Examen de Programación 2

14 de diciembre de 2018

Problema 1 (33 puntos)

Se desea saber si una pila y una cola tienen exactamente los mismos elementos y fueron insertados en ellas exactamente en el mismo orden. Para esto implemente la función `mismosElementos` usando las operaciones de los TADs `Pila` y `Cola`. Al terminar, `p` y `c` deben quedar vacías. Los elementos de `Pila` y `Cola` son de un tipo genérico que admite los usuales operadores de comparación.

```
bool mismosElementos(Pila &p, Cola &c);
```

No se deben hacer comparaciones innecesarias. No se pueden usar funciones auxiliares.

Solución:

```
bool mismosElementos(Pila &p, Cola &c) {
  Pila cInvertida = crearPila();
  while (!esVaciaCola(c)) {
    apilar(frente(c), cInvertida);
    desencolar(c);
  }

  while (!esVaciaPila(p) && !esVaciaPila(cInvertida) &&
         (tope(p) == tope(cInvertida))) {
    desapilar(p);
    desapilar(cInvertida);
  }
  bool res = esVaciaPila(p) && esVaciaPila(cInvertida);

  while (!esVaciaPila(p))
    desapilar(p);

  liberarPila(cInvertida);
  return res;
}
```

Problema 2 (33 puntos)

Implemente una función que devuelve un árbol binario de búsqueda con los elementos que en otro árbol binario de búsqueda pertenecen al rango determinado por dos parámetros, `min` y `max` (`min <= max`), ambos incluidos. Si en el árbol parámetro no hay elementos entre `min` y `max` la función devuelve el árbol vacío.

(a) Escriba la solución en lenguaje funcional: entre : int $\times$ int $\times$ ABB $\rightarrow$ ABB.

(b) Escriba la implementación en C*: `ABB entre(int min, int max, ABB a);`

El árbol resultado no comparte memoria con el parámetro.

No se deben visitar nodos innecesarios. No se pueden usar funciones ni estructuras auxiliares.

Asuma la siguiente representación de árbol binario de búsqueda y que el árbol vacío es `NULL`:

```
struct rep_ABB { int r; rep_ABB * I, * D; };
typedef rep_ABB * ABB;
```

Solución:

(a)
```
entre(min, max, ()) = ()
entre(min, max, (I,r,D)) =
    entre(min, max, D), Si r < min
    entre(min, max, I), Si r > max
    (entre(min, max, I), r, entre(min, max, D)),
           Si min <= r <= max
```

(b)
```
ABB entre(int min, int max, ABB a) {
  ABB res;
  if (a == NULL)
    res = NULL;
  else {
    if (a->r < min)
      res = entre(min, max, a->D);
    else if (a->r > nayor)
      res = entre(min, max, a->I);
    else {
      res = new rep_ABB;
      res->r = a->dato;
      res->D = entre(min, max, a->D);
      res->I = entre(min, max, a->I);
    }
  }
  return res;
}
```

Problema 3 (34 puntos)

Sea el TAD `Mapping` acotado que asocia enteros con enteros. Las claves pertenecen a un rango cuyos límites, `min` y `max` (`min <= max`), ambos incluidos, se establecen al crear el mapping.

(a) Complete junto a las siguientes operaciones la especificación, con pre y post condiciones, del TAD `Mapping`.

```
/* Devuelve un mapping sin asociaciones.
   Las claves pueden estar entre 'min' y 'max' incluidos.
   Precondición: min <= max. */
Mapping crearMap(int min, int max);

/* Devuelve 'true' si y solo si 'clave' está en el rango
   definido por crearMap. */
bool enRango(int clave, Mapping map);
```

(b) Defina la representacion del TAD e implemente las operaciones constructoras. La representación debe permitir que todas las operaciones, excepto la de creación tengan tiempo de ejecución $O(1)$ peor caso. El espacio usado debe ser $O(\text{max} - \text{min})$.

Solución:

(a)
```
/* Devuelve un mapping sin asociaciones.
   Las claves pueden estar entre 'min' y 'max' incluidos.
   Precondición: min <= max. */
Mapping crearMap(int min, int max);

/* Devuelve 'true' si y solo si 'clave' está en el rango
   definido por crearMap. */
bool enRango(int clave, Mapping map);

/* Devuleve 'true' si y solo si en 'map' existe una asociación
   para 'clave'. */
bool existeAsociacion(int clave, Mapping map);

/* Asocia 'clave' con 'valor' en 'map'.
   Precondiciones:
      enRango(clave, map),
      !existeAsociacion(clave, map). */
void asociar(int clave, int valor, Mapping &map);

/* Desasocia 'clave' en 'map'.
   Precondición: existeAsociacion(clave, map). */
void desasociar(int clave, Mapping &map);

/* Devuelve el valor asociado a 'clave' en 'map'.
   Precondición: existeAsociacion(clave, map). */
int valor(int clave, Mapping map);
```

(b)
```
typedef struct rep_mapping * Mapping;

struct rep_mapping {
  int *valores;
  bool *existen;
  int min, max;
```

```
};

Mapping crearMap(int min, int max) {
  Mapping map = new rep_mapping;
  map->valores = new int[max - min + 1];
  map->existen = new bool[max - min + 1];
  for (int i = 0; i < max - min + 1; i++)
    map->existen[i] = false;
  map->min = min;
  map->max = max;
  return map;
}

void asociar(int clave, int valor, Mapping &map) {
  map->valores[clave - map->min] = valor;
  map->existen[clave - map->min] = true;
}
```

Se completa la implementación:

```
bool enRango(int clave, Mapping map) {
  return ((map->min <= clave) && (map->max >= clave));
}


bool existeAsociacion(int clave, Mapping map) {
  return map->existen[clave - map->min];
}

int valor(int clave, Mapping map) {
  return map->valores[clave - map->min];
}

void desasociar(int clave, Mapping &map) {
  map->existen[clave - map->min] = false;
}
```