Examen de Programación 2

Febrero de 2024

Problema 1 (15 puntos)

a) Explique brevemente en qué se diferencian los árboles binarios de búsqueda genéricos de los árboles AVL.

Los árboles binarios de búsqueda son casos particulares de árboles binarios en los que para cada nodo, su valor es mayor al de todos los nodos de su subárbol izquierdo y menor al de todos los nodos de su subárbol derecho. Los AVL o árboles balanceados son casos particulares de árboles binarios de búsqueda que además de la relación de orden entre sus elementos, cumplen una condición de equilibrio: para cada nodo del árbol, la altura de sus subárboles izquierdo y derecho puede diferir en a lo sumo 1.

b) Dada la especificación de un TAD que representa colecciones no acotadas explique cómo modificaría la especificación para que el TAD represente colecciones acotadas.

La función de creación del TAD pasaría a recibir un parámetro que representa la cota. Si bien esto suele ser lo usual, la cota podría manejarse como una constante, en cuyo caso no sería necesario este parámetro adicional.

Por otro lado, la función de inserción debe tener en cuenta la cota, de manera de no sobrepasarla al momento de insertar. Para esto existen varias posibilidades. Una, insertar solamente si el TAD no está lleno, sin hacer nada en caso contrario. También se puede sustituir un elemento utilizando alguna política de reemplazo (elemento más viejo, elemento menos visitado, al azar, etc.). Una tercera alternativa consiste en agregar como precondición que la colección no esté llena; en esta opción, para poder verificar si se cumple la precondición, es necesario agregar un predicado, o un par de funciones que informen el valor de la cota y la cantidad de elementos presentes.

Problema 2 (30 puntos)

Considere la siguiente definición del tipo ***Lista*** para listas de enteros:

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

a) Defina e implemente un procedimiento **insertar** que agrega un elemento al principio de una lista.

```
void insertar(Lista &l, int dato){
   Lista nuevo = new nodoLista;
   nuevo->dato = dato;
   nuevo->sig = l;
   l = nuevo;
}
```

Examen de Programación 2

Febrero de 2024

b) Implemente la función ***merge*** que dados dos arreglos ***a1*** y ***a2*** de enteros de tamaño ***n*** *(con* ***n****>0)*, **ordenados de menor a mayor**, retorna una lista de tipo *Lista* que contenga los elementos presentes en ambos arreglos, **ordenados de menor a mayor**. Los arreglos parámetros pueden contener elementos repetidos, al igual que la lista resultado.

PRE: a1 y a2 están ordenados de menor a mayor, n > 0

Lista merge (int a1, int* a2, int n)*

Ejemplo: a1 = < 2, 4, 4, 6, 7 >, a2 = < 1, 2, 4, 8, 9 >, n = 5 → lista resultado = [1, 2, 2, 4, 4, 4, 6, 7, 8, 9]

La función ***merge*** debe ser O(***n***) en el peor caso, y la implementación debe ser iterativa y usar el procedimiento **insertar**.

```
Lista merge(int* a1, int* a2, int n){
   Lista res = NULL;

   int cont1, cont2;
   cont1 = n-1;
   cont2 = n-1;

   while(cont1>=0 && cont2>=0){
      if(a1[cont1] > a2[cont2]){
            insertar(res,a1[cont1]);
            cont1--;
      }else{
            insertar(res,a2[cont2]);
            cont2--;
      }
   }

   for(int i=cont1; i>=0; i--){
      insertar(res,a1[i]);
   }

   for(int j=cont2; j>=0; j--){
      insertar(res,a2[j]);
   }

   return res;
}
```

c) Justifique brevemente el orden de tiempo de ejecución de la implementación de ***merge***.

La función recorre una vez cada elemento de los arreglos, que son ambos de tamaño n. De esta forma, se visitan 2n elementos, realizando en cada uno de ellos operaciones de tiempo constante (la función *insertar* es O(1)). Por esto, la función *merge* es O(n).

