Segundo Parcial de Programación 2

7 de junio de 2017

Problema 1 (30 puntos)

Considere un TAD `T` que permita almacenar a lo sumo M (se asume $M > 0$) enteros donde se respeta la política LIFO (el último en entrar es el primero en salir).

(a) Especifique el TAD `T` considerando operaciones constructoras, selectoras/destructoras y predicados. La operación de inserción no debe tener precondición, esto es: siempre se pueden agregar elementos pero sólo los últimos M (a lo sumo) se consideran. Incluya en la especificación una operación `clon` que, dado un elemento `t` del TAD `T`, retorne una copia idéntica de `t` que no comparta memoria con el parámetro.

(b) Desarrolle una implementación del TAD `T` en la que las operaciones constructoras (excluyendo `clon`) tengan tiempo de ejecución $O(1)$ en el peor caso. La estructura elegida como representación del TAD debe permitir hacer un uso eficiente del espacio de almacenamiento, contemplando una cantidad de memoria proporcional a los elementos que tenga el TAD en un momento dado. Defina la representación del TAD `T` e implemente las operaciones constructoras (excluyendo `clon`) accediendo directamente a la representación propuesta. Omita el código del resto de las operaciones del TAD (que puede asumir están implementadas). No use TADs auxiliares en la implementación del TAD `T`.

(c) Defina una función `similares` que, dados dos elementos `t1` y `t2` del TAD `T` y dado un entero positivo k (se asume $k > 0$), retorne `true` si y sólo si los primeros k elementos que se obtienen de `t1` y `t2` son iguales, y en el mismo orden. En particular, si `t1` o `t2` tienen menos de k elementos, la función debe retornar `false`. La operación `similares` no es una operación del TAD `T` por lo que en su implementación no se puede acceder a la representación de `T`. La implementación de `similares` no debe modificar los parámetros `t1` y `t2`.

*bool **similares** (T t1, T t2, int k);*

Solución:

(a) La política LIFO implica que sólo se tiene acceso al último elemento ingresado en la estructura, tanto para conocer su valor como para removerlo. El TAD `T` es entonces el TAD Pila, en este caso acotada, con sus operaciones, *crear*, *esVacia*, *apilar*, *tope* y *desapilar*, a las cuales se debe agregar *clon*.

La politica de inserción implica remover el elemento más antiguo cuando la estructura tiene M elementos. Esto debe hacerse internamente en la operación *apilar*, no agregando al TAD una operación similar a *desencolar*, lo cual correspondería a una politica FIFO.

```
/* Devuelve una estructura de tipo T sin elementos
   que puede almacenar hasta M enteros.
   Precondición: M > 0. */
T crearT(unsigned int M);

/* Inserta 'dato' en 't'.
   Si en 't' habia M elementos elimina de 't' su elemento más antiguo
   (M es el valor del parámetro pasado en crearT). */
void apilar(int dato, T & t);

/* Devuelve una copia de 't'.
   La estructura devuelta no comparte memoria con 't'. */
T clon(T t);

/* Remueve de 't' el último elemento que se agregó.
   Si esVacioT(t) no hace nada. */
void desapilar(T & t);

/* Devuelve el último elemento agregado a 't'.
   Precondicion: ! esVacioT(t). */
int tope(T t);
```

```
/* Devuelve 'true' si y sólo si en 't' no hay elementos. */
bool esVacioT(T t);

/* Libera la memoria asignada a 't'. */
void liberarT(T &t);
```

(b) La operación *apilar* debe tener tiempo $O(1)$ y debe poder acceder tanto al último elemento ingresado como al más antiguo. Además se sabe que el tamaño es acotado. Esto podría resolverse mediante un arreglo circular. Sin embargo la restricción de uso de espacio proporcional a la cantidad de elementos mantenidos no permite usar esa estructura. Por lo tanto la estructura correcta es la que habitualmente se usa para una cola no acotada: una estrucutura lineal de nodos simplemente enlazados y una cabecera con punteros al primero y al último.

```
struct nodo {
  int dato;
  nodo* sig;
};

struct cabecera {
  nodo * inicio, * final;
  unsigned int cantidad;
  unsigned int maximo;
};

typedef cabezera* T;


T crearT(unsigned int M) {
  T res = new cabecera;
  res->inicio = res->final = NULL;
  res->cantidad = 0;
  res->maximo = M;
  return res;
}

void apilar(int dato, T & t) {
  nodo * nuevo = new nodo;
  nuevo->dato = dato;
  nuevo->sig = NULL;
  //si ya hay elementos lo pongo al final de la lista
  //sino hay que actualizar el valor inicio del cabezal.
  if (esVacioT(t))
    t->inicio = nuevo;
  else  // t->final != NULL
    t->final->sig = nuevo;

  t->final = nuevo;

  //si tengo M elementos, elimino el más antiguo
  if (t->cantidad < t->maximo)
    t->cantidad++;
  else {
    nodo *liberar = t->inicio;
    t->inicio = t->inicio->sig;
    delete liberar;
  }
}
```

