Examen de Programación 2

14 de diciembre de 2017

Problema 1 (30 puntos)

Considere la siguiente representación de lista de enteros implementada con nodos simplemente enlazados:

```
struct rep_lista { int dato;
                   rep_lista * sig; };
typedef rep_lista * Lista;
```

Se asume que el valor de la lista vacía es `NULL`.

Implemente un procedimiento **removerRep** que toma como parámetro una lista ordenada de enteros y remueve los elementos repetidos:

```
/* Remueve los elementos repetidos de l.
  Precondición: l está ordenada. */
void removerRep(Lista &l);
```

Ejemplos:

Entrada	Salida
L = [1, 1, 2, 3, 3, 3, 4, 4, 5]	L = [1, 2, 3, 4, 5]
L = []	L = []
L = [1]	L = [1]

Solución:

Con un cursor se recorre la lista verificando los elementos consecutivos. Si el elemento siguiente es igual al cursor se elimina el siguiente. En otro caso se avanza el cursor hacia el siguiente.

El cursor debe ser una variable local porque 'l' es pasada por referencia. Hay que notar que el valor de 'l' no va a cambiar aunque sea vacia ya que el primer elemento no se elimina aunque el segundo sea igual a él. Por lo tanto no es necesario tratar de manera especial el primer elemento.

```
void removerRep(Lista &l){
  Lista actual = l;
  while (actual != NULL){
    Lista siguiente = actual->sig;
    if ((siguiente != NULL) && (actual->dato == siguiente->dato)){
      actual->sig = siguiente->sig;
      delete(siguiente);
    } else {
      actual = actual->sig;
    }
  }
}
```

Problema 2 (30 puntos)

Considere el TAD `ColaPrio` que es una cola de prioridad acotada, cuyos elementos son enteros. La prioridad de cada elemento está determinada por su valor y el elemento prioritario es el de menor valor. Se pide:

(a) Especifique completamente el TAD `ColaPrio` (con pre y post condiciones).

(b) ¿Con qué estructura se puede implementar el TAD para que las operaciones de inserción y selectoras tengan tiempo de ejecución $O(\log n)$ en peor caso, siendo n la cantidad de elementos en la estructura?

(c) ¿Cuáles son las dos propiedades características de la estructura de la parte anterior?

Solución:

(a)
```
/* Devuelve una cola de prioridad vacia que puede mantener
   hasta 'capacidad' elementos.  */
ColaPrio crearCP(int capacidad);

/* Inserta 'dato' en 'cp'.
   Precondición: ! esLlenaCp(cp) */
void insertarCP(int dato, ColaPrio &cp);

/* Devuelve true si y sólo si en 'cp' no hay elementos. */
bool esVaciaCP(ColaPrio cp);

/* Devuelve true si y sólo si en 'cp' hay 'capacidad'
   elementos. */
bool esLlenaCP(ColaPrio cp);

/* Devuelve el menor elemento de 'cp'.
   Precondición: ! esVaciaCP(cp) */
int min(ColaPrio cp);

/* Remueve el menor elemento de 'cp'.
   Precondición: ! esVaciaCP(cp) */
void removerMin(ColaPrio &cp);
```

(b) La cola de prioridad se implementa con un *heap* binario, que es un árbol parcialmente ordenado. La operacion para obtener el mínimo se ejecuta en tiempo $O(1)$ en peor caso. Las operaciones para remover el mínimo y para insertar se ejecutan en tiempo $O(\log n)$ en peor caso mediante filtrado descendente y filtrado ascendente respectivamente.

(c) El heap binario tiene propiedad de estructura y propiedad de orden.

Propiedad de estructura Es un árbol completo. Esto significa que cada nivel, excepto tal vez el último está completo. Si el último nivel no está completo los nodos que falan son los de más a la derecha. Por lo tanto el árbol puede representarse con un arreglo en el cual los punteros están implicitos. A la raíz le corresponde la posición 1. A cualquier otro nodo, si le corresponde la posición i a su padre le corresponde la posición $i/2$. Si el nodo al que le corresponde la posición i tiene hijos estos están en las posiciones $2i$ y $2i+1$.

Propiedad de orden El valor de cada nodo es menor que el de cada uno de sus descendientes propios.

Con esta implementación la operación que obtiene el prioritario se realiza en tiempo $O(1)$ en peor caso.

De manera alternativa se puede usar un *AVL*, aunque se debe tener en cuenta que en este caso la operación de obtener el prioritario es $O(\log n)$ en peor caso. La propiedad de orden es que en cada

nodo el valor del elemento es mayor que los de su subárbol izquiedo y menor que los de su subarbol derecho. La propiedad de estructura es que está balanceado según la altura: para cada nodo la alturas de sus subarboles izquierdo y derecho difieren a lo sumo en 1. Todas las operaciones tienen tiempo $O(\log n)$ en peor caso. En las dos operaciones selectoras se debe acceder al menor elemento que es el nodo que está más a la izquierda.

Problema 3 (40 puntos)

Utilizando el TAD `Pila` implemente una función que determine si una expresión tiene delimitadores correctamente balanceados:

```
/* Devuelve true si y sólo si E tiene
   delimitadores correctamente balanceados. */
bool esBalanceada(char E[], int n);
```

La expresión se mantiene en un arreglo y consiste en n símbolos que pueden ser operadores, números, nombres de variable, los delimitadores de apertura '(' y '[' y sus correspondientes delimitadores de cierre ')' y ']'. Decimos que una expresión tiene delimitadores correctamente balanceados si tiene alguna de estas tres formas:

1. Secuencia de símbolos, posiblemente vacía, sin delimitadores de apertura ni de cierre
2. e1 '(' e2 ')' e3
3. e4 '[' e5 ']' e6

donde e1, . . . , e6 son expresiones, posiblemente vacías, con delimitadores correctamente balanceados.

Ejemplos:

Expresiones correctas	Expresiones incorrectas
a * 3	(a * 3))
2 [5 + 3 (x + y) - [ ]]	2 [5 + 3 (x + y])

Solución:

La solución es la que está presentada en el material teórico *Tema 7 - TAD Lista, Pila y Cola Aplicación: Balanceo de paréntesis*, páginas 24 a 26.

```
bool esBalanceada(char E[], int n) {
  Pila p = crearPila();
  int i = 0;
  bool esB = true;
  while (esB && (i < n)) {
    if ((E[i] == '(') || (E[i] == '[')) {
      apilar(E[i], p);
    } else if (E[i] == ')') {
      if (esVacia(p) || tope(p) != '(')
        esB = false;
      else
        desapilar(p);
    } else if (E[i] == ']') {
      if (esVacia(p) || tope(p) != '[')
        esB = false;
      else
        desapilar(p);
    }
    i++;
  }
  return esB && esVacia(p);
}
```