Primer Parcial de Programación 2

3 de mayo de 2019

En recuadros con este formato aparecerán aclaraciones que cumplen una función explicativa pero que no eran requeridos como parte de la solución.

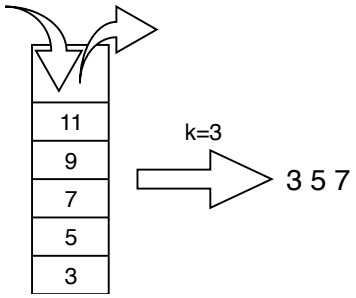
Problema 1 (8 puntos)

Dada la especificación de la Figura 1a del TAD Pila de Enteros, escriba un procedimiento iterativo `imprimirKAntiguos` que reciba una pila `p` y un entero positivo `k`, e imprima los `k` números más antiguos de la pila en el orden en que fueron ingresados. Si la pila tiene menos de `k` elementos, se deberán imprimir todos. Al finalizar el procedimiento la pila `p` debe quedar en la misma condición en que estaba originalmente.

```
void imprimirKAntiguos(int k, Pila p)
```

```
Pila crearPila ();
/* Devuelve la pila vacía. */
Pila apilar (int s, Pila p);
/* Inserta s en la cima de p. */
int cima (Pila p);
/* Devuelve la cima de p.
Precondicion: ! esVaciaPila(p). */
Pila desapilar (Pila p);
/* Remueve la cima de p.
Precondicion: ! esVaciaPila(p). */
bool esVaciaPila (Pila p);
/* Devuelve 'true' si p es vacia,
 'false' en otro caso. */
void destruirPila (Pila &p);
/* Libera la memoria asociada a p. */
```

(a) Especificación del TAD Pila de Enteros.



(b) Ejemplo de `imprimirKAntiguos` con k=3.

Figura 1: Especificación del TAD Pila de Enteros y ejemplo `imprimirKAntiguos`.

Solución:

```
//k entero positivo, mayor a cero.
void  imprimirKAntiguos (int k, Pila p) {
  Pila aux = crearPila();
  while (! esVaciaPila(p)) {
    aux = apilar(cima(p), aux);
    p = desapilar(p);
  }
  while (!esVaciaPila(aux) && (k > 0)) {
    printf("%d ", cima(aux));
    k--;
    p = apilar(cima(aux), p);
    aux = desapilar(aux);
  }
  while (!esVaciaPila(aux)) {
    p = apilar(cima(aux), p);
    aux = desapilar(aux);
  }
  destruirPila(aux);
}
```

Problema 2 (14 puntos)

Considere la siguiente definición de nodos de listas dinámicas de enteros:

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

(a) Defina un procedimiento **recursivo** `insertarOrdenado` que dada una lista de enteros L ordenada de menor a mayor, sin elementos repetidos, y dado un entero x, inserte a x en L manteniendo el orden. Si x estaba en L, el procedimiento no tendrá efecto.

```
void insertarOrdenado(Lista & L, int x)
```

(b) Defina una función iterativa `ordenar` que dada una lista de enteros L, que puede contener elementos repetidos, retorne una nueva lista, que no comparta memoria con L, que esté ordenada de menor a mayor y contenga a los elementos de L. Cada elemento de L debe aparecer una sola vez en la lista resultado; esto es, la lista ordenada resultante no debe tener elementos repetidos. Utilice el procedimiento `insertarOrdenado` para definir la función `ordenar` .

(c) Calcule el orden de tiempo de ejecución para el peor caso de `insertarOrdenado` y `ordenar`. Justifique acotando superiormente el tiempo de ejecución para el peor caso del procedimiento y la función respectivamente.

Solución:

(a)
```
void insertarOrdenado(Lista & L, int x) {
  if ((L == NULL) || (x < L->dato)) {
    Lista sig = L;
    L = new nodoLista;
    L->dato = x;
    L->sig = sig;
  } else if (x > L->dato)
    insertarOrdenado(L->sig, x);
}
```

(b)
```
Lista ordenar(Lista L) {
  Lista res = NULL;
  // se puede iterar con L porque se pasa por valor
  while (L != NULL) {
    insertarOrdenado(res, L->dato);
    L = L->sig;
  }
  return res;
}
```

(c) **insertarOrdenado** Sea n el largo de la lista. El peor caso se da cuando x es mayor que todos los elementos de L.

$$T(n) \leq \begin{cases} c_1 & \text{si } n = 0, \\ T(n-1) + c_2 & \text{en otro caso.} \end{cases}$$

Mediante sustitución hacia atrás:

$$\begin{aligned} T(n) &\leq T(n-1) + c_2 \\ &\leq T(n-2) + 2\ c_2 \\ &\quad \cdots \\ &\leq T(n-i) + i\ c_2 \\ &\quad \cdots \\ &\leq T(n-n) + n\ c_2 \\ &\leq c_1 + n\ c_2. \end{aligned}$$

Se deduce que $T(n) = O(n)$.

