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

Segundo parcial de Programación 2

Diciembre de 2024

*En la primera hoja poner: **número de parcial (arriba a la derecha, grande!)**, nombre completo, cédula y cantidad de hojas que entrega. En las siguientes hojas indicar el nombre, la cédula y el número de hoja. Escribir de manera clara y legible (con lapiz o lapicera). Empezar cada problema en hoja nueva y no escribir las hojas de ambos lados.*

Problema 1 -- 30 puntos: a) 5 + b) 8+14+3

Se desea modelar un TAD ***Votacion*** que permita registrar votos a diferentes listas electorales, de tipo ***char** *. Considere solamente las siguientes operaciones del TAD ***Votacion***:

```
struct RepVotacion; typedef RepVotacion * Votacion;

// PRE: 'cant'>0. POS: Devuelve el TAD Votacion vacío, para el cual se estiman 'cant' listas electorales diferentes.
Votacion crearVotacion (int cant);

// POS: Registra un voto a la lista electoral 'le' en 'v'.
void insertarVoto (Votacion & v, char * le);

// POS: Devuelve la cantidad total de votos registrados en 'v'.
int cantidad (Votacion v);

// POS: Devuelve la cantidad de votos que tiene la lista electoral 'le' en 'v'.
int cantVotos (Votacion v, char * le);

/* PRE: cantidad(v) > 0. POS: Devuelve el nombre de la lista más votada en 'v'. Si hay empate entre dos o más listas
electorales (más votadas), devuelve el nombre de la primera que llegó a la mayor cantidad de votos. */
char * masVotada (Votacion v);
```

a) Defina una representación del TAD ***Votacion*** (***RepVotacion***), en la que ***insertarVoto*** y ***cantVotos*** tengan O(1) de tiempo de ejecución en el caso promedio, y ***cantidad*** y ***masVotada*** tengan O(1) peor caso. Asuma que existe una función ***h*** tal que: ***h(le, cota)*** retorna un entero en el rango **[**0 : ***cota***-1**]**, donde ***le*** representa el nombre de una lista electoral (de tipo **char *)** y ***cota*** es un entero positivo (>0). Considere que ***h*** tiene distribución uniforme (distribuye bien, equitativamente).

b) Implemente ahora las operaciones ***crearVotacion***, ***insertarVoto*** y ***masVotada***, asumiendo definidas las operaciones ***cantidad*** y ***cantVotos***. Considere que los elementos de tipo ***char** *** se pueden asignar y comparar directamente usando los operadores básicos = y ==, como si fueran enteros. Al implementar ***insertarVoto*** asuma que la cantidad de listas electorales diferentes ya votadas no supera a la cantidad estimada inicialmente en ***crearVotacion***.

Problema 2 -- 25 puntos: a) 18 + b) 7

Considere el TAD **Cola de Prioridad (*CP*)** de elementos de un tipo genérico ***T*** donde las prioridades están dadas por números naturales. En estas colas de prioridad, los elementos se pueden repetir pero las prioridades no.

```
// POS: Devuelve la cola de prioridad vacía
CP crear ();

/* POS: Agrega 'x' con prioridad 'p' a 'cp', si 'p' no es una prioridad usada ya en 'cp'. En caso contrario, no tiene efecto */
void agregar (T x, unsigned int p, CP & cp);

// POS: Retorna true si y solo si 'cp' es vacía
bool esVacia (CP cp);

/* PRE: !esVacia(cp). POS: Remueve y retorna el elemento con mayor prioridad (valor más grade) ingresado en 'cp'. */
T prioritario (CP & cp);

// Retorna una copia de 'cp', sin compartir memoria
CP copia (CP cp);

// Destruye la cola de prioridad 'cp', liberando su memoria
void destruir(CP & cp);
```

a) Utilizando únicamente las operaciones especificadas del TAD ***CP***, escriba el código de la operación ***bool indistinguibles (CP cp1, CP cp2)***, que dadas dos colas de prioridad ***cp1*** y ***cp2***, retorna true si y solo si ambas colas de prioridad permiten obtener los mismos elementos y en el mismo orden. Asuma que los elementos de tipo ***T*** se pueden comparar usando ==. Las colas ***cp1*** y ***cp2*** no se deben modificar.

b) Mencione una posible implementación de ***CP*** para la cual ***indistinguibles*** sea O(***max(n.m)***) peor caso, siendo ***n*** y ***m*** las cantidades de elementos de ***cp1*** y ***cp2***. **NO se pide implementar *CP*. Justifique** brevemente el cumplimiento del orden O(***max(n.m)***) para la impementación mencionada.

