Facultad de Ingeniería

Examen de: Estructuras de datos y Algoritmos 1			**Código de materia:**
Fecha: Mayo de 2022	**Id Examen:**	**Acta:**	**Hoja 1** de **2**

Problema 1 (33 puntos)

a) Defina un procedimiento ***abb_a_lista*** que dado un árbol binario de búsqueda de enteros (de tipo *ABB*) inserte los elementos que sean números pares en una lista (de tipo *Lista*) que se asume inicialmente vacía. Los elementos en la lista deberían estar ordenados de mayor a menor. Si no hay elementos pares en el árbol, en particular si éste es vacío, la lista deberá quedar vacía. No se permite definir funciones o procedimientos auxiliares.

```
typedef struct nodoABB * ABB
struct nodoABB { int dato; ABB izq, der; }

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

void abb_a_lista (ABB t, Lista & l)
```

b) Indique el tiempo de ejecución en el peor caso y el caso promedio ***abb_a_lista***. Justifique.

Problema 2 (33 puntos)

Considere un árbol general de enteros representado mediante un árbol binario de enteros con la semántica: puntero al primer hijo (pH), puntero al siguiente hermano (sH).

```
typedef struct nodoAG * AG;
struct nodoAG { int dato; nodoAG *pH, *sH; }
```

Implemente la función ***bool existeNodo (AG t, int n)*** que retorne true si y solo si en el árbol *t* existe al menos un nodo que tiene *n* o más hijos, asumiendo *n*>0 y t->sH==NULL (si el árbol *t* no es vacío, el nodo raíz no tiene hermanos). Si *t* es NULL la función deberá retornar false. Si utiliza operaciones auxiliares, deberá implementarlas.

Problema 3 (34 puntos)

Considere la siguiente especificación, con pre y postcondiciones, de un TAD *cola de prioridad* no acotado de elementos de tipo string (*char* *) con prioridades que toman valores enteros:

```
// PRE: -
// POS: retorna una nueva cola de prioridad vacía
ColaPrioridad crearColaPrioridad ();


// PRE: -
/* POS: inserta un string s con prioridad p a la cola de prioridad. Los elementos
con igual prioridad se consideran en orden FIFO. */
void encolar(ColaPrioridad &cp, char* s, int p);
```

```
// PRE: -
// POS: retorna la cantidad de elementos presentes en la cola de prioridad
unsigned int cantidad (ColaPrioridad cp);

// PRE: cantidad(cp)!=0
/* POS: retorna y elimina el elemento con mayor prioridad (mayor valor entero) de la
cola de prioridad. Ante elementos de igual prioridad máxima, retorna y elimina el
más antiguo (orden FIFO) */
char* obtener(ColaPrioridad &cp);

// PRE: -
// POS: retorna una copia de la cola de prioridad parámetro sin compartir memoria
ColaPrioridad copia (ColaPrioridad cp);

// PRE: -
// POS: destruye la cola de prioridad, liberando su memoria
void destruir (ColaPrioridad &cp);
```

Se pide:

a) Defina una representación del TAD en la que las operaciones ***crearColaPrioridad, encolar*** y ***cantidad*** tengan O(1) de tiempo de ejecución en el peor caso. Justifique brevemente la elección de la representación, pero no se pide escribir los códigos de las operaciones del TAD.

b) Implemente la función ***bool indistinguibles (ColaPrioridad cp1, ColaPrioridad cp2)*** que dadas dos colas de prioridad retorne true si y solo si son indistinguibles. Esto es, si los elementos de tipo string (char *) que se obtienen de ambas son los mismos y en el mismo orden. La función no deberá acceder a la representación del TAD (su implementación), ni modificar las colas de prioridad parámetro, ni dejar memoria colgada.

SOLUCIONES

Problema 1

Parte a)

```
void abb_a_lista (ABB t, Lista & l){
      if (t != NULL){
            abb_a_lista(t->izq, l);
            if ((a->dato % 2) == 0){
                  LISTA nodo = new nodoLista;
                  nodo->dato = a->dato;
                  nodo->sig = l;
                  l = nodo;
            }
            abb_a_lista(t->der, l);
      }
}
```

Parte b) El orden de tiempo de ejecución en el peor caso y el caso promedio coinciden y es O(n), siendo n la cantidad de nodos del árbol, ya que se recorre todo el árbol realizando operaciones de O(1) para cada nodo. Notar que cada inserción en la lista se hace al comienzo, insumiendo un tiempo constante cada vez.

Problema 2

```
bool existeNodo (AG t, int n){
      if (t == NULL) return false;
      else return (hijos(t) >= n) || existeNodo(t->pH, n) || existeNodo(t->sH, n);
}

// Retorna la cantidad de hijos de t en un árbol pH-sH. Precondición: t != NULL.
int hijos (AG t){
      int cantHijos = 0;
      t = t->pH;
      while (t != NULL){
            cantHijos++;
            t = t->sH;
      }
      return cantHijos;
}
```

Problema 3

Parte a)

```
struct nodoCola {
   char* dato;
   int prioridad;
   nodoCola* sig;
};
```

```
struct representacionColaPrioridad {
  nodoCola* ppio;
  unsigned int largo;
};

typedef representacionColaPrioridad* ColaPrioridad;

/* Para cumplir los requisitos de orden se mantienen: un puntero al inicio de una
lista simplemente enlazada (donde cada nodo tiene una cadena, su prioridad y un
puntero al eventual siguiente nodo) y un entero con la cantidad de elementos presentes
en la lista. Las inserciones se hacen al inicio de la lista y se aumenta el largo en
1 para asegurar O(1) peor caso. El O(1) de la operación cantidad se logra con el
atributo largo de la representación, y para crearColaPrioridad simplemente se crea,
en O(1), la representación con la lista vacía (ppio) y largo igual a 0. Notar que la
operación obtener debería buscar y eliminar el nodo de la lista con prioridad mayor;
si esta prioridad se repite se debe borrar la última instancia para respetar el orden
FIFO */
```

Parte b)

```
bool indistinguibles(ColaPrioridad cp1, ColaPrioridad  cp2){
  clon_cp1 = copia(cp1);
  clon_cp2 = copia(cp2);
  while (cantidad(clon_cp1)!=0 && cantidad(clon_cp2)!=0 &&
        strcmp(obtener(clon_cp1), obtener(clon_cp2))==0){}
  bool res = (cantidad(clon_cp1)==0 && cantidad(clon_cp2)==0);
  destruir(clon_cp1);
  destruir(clon_cp2);
  return res;
}
```