Problema 3 (55 puntos)

Considere un TAD ***Conjunto*** de números reales (de tipo *float*) que tiene (entre otras) las siguientes operaciones:

- ***Conjunto crear ()***, que retorna el conjunto vacío.
- ***void insertar (Conjunto & c, float x)***, que agrega *x* a *c* si *x* no pertenece a *c*.
- ***float min (Conjunto c)***, que retorna el mínimo elemento del conjunto *c*, que se asume no vacío (precondición).
- ***void union (Conjunto & c1, Conjunto c2)***, que agrega los elementos de *c2* en *c1*, sin modificar *c2*.

Desarrolle una implementación del TAD ***Conjunto*** en la que ***crear*** y ***min*** sean O(1) en el peor caso e ***insertar*** sea $O(\log_2(n))$ en el caso promedio, siendo *n* la cantidad de elementos del conjunto. Concretamente:

a) Defina *representacionConjunto*, teniendo en cuenta que el tipo ***Conjunto*** es un puntero a su representación (*representacionConjunto*). Puede definir tipos auxiliares. Explique cada campo de la representación elegida.

Para satisfacer el requerimiento de orden $O(\log_2(n))$ de la función *insertar* se puede utilizar un ABB con valores de tipo *float*. Esta estructura, además, es compatible con el requerimiento O(1) de *crear*. El único inconveniente es que la función *min* no es O(1) para ABBs ya que se debe realizar una búsqueda entre los elementos del árbol. Una solución posible es agregar un campo a la representación que mantenga el mínimo actual del árbol, y que sea modificado en caso de que este cambie (por ejemplo, en una inserción).

```
struct nodoABB{
    float dato;
    nodoABB* izq;
    nodoABB* der;
};

typedef nodoABB* ABB;

struct representacionConjunto{
    ABB elems;
    float min;
};

typedef representacionConjunto* Conjunto;
```

NO es aceptable una implementación en la que los elementos se mantienen en una lista, esté ordenada o no, porque para insertar hay que recorrer todos los elementos para determinar si el elemento a insertar pertenece al conjunto.

b) Implemente *crear*, ***insertar***, ***min*** y ***union***. Para implementar ***union*** utilice la operación ***insertar***.

```
Conjunto crear(){
    Conjunto c = new representacionConjunto;
    c->elems = NULL;
    return c;
};

void insertarABB(ABB &a, float dato){
```

```
    if(a == NULL){
        a = new nodoABB;
        a->dato = dato;
        a->izq = NULL;
        a->der = NULL;
    }else if(a->dato > dato){
        insertarABB(a->izq, dato);
    }else if(a->dato < dato){
        insertarABB(a->der, dato);
    }
}

void insertar(Conjunto &c, float x){
    if((c->elems == NULL) || (x < c->min))
        c->min = x;
    insertarABB(c->elems, x);
};

float min(Conjunto c){
    return c->min;
};

void unionABB(ABB &a1, ABB a2){
    if(a2 != NULL){
        unionABB(a1,a2->izq);
        unionABB(a1,a2->der);
        insertarABB(a1,a2->dato);
    }
}

void union(Conjunto &c1, Conjunto c2){
    unionABB(c1->elems,c2->elems);
};
```

Se necesita la función auxiliar insertarABB porque, por ejemplo, la expresión insertar(c->elems, x) es un error, ya que el primer parámetro de insertar es de tipo Conjunto y el de esta expresión es de tipo ABB.
Por razones similares se necesita la función auxiliar unionABB porque la expresión union(c1, c2→elems) es un error.

Se puede implementar una solución alternativa, sin el struct ni las funciones auxiliares, manteniendo en cada nodo el valor del mínimo elemento del subárbol del cuál el nodo es raíz. La desventaja de esta implementación el uso de espacio innecesario.

c) Indique el orden de tiempo de ejecución para el *peor caso* y para el *caso promedio* de la operación ***union***. Justifique brevemente ambos órdenes.

El orden de a función union es el mismo que el de unionABB.

La función unionABB realiza una inserción sobre a1 por cada elemento de a2. Llamémosle *n* a la cantidad de elementos de a1 y *m* a la cantidad de elementos de a2. La inserción en un ABB es O(n) peor caso y $O(\log_2(n))$ caso promedio. Como unionABB siempre hace m inserciones, es orden $O(m*n)$ peor caso y $O(m*\log_2(n))$ caso promedio. De esta forma, *union* también es $O(m*n)$ peor caso y $O(m*\log_2(n))$ caso promedio.