SOLUCIONES

Problema 1

Parte a)

```
struct nodoHash {
  char * lista;
  int votos;
  nodoHash * sig;
}
```

```
struct repVotacion {
  nodoHash ** tabla; // tabla de hash
  int cota; // tamaño de la tabla; cantidad estimada de listas electorales diferentes
  int cantidad; // cantidad total de votos
  char * mas_votada; // nombre de la lista electoral más votada
  int cant_mas_votada; // cantidad de votos de la lista electoral más votada
}
```

Se usa *open hashing* como representación, ya que se pide que *insertarVoto* y *cantVotos* tengan O(1) de tiempo de ejecución en el caso promedio. Notar que la función *h* cumple el requisito para ser una (buena) función de hash y que la cantidad de listas electorales diferentes puede ser estimada (ver *crearVotacion*). Adicionalmente, dado que se pide que las operaciones *cantidad* y *masVotada* tengan O(1) peor caso, se lleva en la representación también la cantidad total de votos registrados y, el nombre y cantidad de votos de la lista electoral más votada.

Parte b)

```
Votacion crearVotacion (int cant){
  Votacion v = new repVotacion;
  v->tabla = new nodoHash* [cant];
  for (int i=0; i<cant; i++)
        v->tabla[i]=NULL;
  v->cantidad = 0;
  v->cota = cant;
  return v;
}

/* No se inicializa v->mas_votada ni
v->cant_mas_votada acá, ya que no
hay votos en una tabla vacía. Se
considera al insertarVoto */
```

```
void insertarVoto (Votacion & v, char * le){
  nodoHash * l =  v->tabla[h(le, v->cota)];
  int votos = 1;
  while (l != NULL && l->lista != le)
        l = l->sig;
  if (l != NULL){
        l->votos++;  votos = l->votos;
  }
  else{  nodoHash* nuevo = new nodoHash;
        nuevo->lista = le;
        nuevo->votos = votos;
        nuevo->sig = v->tabla[h(le, v->cota)];
        v->tabla[h(le, v->cota)] = nuevo;
  }
  if (v->cantidad == 0 ||
        votos > v->cant_mas_votada){
              v->cant_mas_votada = votos;
              v->mas_votada = le;
  }
  v->cantidad++;
}
```

```
char * masVotada (Votacion v)
{
  return v->mas_votada;
}
```

Problema 2

Parte a)

```
bool indistinguibles (CP cp1, CP cp2){
      if (esVacia(cp1) || esVacia(cp2))
            return (esVacia(cp1) && esVacia(cp2));
      else{
            cp1_copia = copia(cp1);
            cp2_copia = copia(cp2);
            bool iguales = true;
            while (iguales && !esVacia(cp1_copia) && !esVacia(cp2_copia))
                  iguales = (prioritario(cp1_copia) == prioritario(cp2_copia));
            iguales = iguales && esVacia(cp1_copia) && esVacia(cp2_copia);
            destruir (cp1_copia);
            destruir (cp2_copia);
            return iguales;
      }
}
```

Parte b)

Una implementación que cumple lo requerido es una lista simplemente encadenada con puntero al inicio, ordenada por prioridad. El prioritario (si la cola de prioridad no es vacía) está al comienzo de la lista. Cada nodo contiene el elemento, su prioridad y el puntero al siguiente nodo.

Notar que en la implementación anterior, ***copia*** y ***destruir*** tendrían en el peor caso como orden (O) la cantidad de elementos (copia simple de una lista), en tanto que ***esVacia*** y ***prioritario*** serían O(1) pero caso, ya que ambas realizan acciones de tiempo constante al inicio de la lista ordenada (que implementa la cola de prioridad).

```
bool indistinguibles (CP cp1, CP cp2){
        if (esVacia(cp1) || esVacia(cp2))                                   // O(1)
                return (esVacia(cp1) && esVacia(cp2));                      // O(1)
        else{   cp1_copia = copia(cp1);  cp2_copia = copia(cp2);            // O(max(n,m))
                bool iguales = true;                                        // O(1)
                while (!esVacia(cp1_copia) && !esVacia(cp2_copia) && iguales)   // O(min(n,m))
                        iguales = (prioritario(cp1_copia) == prioritario(cp2_copia));   // O(1)
                iguales = iguales && esVacia(cp1_copia) && esVacia(cp2_copia);  // O(1)
                destruir (cp1_copia); destruir (cp2_copia);                 // O(max(n,m))
        }
        return iguales;                                                     // O(1)
}
```

*Siendo **n** y **m** las cantidades de elementos de **cp1** y **cp2**.*

*Por la regla de la suma, **indistinguibles** es **O(max(n,m))** para la implementación considerada de la CP.*