Universidad De La República | Administración Nacional de la Educación Pública

Consejo de Educación Técnico Profesional | Facultad de Ingeniería

Estructuras de Datos y Algoritmos

Examen 19/02/2013

_____	_____	_____
Nombre y Apellido	C.I.	Turno

Por favor siga las siguientes indicaciones:

- Escriba con lápiz.
- Escriba su nombre y c.i. en todas las hojas que entregue.
- Numere las hojas he indique el total de hojas en la primera de ellas.
- Escriba las hojas de un solo lado.
- Comience cada ejercicio en una hoja nueva.
- El total máximo de puntos del examen es de **100.**
- Se necesitan 60 puntos para aprobar.
- El parcial contiene un total de: **3 carillas.**
- La prueba es individual y sin material.
- Solo se contestan dudas acerca de letra de los ejercicios.

Ejercicio 1(30 Puntos)

Considere la siguiente declaración, en C/C++, del tipo *ListaEnt* de listas encadenadas de enteros:

```
struct nodoLista {
      int info;
      nodoLista *sig; };
typedef nodoLista* ListaEnt;
```

a) Implemente una función ***sinPositivos*** que dadas dos listas de ListaEnt, devuelva una tercera lista con los elementos negativos de las listas pasadas como parámetro. La lista resultado no debe compartir memoria con las listas pasadas como parámetro. La lista resultado puede tener elementos repetidos. Si no hay elementos negativos se debe retornar la lista vacía.

Nota: No se pueden utilizar funciones auxiliares, ni recorrer las listas parámetro o la lista resultado más de una vez.

b) ¿Cuál es el orden del tiempo de ejecución en el peor caso de la operación de la parte a)? Justifique.

c) Si estuvieran ordenadas las listas parámetro ¿cambia el orden de tiempo de ejecución? Justifique.

Ejercicio 2(35 Puntos)

Considere las siguientes declaraciones, en C/C++, del tipo de los árboles binarios de búsqueda de enteros, de tipo ABB:

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

Se pide:

a) Implementar una función que dado un ABB de enteros y un numero entero X devuelva una copia de los X primeros niveles sin compartir memoria con el árbol pasado como parámetro.

b) Implemente un procedimiento recursivo ***imprimirMenores*** que, dados un árbol binario de búsqueda de enteros A de tipo ABB (sin elementos repetidos) y un entero x, imprima todos los elementos de A que son menores que x, ordenados de mayor a menor.

NOTAS: El procedimiento debe evitar recorrer todo el árbol, si no es estrictamente necesario hacerlo. No se permiten definir estructuras de datos auxiliares.

Ejercicio 3(35 Puntos)

a) De una especificación de un TAD **Cola de Prioridad** no acotada de elementos de un tipo genérico T, donde las prioridades estén dadas por números enteros y que permita obtener, y eliminar, elementos tanto de mínima como de máxima prioridad. Considere un conjunto mínimo de operaciones constructoras, selectoras/destructoras (Min, BorrarMin, Max, BorrarMax) y predicados. Tenga en cuenta que pueden haber elementos de igual prioridad estos se almacenan según el orden de llegada.

b) Desarrolle una implementación del TAD anterior donde las operaciones selectoras Min y Max, y los predicados tengan O(1) de tiempo de ejecución en el peor caso. Se debe dar una representación e implementar las operaciones constructoras, las selectoras/destructoras Min y BorrarMin, y los predicados. Omita el código de Max y Borrar Max. No se permite usar TADs auxiliares en este ejercicio.

c) ¿Qué varía en la especificación de la parte a) si la Cola de Prioridad es acotada? Detalle brevemente.

Solución ejercicio 1

Parte a)

```
ListaEnt sinPositivos(ListaEnt l1, ListaEnt l2){
	ListaEnt lres = NULL;
	ListaEnte lres_ult = NULL;
	ListaEnt laux1=l1;
	ListaEnt laux2=l2;

	while(laux1 != NULL || laux2 !=NULL){

		if ((laux1 != NULL)&&(laux1->info < 0)){
		  if (lres == NULL){
		    lres = new nodoLista;
		    lres->info = laux1->info;
		    lres->sig = NULL;
		    lres_ult=lres;
		  }
		  else {
		    lres_ut->sig = new nodoLista;
		    lres_ut = lres_ut->sig;
		    lres_ult->info = laux1->info;
		    lres_ult->sig = NULL;
		    lres_ult=lres;
		  }
		}

		if ((laux2 != NULL)&&(laux2->info < 0)){
		  if (lres == NULL){
		    lres = new nodoLista;
		    lres->info = laux2->info;
		    lres->sig = NULL;
		    lres_ult=lres;
		  }
		  else {
		    lres_ut->sig = new nodoLista;
		    lres_ut = lres_ut->sig;
		    lres_ult->info = laux2->info;
		    lres_ult->sig = NULL;
		    lres_ult=lres;
		  }
		}
	}
	return lres;
}
```

Parte b)

El orden del tiempo de ejecución en el peor caso es n, siendo n el largo máximo entre los largos de l1 y l2. Esto es porque en el while se itera tantas veces como elementos tenga la lista más larga y dentro de este while se realizan operaciones simples de orden 1.

