin elementos, */
Lista crear();

/* Devuelve 'true' si y solo si 'l' no tiene elementos. */
bool esVacia(Lista l);

/*
  Devuelve la lista resultado de insertar 'a' al inicio de 'l'.
*/
Lista insertar(int a, Lista l);

/*
  Devuelve el primer elemento de 'l'.
  Precondición: ! esVacia(l).
*/
int primero(Lista l);

/*
  Devuelve 'l' sin su primer elemento .
  Precondición: ! esVacia(l).
*/
Lista siguiente(Lista l);

/* Devuelve 'true' si y solo si 'a' es un elemento de 'l'. */
bool pertenece (int a, Lista l);

/* Devuelve la cantidad de elementos de 'l'.  */
int longitud (Lista l);

/* Devuelve 'true' si y solo si en 'l' no hay elementos repetidos. */
bool sinRepetidos(Lista l);
```

Se puede asumir que el tiempo de ejecución de `crear`, `esVacia`, `primero`, `insertar` y `siguiente` es $O(1)$, y que el de `pertenece` y `longitud` es $O(n)$, donde n es la longitud de la lista parámetro.

(a) Asuma que los enteros pertenecen al dominio $\{1, \ldots, M\}$ donde M es una constante conocida. Implemente la función `sinRepetidos` con un tiempo de ejecución $O(n)$ en peor caso.

(b) No hay restricción al valor que pueden tomar los enteros y cualquiera de ellos tiene la misma probabilidad de pertenecer a la lista. Implemente la función `sinRepetidos` con un tiempo de ejecución $O(n)$ promedio.

Solución:

(a)
```
bool sinRepetidos(Lista l) {
  // Si M < n el resultado debe ser false
  // Por lo tanto se podría verificar eso antes del siguiente ciclo
  // De todas formas el costo de esa consulta sería O(n), el costo de pertenece.
  bool dominio [M + 1];
  for (int i = 1; i <= M; i++)
    dominio[i] = false;

  while (! esVacia(l)) && ! dominio[primero(l)] ) {
    dominio[primero(l)] = true;
    l = siguiente(l);
  }
  return esVacia(l);
}
```

(b)
```
bool sinRepetidos(Lista l) {
  int n = longitud(l);
  Lista * tabla = new Lista[n];
  for (int i = 0; i < n; i++)
    tabla[i] = rear();

  while ((! esVacia(l)) && (! pertenece(primero(l), tabla[primero(l) %n]))) {
    insertar(primero(l), tabla[primero(l) %n]);
    l = siguiente(l);
  }
  return esVacia(l);
}
```