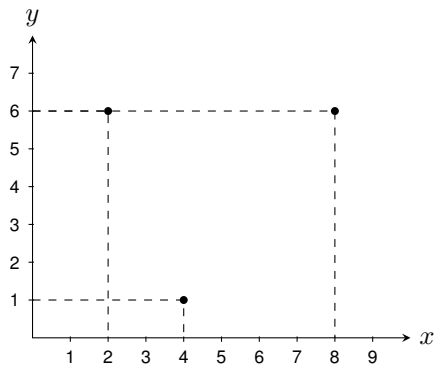
(c) Como *similares* no es parte del TAD sólo puede implementarse mediante las operaciones declaradas en la especificación. Debido a la politica LIFO sólo se pueden comparar, sin modificar las estructuras, los elementos que están en el tope de $t1$ y $t2$. Entonces, lo que se pide requiere desapilar hasta $k-1$ elementos de cada pila, lo cual modifica los parámetros. Por lo tanto deben clonarse los parámetros y proceder a hacer las comparaciones en esas copias.

```
bool similares (T t1, T t2, int k) {
  T clon1 = clon(t1);
  T clon2 = clon(t2);
  while ((k > 0) && (! esVacioT(clon1)) && (! esVacioT(clon2)) &&
                   (tope(clon1) == tope(clon2))) {
    k--;
    desapilar(clon1);
    desapilar(clon2);
  }
  liberarT(clon1);
  liberarT(clon2);
  return (k == 0);
}
```

Problema 2 (30 puntos)

Se debe implementar el TAD `Grafica` que modela mediante puntos una función parcial de enteros positivos en enteros positivos. Se considera un punto (x, y) como una asociación de una coordenada x cuyo valor asociado es la coordenada y. La cantidad máxima de asociaciones, N, que se puede mantener en un momento determinado está acotada por un parámetro definido en el constructor. Por ser una función, no puede haber dos puntos diferentes con la misma coordenada x. Se puede asumir que todos los puntos tienen la misma probabilidad de pertenecer a una gráfica. La especificación de `Grafica` se encuentra en el Listado 1.

Ejemplo:



En la gráfica representada en el diagrama hay 3 coordenadas para las cuales se mantiene una asociación. El valor asociado a 2 es 6, el valor asociado a 4 es 1 y el valor asociado a 8 es 6.
Si $N = 3$ no se podrían agregar nuevas asociaciones, aunque éstas pueden eliminarse y también actualizarse (ver las operaciones en la especificación del TAD `Grafica`).

Listado 1: Especificación de `Grafica`

```
/* Devuelve una gráfica vacia que puede mantener hasta 'N' asociaciones.
   Precondición: N > 0. */
Grafica CrearGrafica(int N);

/* Si en 'g' hay menos de 'N' asociaciones, asigna 'y' como valor asociado a 'x';
   en caso contrario la operación no tiene efecto.
   Precondición: x > 0, y > 0, Valor(x, g) == -1. */
void Asociar(int x, int y, Grafica &g);

/* Actualiza la asociación de 'x' en 'g', asignando 'y' como su valor asociado.
   Precondición: x > 0, y > 0, Valor(x, g) != -1. */
void Actualizar(int x, int y, Grafica &g);

/* Si en 'g' existe asociación para 'x', devuelve el valor asociado a 'x';
   en otro caso devuelve -1.
   Precondición: x > 0. */
int Valor(int x, Grafica g);

/* Elimina de 'g' la asociación para 'x', si existe tal asociación;
   en caso contrario la operación no tiene efecto.
   Precondición: x > 0. */
void Desasociar(int x, Grafica &g);
```

(a) Implemente el TAD `Grafica` usando dispersión abierta con `h(x) = x % N` como función *hash*. Los tiempos de ejecución de la operaciones para `Asociar` y `Actualizar` deben tener $O(1)$ en el peor caso, mientras que los de `Valor` y `Desasociar` deben ser $O(1)$ en el caso promedio. Se puede asumir que el número total de actualizaciones es menor a N.

Utilice la definición de $Punto$ y la especificación de $Lista$ de puntos (que se asume implementada) del Listado 2.

(b) Justifique brevemente por qué su implementación de `Grafica` cumple con los tiempos de ejecucion requeridos.