Parte c)

Si las listas estuvieran ordenadas la ventaja que se tiene es que no es necesario recorrerlas todas, ya que cuando se encuentre un elemento mayor o igual a cero, sabemos que no hay más elementos negativos en el resto de la lista.
Sin embargo el orden del tiempo de ejecución no cambia ya que debemos considerar el peor caso y ese es en el que al menos todos los elementos de la lista de mayor largo son negativos. Siendo que la lista de mayor largo (sea n este largo) es la que pauta el orden y tiene todos sus elementos negativos, se la debe recorrer toda, por tanto el orden es n, igual que en la parte anterior.

Solución ejercicio 2

Parte a)

```
void copia_niveles(ABB A, int X, ABB & A_copia){

    if ((A!=NULL)&&(X>0)){
      A_copia = new nodoABB;
      A_copia->dato = A->dato;
      A_copia->izq = NULL;
      A_copia->der = NULL;

      copia_niveles(A->izq, X-1,A_copia->izq);
      copia_niveles(A->der, X-1,A_copia->der);
    }
}
```

Parte b)

```
void imprimirMenores(ABB A, int x){
    if (A!=NULL){
     if (A->dato < x){
        imprimirMenores(A->der, x);
        printf("%d ",A->dato);
        imprimirMenores(A->izq, x);
     }
     else {
       imprimirMenores(A->izq, x);
     }
    }
}
```

Solución ejercicio 3

Parte a)

```
struct nodoCola{
    T elem;
    int prio;
    nodoCola * sig;
}

struct cabezal{
    nodoCola * inicio;
    nodoCola * fin;
}


typedef cabezal * Cola;

//constructoras

Cola CreoColaVacia();
//Devuelve una cola con el cabezal creado pero sin elementos.

void encolar(Cola & C, T elem, int prio);
//Agrega a la cola C el elemento elem, en la posicion de acuerdo a la prioridad prio
//Los elementos se ordenan de acuerdo a la prioridad de menor a mayor (menor prioridad
significa mayor importancia)

//predicados
bool esVacia(Cola C);
//Devuelve true si la cola C es vacía y false en caso contrario


//selectoras

//Precondición: la cola C no es vacía.
T obtenerMin(Cola C);
//Devuleve el elemento de C que tiene mínima prioridad.

//Precondición: la cola C no es vacía.
T obtenerMax(Cola C);
//Devuleve el elemento de C que tiene máxima prioridad.
```

```
//destructuras

//Precondición: la cola C no es vacía.
void borrarMin(Cola & C);
//Borra el elemento de C que tiene mínima prioridad

//Precondición: la cola C no es vacía.
void borrarMax(Cola & C);
//Borra el elemento de C que tiene máxima prioridad

void destruirCola(Cola & C);
//Destruye la cola C, liberando toda la memoria asociada a la misma.
```

Parte b)

```
//constructoras

Cola CreoColaVacia(){
    Cola C = new cabezal;
    C->inicio = NULL;
    C->fin = NULL;
    return C;
}

void encolar(Cola & C, T elem, int prio){

    Cola act =  C->inicio;
    Cola ant =  NULL;

    while (act != NULL && act->prio >= prio){
        ant = act;
        act = act -> sig;
    }

    if ((ant != NULL)&&(act !=NULL)){
      //agrego en el medio o al final
      ant->sig = new nodoCola;
      ant = ant->sig;
      ant->elem = T;
      ant->prio = prio;
      ant->sig = act;

      if (act == NULL){
        //agregue al final, actualizo el fin
        C->fin = ant;
```

```
      }

    }
    else if (ant == NULL){
        //agrego al principio
        C->inicio = new nodoCola;
        C->inicio->elem = T;
        C->inicio->prio = prio;
        C->inicio->sig = act;

        if (act == NULL){
          //la cola estaba vacia antes de agregar este elemento
          C->fin = C->inicio;
        }
    }
}

//predicados

bool esVacia(Cola C){
    if (C==NULL)
      return true;
    else if (C->inicio == NULL)
          return true;
        else return false;
}

//selectoras

//Precondición: la cola C no es vacía.
T obtenerMin(Cola C){
  return C->fin->T;
}


//Precondición: la cola C no es vacía.
T obtenerMax(Cola C){
  return C->incio->T;
}

//destructuras

//Precondición: la cola C no es vacía.
void borrarMin(Cola & C){
    Cola act =  C->inicio;
```

```
    Cola ant =  NULL;

    while (act != NULL){
         ant = act;
         act = act -> sig;
    }

    if (ant != C->inicio){
      //la cola tenía al menos 2 elementos
      delete act;
      ant->sig = NULL;
      C->fin = ant;
    }
    else {
        //la cola tenía un solo elemento
        delete act;
        C->incio = NULL;
        C->fin = NULL;
    }
}

//Precondición: la cola C no es vacía.
void borrarMax(Cola & C){
    Cola c_aux = C->incio;
    C->incio = C->incio->sig;
    delete c_aux;
}

void destruirCola(Cola & C){

    Cola act =  C->inicio;
    Cola ant =  NULL;

    //libero la memoria de los nodos
    while (act != NULL){
         ant = act;
         act = act -> sig;
         delete ant;
    }

    delete C; //libero la memoria del cabezal
}
```

Parte c)

Si la cola de prioridad fuera acotada podría utilizar un Array con Tope para su representación. Los elementos se guardarían en el arreglo en lugar de en la lista encadenada y el elemento de mayor prioridad se encontraría en la posición 0 y el de menor prioridad en la posición tope -1.