Instituto de Computación. Facultad de Ingeniería. Universidad de la República

Solución Segundo Parcial Programación 2

Julio de 2021

Problema 1 (15 puntos)

Considera la siguiente especificación del TAD ***PilaChar*** para una pila no acotada de elementos de tipo ***char***:

```
PilaChar crearPila ();
// Devuelve la pila vacía de caracteres

void apilar (char c, PilaChar &p);
// Inserta c en la cima de p

bool esVaciaPila (PilaChar p);
// Retorna true si y solo si la pila p es vacía

void desapilar (PilaChar & p);
// Remueve la cima de p.
// Precondicion: !esVaciaPila(p)

char tope (PilaChar p);
// Devuelve la cima de p.
// Precondicion: !esVaciaPila(p)
```

Llamamos *palíndrome con separador '-'* a una secuencia de caracteres de la forma *s*-*t*, donde '-' es un *char* y, *s* y *t* son secuencias de caracteres sin el caracter '-', que cumplen que: *t* es el reverso de *s*. En particular, si *s* y *t* son secuencias vacías, *s*-*t* es un *palíndrome con separador '-'*.
Se quiere determinar si una secuencia de caracteres dada, almacenada en una pila de tipo ***PilaChar***, es un *palíndrome con separador '-'*. La longitud de las secuencias de caracteres no está acotada.
Concretamente, se pide implementar la función:

bool esPalindromeConSeparador (PilaChar p)

Considere los siguientes ejemplos:

- Palíndromes: abac-caba, -, a-a
- No palíndromes: abac!caba, bac-caba, abac-cab, -abc, abc-, -a-a-, *la secuencia vacía*

Solo se pueden usar variables auxiliares de tipo *bool*, *char* o *PilaChar*. No se puede usar funciones o procedimientos auxiliares, salvo las operaciones del TAD *PilaChar*. Se puede modificar la pila parámetro de entrada *p*. La función *esPalindromeConSeparador* debe ser eficiente, en el sentido de que la ejecución debe terminar si al obtener un elemento de *p* se podría saber que no se cumple la condición pedida.

Problema 2 (25 puntos)

Considere un TAD ***Conjunto*** de reales que tiene, entre otras, las siguientes operaciones:

- ***void eliminar (Conjunto & c, float x)***, que elimina el elemento *x* del conjunto *c*, si está. En caso contrario, la operación no tendrá efecto.
- ***Conjunto copia (Conjunto c)***, que retorna una copia del conjunto *c* sin compartir memoria con éste y sin modificar *c*.

Considere una implementación del TAD ***Conjunto*** usando ***hashing* abierto**, con la siguiente representación:

```
struct nodoHash {
    float dato;
    nodoHash * sig;
};
```

```
struct representacionConjunto {
    nodoHash ** tabla;   // tabla de hash
    int cota;            // tamaño de la tabla de hash
    int cantidad;        // cantidad actual de elementos en el conjunto
};
typedef representacionConjunto * Conjunto;
```

Se pide implementar las operaciones ***eliminar*** y ***copia,*** considerando la representación previa (donde no se permiten nodos con datos repetidos) y asumiendo la existencia de una función de hash: ***unsigned int h (float x)***.

Instituto de Computación. Facultad de Ingeniería. Universidad de la República

Solución Segundo Parcial Programación 2

Julio de 2021

Problema 3 (5 puntos parte a - 15 puntos parte b)

a) Describa brevemente las propiedades de ***Estructura*** y de ***Orden*** que debe cumplir un árbol binario de enteros, que admite elementos repetidos, para ser un ***heap*** (***montículo binario***).

b) Implemente una función **iterativa *esHeap*** que, dado un arreglo de enteros con tope que representa a un árbol binario (que puede tener elementos repetidos), devuelva true si y solo si el arreglo cumple la ***propiedad de Orden***. En particular, si el arreglo no tiene elementos el resultado deberá ser *true*. Considere que en un arreglo con tope no vacío, los elementos están entre las posiciones 1 y *tope* del arreglo. Asimismo, asuma que el *tope* del arreglo es menor que el tamaño (*tam*) del arreglo. El tiempo de ejecución en el peor caso de la función debe ser *O(n)*, siendo *n* la cantidad de elementos en el arreglo.

