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

Segundo parcial de Programación 2

Junio de 2023

Problema 1 (21 puntos: 14+7)

Considere un TAD **Cola de Prioridad (*CP*) no acotada** de elementos de un tipo genérico *T* donde las prioridades están dadas por números naturales. Se admiten prioridades y elementos repetidos.

struct ***representacionCP****;* *typedef representacionCP ** ***CP****;* ***CP crear*** *();* *// Devuelve la cola de prioridad vacía* ***void agregar (T x, unsigned int p, CP & cp);*** *// Agrega x con prioridad p a cp* ***bool esVacia*** *(**CP** cp);* *// Retorna true si y solo si cp es vacía*	***T prioritario*** *(**CP** cp);* */* Retorna el elemento con mayor prioridad (valor más grade) ingresado en cp. Ante igual prioridad retorna el primero en ingresar */* *// Precondición: !esVacia(cp)* ***void eliminar*** *(**CP** & cp);* */* Remueve el elemento con mayor prioridad (valor más grade) ingresado en cp. Ante igual prioridad elimina el primero en ingresar */* *// Precondición: !esVacia(cp)*

Parte a)

Defina una **representación** para el TAD *CP* (<u>*representacionCP*</u>) de tal manera que las operaciones *crear*, *agregar*, *esVacia* y *prioritario* tengan O(1) de tiempo de ejecución en el peor caso. **Explique** para cada una de estas cuatro operaciones como se cumplen las restricciones establecidas para la representación propuesta. Escriba luego el **código de *crear* y *agregar***, y asuma implementadas las restantes operaciones (<u>no escriba sus códigos</u>).

Parte b)

Sin considerar la Parte a), defina ahora una **representación** para el TAD *CP* (<u>*representacionCP*</u>) de tal manera que las operaciones *crear*, *esVacia*, *prioritario* y *eliminar* tengan O(1) de tiempo de ejecución en el peor caso. **Explique** para cada una de estas cuatro operaciones como se cumplen las restricciones establecidas para la representación propuesta. <u>No escriba el código de las operaciones del TAD</u>.

Problema 2 (34 puntos: 17+17)

Considere la siguiente representación de árboles binarios de búsqueda de enteros (ABB), con información de la altura en cada nodo:

```
struct nodoABB{
  int dato;
  unsigned int altura; // altura del árbol que tiene al nodo como raíz
  nodoABB * izq, * der;
}
typedef nodoABB * ABB;
```

Notar que en estos árboles la altura puede calcularse en O(1) peor caso:

```
unsigned int altura (ABB t){
  if (t==NULL) return 0; else return t->altura;
}
```

Parte a)

Adaptar la función vista en el curso que inserta un elemento *x* en un árbol binario de búsqueda *t*, de tal manera que además de agregar *x* en *t*:

- actualice el campo *altura* en los nodos en que corresponda, y
- asumiendo que *t* cumple la condición de AVL, devuelva true si y solo si el árbol sigue cumpliendo esa condición después de la inserción. Notar que NO se pide balancear el árbol.

```
// pre: x no pertenece a t
bool insertar(ABB & t, int x)
```

La función ***insertar*** debe ser **recursiva** y no puede usar funciones ni procedimientos auxiliares, excepto la función *altura* dada y operaciones entre enteros como *max* (que retorna el máximo entre dos enteros) y *abs* (que retorna el valor absoluto de un entero), que se asumen implementadas.

