Escuela de Ingeniería

Examen de: Estructuras de datos y Algoritmos 1 **Código de materia: 1774**
Fecha: 19-10-2022 **Grupo: Todos** **Hoja 1 de 4**
Duración: 2 horas **Sin Material**

Ejercicio 1

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 **`int maximoHijos(nodoAG * raiz)`** que dado un árbol general retorna el valor máximo de número de hijos entre todos sus nodos. Si el árbol es vacío retorna cero. Asuma que existe una función `int cantHijos(nodoAG * raiz)`, que retorna la cantidad de hijos que dependen del parámetro raíz, que no puede ser vacío (es precondición).

```
int maximoHijos(nodoAG* r){
	if (r){
		int cantR = cantHijos(r); //Se asume implementada
		int maxPh = 0;
		if (r->pH) maxPh = maximoHijos(r->pH);
		int maxSh = 0;
		if (r->sH) maxSh = maximoHijos(r->sH);
		if (cantR >= maxPh && cantR >= maxSh)
			return cantR;
		if (maxPh >= maxSh)
			return maxPh;
		return maxSh;
	}
	return 0;
}
```

Ejercicio 2

Considere la siguiente definición de listas de enteros:

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

Implemente un procedimiento iterativo **`void quitarMayores(Lista &l, int x)`**, que dados una Lista y un entero "x", elimine de la lista todos los elementos mayores estrictos que el valor x. Si la lista es vacía o no contiene elementos mayores a x, el procedimiento no tendrá efecto.

```
void borro(Lista& l) {
	Lista borrar = l;
	l = l->siguiente;
	delete borrar;
}
void quitarMayores(Lista& l, int x) {
	while (l && l->dato > x)
		borro(l);
	Lista aux = l;
	while (aux) {
		if (aux->siguiente && aux->siguiente->dato > x)
			borro(aux->siguiente);
		else
			aux = aux->siguiente;
```

```
        }
}
```

Ejercicio 3

Una empresa quiere gestionar su stock de productos. Cada producto se identifica por un número entero no negativo y el precio del producto de tipo float. Pueden existir varias unidades de un mismo producto, por lo que también se registra la cantidad en existencia.

Para administrar las existencias se define el TAD Stock con la siguiente especificación:

```
struct representacionStock;
typedef representacionStock* Stock;

//POS: retorna un nuevo Stock vacío
Stock crearStock();

//POS: agrega el producto id al Stock si no existe.
//     Si existe modifica precio y cantidad
void agregar(Stock& s, unsigned int id, float precio, int cantidad);

//POS: elimina cantidad de unidades del producto id del Stock, si existe.
//     Si no existe, o la cantidad existente es menor, elimina el producto id.
void eliminar(Stock& s, unsigned int id, int cantidad);

//POS: muestra existencias en stock: id, precio y cantidad, ordenado por id
void listar(Stock s); //O(n)

//POS: retorna la cantidad de unidades existentes del producto id
int cantidad(Stock s, unsigned int id);

//PRE: cantidad(id)>0
//POS: retorna el precio asignado al producto id
float precio(Stock s, unsigned int id);

//POS: retorna la cantidad total de productos diferentes en Stock
int cantProductosDiferentes(Stock s); // O(1)

//POS: destruye el Stock liberando memoria
void destruir(Stock& s);
```

a) REPRESENTACIÓN

```
struct representacionStock {
       nodoAB* raiz;   //para O(log n) caso promedio de precio
       int cantTotal; //para O(1) peor caso de cantProductosDiferentes
};
struct nodoAB {
       unsigned int id;
       float precio;
       int cantidad;
       nodoAB* izq;
       nodoAB* der;
};
```

b) IMPLEMENTACIÓN

```
void agregar(Stock& s, unsigned int id, float precio, int cantidad) {
	if (agregarAux(s->raiz, id, precio, cantidad))
		s->cantTotal += 1;
}

//auxiliar para insertar en ABB, retorna true si agregó nuevo producto, false si ya existía
bool agregarAux(nodoAB* &r, unsigned int id, float precio, int cant) {
	if (!r) {
		r = new nodoAB;
		r->id = id;
		r->precio = precio;
		r->cantidad = cant;
		r->izq = r->der = NULL;
		return true;
	}
	if (r->id == id) {
		r->precio = precio;
		r->cantidad += cant;
		return false;
	}
	if (r->id > id)
		return agregarAux(r->izq, id, precio, cant);
	else
		return agregarAux(r->der, id, precio, cant);
}



void listar(Stock& s) {
	listarAux(s->raiz);
}

//auxiliar para listar el ABB inorder
void listarAux(nodoAB* r) {
	if (r) {
		listarAux(r->izq);
		cout << r->id << " " << r->precio << " " << r->cantidad << endl;
		listarAux(r->der);
	}
}



float precio(Stock& s, unsigned int id) {
	return precioAux(s->raiz, id);
}
```

```
//auxiliar para buscar en ABB y retornar precio del id
float precioAux(nodoAB* r, unsigned int id) {
	if (!r)
		return 0;
	if (r->id > id)
		return precioAux(r->der, id);
	else if (r->id < id)
		return precioAux(r->izq, id);
	else
		return r->precio;
}


int cantProductosDiferentes(Stock& s) {
	return s->cantTotal;
}
```

c) VALOR TOTAL DEL STOCK

```
//PRE: -
//POS: retorna el valor económico de la existencia en Stock de los productos con id entre 0 y k
float valorDelStock(Stock& s, int k) {
	int acum = 0;
	for (int i = 0; i <= k; i++) {
		int cant = cantidad(s, i);
		if (cant != 0)
			acum += precio(s, i);
	}
	return acum;
}
```