bool esHeap (ArrayTope a)

```
struct cabezalArrayTope {
    int * arreglo;  // arreglo
    int tope;       // tope del arreglo
    int tam;        // tamaño del arreglo
};
typedef cabezalArrayTope * ArrayTope;
```

Considere los siguientes **ejemplos**:

Arreglo con tope de tamaño (tam) 20	**esHeap(arreglo)**
arreglo = [1, 2, 3, 4, 5, 6, 7, 8], tope = 8	true
arreglo = [2, 1, 3, 4, 5, 6, 7, 8], tope = 8	false
arreglo = [], tope = 0	true
arreglo = [2], tope = 1	true
arreglo = [1, 2, 3, 8, 7, 6, 5], tope = 7	true
arreglo = [1, 2, 3, 2, 2, 3, 3, 2, 2, 4, 4, 3, 3], tope = 13	true
arreglo = [1, 6, 4, 9, 9, 3, 5], tope = 7	false

Instituto de Computación. Facultad de Ingeniería. Universidad de la República

Solución Segundo Parcial Programación 2

Julio de 2021

Solución Problema 1

Se va a determinar si el reverso de s es igual a t. Para obtener el reverso de s se crea una pila auxiliar en la que se apila el tope de p y se lo desapila hasta que p quede vacía o hasta encontrar el separador.

```
bool esPalindromeConSeparador (PilaChar p)
{
        bool res = false;
        if (! esVaciaPila(p))
        {
                char separador = '-';
                PilaChar sReverso = crearPila ();
                while (!esVaciaPila (p) && (tope(p) != separador))
                {
                        apilar(tope (p), sReverso);
                        desapilar (p);
                }
                // Si p quedó vacía entonces no había separador,
                // por lo que el resultado debe ser false.
                // Si p no quedó vacía entonces el separador está en el tope de p.
                if (!esVaciaPila (p))
                {
                        desapilar (p);
                        while (!esVaciaPila(p) && !esVaciaPila(sReverso) && (tope (p) == tope
(sReverso)))
                        {
                                desapilar (p);
                                desapilar (sReverso);
                        }
                        res = esVaciaPila(p) && esVaciaPila(sReverso);
                }
        }
        return res;
```

En la anterior solución se llama dos veces a tope(p) en cada iteración del primer while. Aunque se puede suponer que en este caso esas invocaciones no son costosas, en general se debe evitar hacerlo. Presentamos una segunda solución que evita esa doble llamada. Hay otras posibles soluciones, algunas de las cuales son específicas del lenguaje C.

```
bool esPalindromeConSeparador (PilaChar p)
{
        bool res = false;
        if (! esVaciaPila (p))
        {
                char separador = '-';
                PilaChar sReverso = crearPila ();
```

Solución Segundo Parcial Programación 2

Julio de 2021

```
                char c = tope (p);
                desapilar(p);
                while (!esVaciaPila (p) && (c != separador))
                {
                        apilar(c, sReverso);
                        c = tope (p);
                        desapilar (p);
                }
                // Si en p no había separador el último elemento todavía no se apiló en
sReverso.
                if (c != separador)
                        apilar(c, sReverso);
                while (!esVaciaPila(p) && !esVaciaPila(sReverso) && (tope (p) == tope
(sReverso)))
                {
                        desapilar (p);
                        desapilar (sReverso);
                }
                res = esVaciaPila(p) && esVaciaPila(sReverso);
        }
        return res;
}
```

Instituto de Computación. Facultad de Ingeniería. Universidad de la República

Solución Segundo Parcial Programación 2

Julio de 2021

Solución Problema 2

Solución con eliminar recursivo:

```
//Precondición: Existe a lo sumo una única ocurrencia de x en la lista índice
//Postcondición: En caso de encontrar al elemento x, lo elimina de la lista índice y devuelve true.
bool eliminar_recursivo (nodoHash * & indice, float x)

{
        bool eliminado = false;
        if (indice != NULL)
        {
                if (indice->dato == x)
                {
                        nodoHash * eliminar = indice;
                        indice = indice->sig;
                        delete eliminar;
                        eliminado = true;
                }
                else
                        eliminado = eliminar_recursivo (indice→sig, x);
        }
        return eliminado;
}

void eliminar (Conjunto & c, float x)
{
        unsigned int posicion   = h (x) % c->cota;
        eliminado = eliminar_recursivo (c->tabla [posicion], x);
        if (eliminado) c->cantidad--;
}
```