Parte b)

Implementar un procedimiento iterativo ***imprimir*** que imprima los datos (de tipo *int*) de un *ABB t* **por niveles**, desde la raíz y en cada nivel de izquierda a derecha. El procedimiento debe recorrer el árbol una sola vez, utilizando una cola de árboles de tipo *ABB*, llamada *ColaABB,* que se asume implementada (NO debe implementar *ColaABB*). Utilice un procedimiento *impInt*, que se asume implementado, para imprimir valores (datos) de tipo *int*. No se permite usar otros TADs o estructuras de datos auxiliares.

```
void imprimir(ABB t)
```

ColaABB crear (); *// Devuelve la cola vacía de árboles* *void* ***encolar*** *(**ABB** t, **ColaABB** & c);* *// Inserta el árbol t en la cola c*	***bool esVacia (ColaABB** c);* *// Retorna true si y solo si la cola c es vacía* ***ABB desencolar (ColaABB** & c);* *// Remueve y retorna el primer árbol ingresado en c.* *// Precondición: !esVacia(c)*

Soluciones

Problema 1

Parte a)

La representación propuesta contiene: una lista simplemente encadenada con puntero al inicio (*lista*) y un puntero al nodo prioritario de la lista (*prioritario*):

```
struct nodoLista{
        T dato;
        unsigned int prioridad;
        nodoLista * sig:
}
```

```
struct representacionCP{
        nodoLista *  lista;
        nodoLista * prioritario;
}
```

crear genera un *CP* con *lista* y *prioritario* en NULL; ***agregar*** inserta el elemento con la prioridad dada en un nuevo nodo al inicio de la *lista* y actualiza *prioritario* con el puntero a este nodo si *prioritario* es NULL o la prioridad parámetro es mayor estricta que la del nodo *proritario*; ***esVacia*** retorna true si la *lista* es vacía; y ***prioritario*** retorna el *dato del nodo* al que apunta *prioritario*. Todas las operaciones son O(1) peor caso. La operación ***eliminar*** debería recorrer la lista para eliminar el (nodo) prioritario y actualizar el acceso al nuevo prioritario; esto es, con la dirección del nodo más cercano al final de la lista con el valor mayor de prioridad (si está repetida la prioridad, el primero en llegar es quien está más cerca del final de la lista ya que se inserta siempre al comienzo).

```
CP crear(){
        CP ret = new representacionCP;
        ret-> lista = NULL;
        ret->prioritario = NULL;
        return ret;
}
```

```
CP agregar(T x, unsigned int p, CP & cp){
        nodoLista * nodo = new nodoLista;
        nodo->dato = T;
        nodo->prioridad = p;
        nodo->sig = cp-> lista;
        cp->lista = nodo;
        if (cp->prioritario == NULL ||
            p > cp->prioritario->prioridad)
                cp->prioritario = nodo;
}
```

Parte b)

Una lista simplemente encadenada (de nodos de tipo *nodoLista,* de la Parte a) ordenada por prioridad (de mayor a menor) permitiría tener al elemento prioritario al inicio. Las inserciones serían ordenadas y en caso de prioridades repetidas, al agregar un elemento debería ir al final de los que tienen la misma prioridad. De esta manera, obtener el elemento prioritario y eliminarlo es O(1) peor caso, al igual que crear la lista vacía y chequear si está vacía. Esto es ***crear***, ***esVacia***, ***prioritario*** y ***eliminar*** tendrían O(1) de tiempo de ejecución en el peor caso.

Problema 2

Parte a)

```
// Precondición: x no está en t
bool insertar(ABB & t, int x){
        if (t==NULL){
                t = new nodoABB;
                t->dato = x;
                t->altura = 1;
                t->izq = t->der = NULL;
                return true;
        }
        else{   bool res;
                if (x < t->dato)
                        res = insertar(t->izq, x);
                else    res = insertar(t->der, x);
                t->altura = max(altura(t->izq), altura(t->der))+1;
                return  res && (abs(altura(t->izq) - altura(t->der)) <= 1);
        }
}
```

Parte b)

```
void imprimir (ABB t){
        ColaABB c = crear();
        if (t != NULL) encolar(t, c);
        while (!esVacia(c)){
                t = desencolar(c);
                impInt(t->dato);
                if (t->izq != NULL) encolar(t->izq, c);
                if (t->der != NULL) encolar(t->der, c);
        }
}
```