Listado 2: Definición de `Punto` y especificación de `Lista` de puntos

```
struct Punto { int x, y; };

/* Devuelve una lista vacia. */
Lista CrearLista();

/* Agrega `pt' al inicio de `l'. */
void Cons(Punto pt, Lista &l);

/* Devuelve el primer elemento de 'l'.
   Precondición: ! EsVaciaLista(l). */
Punto Primero(Lista l);

/* Devuelve 'l' sin el primer elemento.
   Precondición: ! EsVaciaLista(l). */
Lista Resto(Lista l);

/* Devuelve true si y sólo si
   'l' es vacia. */
bool EsVaciaLista(Lista l);

/* Devuelve true si y sólo si en 'l' hay un
   punto cuya primera coordenada es 'x'. */
bool ExisteX(int x, Lista l);

/* Remueve el primer elemento de 'l' cuya
   primera coordenada es 'x'.
   Precondición: ExisteX(x,l). */
void RemoverX(int x, Lista &l);
```

El tiempo de ejecución en el peor caso de las operaciones `CrearLista`, `Cons`, `Primero`, `Resto` y `EsVacialista` es $O(1)$; el de `ExisteX` y `RemoverX` es $O(n)$, siendo n la cantidad de elementos de la lista.

Solución:

(a) El TAD es una tabla que, como se pide, se implementa con dispersión abierta, lo cual implica mantener un arreglo de listas. En cada lista podría mantenerse más de una asociación dado que es posible que haya colisiones (aunque se espera que haya pocas como consecuencia de la igual probabilidad de los puntos). Por lo tanto para encontrar el valor asociado a una clave no se debe asumir que la asociación es el primer elemento, sino que se debe recorrer la lista. En la lista, además de las asociaciones, y como consecuencia del requerimiento de actualizar en $O(1)$ en peor caso, puede haber elementos que corresponden a asociaciones desactualizadas. Esto implica que al eliminar una asociación no alcanza con remover la primera aparción de la clave x (que es la actual asociación), sino que se debe asegurar eliminar las anteriores asociaciones. De no hacerlo, una llamada a `valor` devolvería el antiguo valor en lugar de -1.

```
struct rep_grafica {
  Lista * listas; // arreglo de listas
  int cantidad;
  int tamanio;
};

typedef rep_grafica *Grafica;

Grafica CrearGrafica(int N) {
  Grafica nuevo = new rep_grafica;
  nuevo->cantidad = 0;
  nuevo->tamanio = 2 * N;
  nuevo->listas = new Lista[2 * N];
  for (int i = 0; i < N; i++)
    nuevo->listas[i] = CrearLista();
  return nuevo;
}


void Asociar(int x, int y, Grafica &g) {
  if (g->cantidad < (g->tamanio / 2) {
    Punto pt = {x, y};
    Cons(pt, g->listas[x % g->tamanio]);
    g->cantidad++;
  }
}

void Actualizar(int x, int y, Grafica &g) {
```

```
  Punto pt = {x, y};
  Cons(pt, g->listas[x % g->tamanio]);
}

int Valor(int x, Grafica g) {
  int result;
  Lista cursor = g->listas[x % g->tamanio];
  while ((! EsVaciaLista(cursor)) && (Primero(cursor).x != x)) {
      cursor = Resto(cursor);
  }
  if (EsVaciaLista(cursor))
    result = -1;
  else
    result = Primero(cursor).y;
  return result;
}

void Desasociar(int x, Grafica &g) {
  int hash = x % g->tamanio;
  Lista l = g->listas[hash];
  if (ExisteX(x, l)) {
    g->cantidad--;
    do {
      RemoverX(x, l);
    } while ExisteX(x, l);
    // como l se pasa por referencia en RemoverX pudo ser modificada
    g->listas[hash] = l;
  }
}
```

(b) Las operaciones `Asociar` y `Actualizar` siempre se ejecutan en tiempo constante porque insertan al principio de su lista. Por lo tanto sus tiempos están en $O(1)$ en el peor caso.

Como se asume equiprobabilidad en los valores de x las asociaciones quedan uniformente distribuidas en las listas. La cota en la cantidad de asociaciones asegura que el factor de carga es menor o igual a 1. Esto no es modificado por el hecho de que cada actualización incrementa en un elemento el tamaño de alguna lista ya que se asume que la cantidad de actualizaciones es menor que N. Debido a estos hechos el tamaño de las listas es $O(1)$ en promedio.

Como la operación `Valor` hace en el peor caso una recorrida de una lista de longitud promedio $O(1)$, su tiempo de ejecución es $O(1)$ en promedio.

La operación `Desasociar` hace una llamada a `ExisteX` y una a `RemoverX` por cada elemento de la lista cuya primera coordenada sea igual a `x`, y una llamada final a `ExisteX`. Como quedo establecido, la cantidad de elementos a eliminar es $O(1)$ en promedio. Por lo tanto en promedio hay una cantidad $O(1)$ de llamadas a funciones cuyos tiempos son $O(1)$, por lo que el tiempo de ejecución de `Desasociar` también es $O(1)$ en promedio.