ordenar Sea n el largo de la lista. El peor caso se da cuando en cada llamada a `insertarOrdenado` se da el peor caso. Llamamos $T_{ins}(i)$ a la cota superior del tiempo de `insertarOrdenado` en una lista de largo i. Entonces

$$T(n) \leq \sum_{i=0}^{n}(T_{ins}(i) + c_{ord}) \leq \sum_{i=0}^{n}(c_{ins}\ i + c_{ord}) = c_{ins}n(n+1)/2 + c_{ord}(n+1).$$

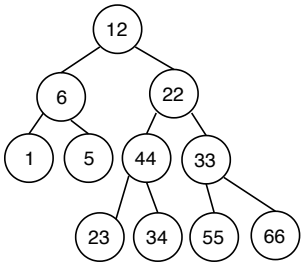
Se deduce que $T(n) = O(n^2)$.

Problema 3 (14 puntos)

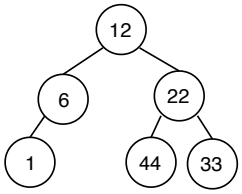
Considere la siguiente definición del tipo `AB` de árboles binarios de enteros:

```
struct nodoAB {
  int dato ;
  nodoAB *izq ;
  nodoAB *der ;
} ;
typedef nodoAB *AB ;
```

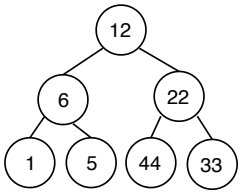
(a) Un árbol perfecto es un árbol en el cual todos los nodos interiores tienen dos hijos y todos las hojas están en el mismo nivel, ver Figura 2. Un árbol vacío y un árbol con un sólo nodo son árboles perfectos.



(a) No es árbol perfecto porque las hojas están en diferentes niveles.



(b) No es árbol perfecto porque el nodo interior 6 no tiene 2 hijos.



(c) Sí es árbol perfecto.

Figura 2: Árbol perfecto.

Implemente una función recursiva `maxAlturalPerfecto` que, dado un árbol `t` devuelve la altura del subárbol máximo perfecto con la misma raíz que `t`. Notar que la altura del subárbol máximo perfecto está determinada por el nodo más cercano a la raíz que no tenga alguno de los hijos, ver nodo 6 de la Figura 2b. Recordar que la altura del árbol vacío es 0 y la de un árbol con un sólo nodo es 1.

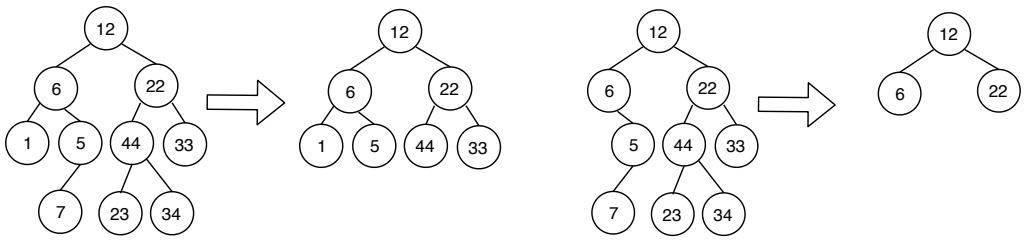
```
int maxAlturaPerfecto(AB t)
```

(b) Implemente un procedimiento recursivo `borrarRecursivo` que, dado un árbol `t` lo elimina completamente, liberando la memoria ocupada.

```
void borrarRecursivo(AB & t)
```

(c) Implemente un procedimiento que, dado un árbol `t` borra todos los nodos necesarios para obtener el subárbol máximo perfecto con la misma raíz que `t`, ver Figura 3.

```
void obtenerPerfecto(AB & t)
```



(a) Subárbol perfecto máximo de altura 3.

(b) Subárbol perfecto máximo de altura 2.

Figura 3: Ejemplos de aplicar `obtenerPerfecto`.

Solución:

(a)
```
int maxAlturaPerfecto(AB t) {
  int res = 0;
  if (t != NULL) {
    int hizq = maxAlturaPerfecto(t->izq);
    int hder = maxAlturaPerfecto(t->der);
    res = 1 + ((hizq <= hder)?hizq:hder); // 1+min(hizq,hder);
  }
  return res;
}
```

(b)
```
//Para el arbol vacio, no tiene efecto.
void borrarRecursivo(AB &t) {
  if (t != NULL) {
    borrarRecursivo(t->izq);
    borrarRecursivo(t->der);
    delete t;
    t = NULL;
  }
}
```

(c)
```
/* Borra de 't' los nodos que están en un nivel mayor a 'altura'.
   Precondición: Los niveles de 't' desde 1 hasta 'altura' están completos. */
void podar(int altura, AB &t) {
  if (altura == 0)
    borrarRecursivo(t);
  else {
    // t no es vacio por precondición
    podar(altura - 1, t->izq);
    podar(altura - 1, t->der);
  }
}
//Para el arbol vacio, no tiene efecto.
void obtenerPerfecto(AB &t) {
  podar(maxAlturaPerfecto(t), t);
}
```