Solución con eliminar iterativo:

```
//Precondición: Existe a lo sumo una única ocurrencia de x en la lista índice
//Postcondición: En caso de encontrar al elemento x, lo elimina de la lista índice y devuelve true.
bool eliminar_iterativo (nodoHash * & indice, float x)
{
        bool eliminado = false;
        if (indice != NULL)
        {
                //En este caso estoy parado sobre el puntero del bucket hacia la lista
                if (indice->dato == x)
                {
                        nodoHash * eliminar = indice;
                        indice = indice->sig;
                        delete eliminar;
                        eliminado = true;
```

```
                }
                else
                        //En este caso voy a recorrer la lista con un puntero auxiliar
                        nodoHash * indice_aux = indice;
                        while (indice_aux->sig != NULL && !eliminado)
                        {
                                if (indice_aux->sig->dato == x)
                                {
                                        nodoHash * eliminar = indice_aux->sig;
                                        indice_aux->sig = eliminar->sig;
                                        delete eliminar;
                                        eliminado = true;
                                }
                                else
                                         indice_aux = indice_aux->sig;
                        }
        }
        return eliminado;
}

void eliminar (Conjunto & c, float x)
{
        unsigned int posicion   = h (x) % c->cota;
        eliminado = eliminar_iterativo (c->tabla [posicion], x);
        if (eliminado) c->cantidad--;
}
```

Solución copia:

```
//Postcondición: devuelve en copia_indice una copia sin compartir memoria de la lista apuntada
por índice.
void copiar_recursivo (nodoHash * indice, nodoHash * & copia_indice)
{
        if (indice == NULL) copia_indice = NULL;
        else
        {
                copia_indice            = new nodoHash;
                copia_indice->dato      = indice->dato
                copiar_recursivo (indice->sig, copia_indice->sig);
        }
}

Conjunto copia (Conjunto c)
{
        //Copio el cabezal
        Conjunto c_copia        = new representacionConjunto;
        c_copia->cota           = c->cota;
```

Solución Segundo Parcial Programación 2

Julio de 2021

```
        c_copia->cantidad       = c->cantidad;
        c_copia->tabla          = new nodoHash*[c_copia->cota];

        //recorro la estructura y copio cada una de las entradas
        for (int i = 0; i < c_copia->cota; i++)
                copiar_recursivo (c->tabla [i], c_copia->tabla [i])

        return c_copia;
}
```

Instituto de Computación. Facultad de Ingeniería. Universidad de la República

Solución Segundo Parcial Programación 2

Julio de 2021

Solución Problema 3

Parte a)

Propiedad de estructura : Un heap es un árbol binario completamente lleno, con la posible excepción del nivel más bajo, el cual se llena de izquierda a derecha. Notar que se puede representar como un arreglo donde para cualquier elemento en la posición i del arreglo, el hijo izquierdo está en la posición 2*i, el hijo derecho en la posición siguiente: 2*i+1 y el padre está en la posición $\lfloor i / 2 \rfloor$.

Propiedad de orden: Para todo nodo X, la clave en el padre de X es: menor (o menor o igual) que la clave en X, con la excepción obvia de la raíz (donde esta el mínimo elemento).

Parte b)

```
bool esHeap (ArrayTope a)
{
        bool condicion  = true;
        int indice      = 1;
        while (indice*2 <= a->tope && condicion)
        {
                if  (a->arreglo [indice] > a->arreglo [indice*2] || (indice*2+1
      <= a->tope && a->arreglo [indice] > a->arreglo [indice*2+1]))
                        condicion = false;
                indice++;
        }
        return condicion;
}
```