Rellenar estos cuadros:

Nombre:	
C.I.:	

TECNÓLOGO EN INFORMÁTICA

ESTRUCTURAS DE DATOS Y ALGORITMOS

Prototipo

2008

Ejercicio 1 (45 puntos)

Una coordenada es un tipo de datos de nombre Coord que contiene un par (X, Y) de elementos de tipo INTEGER.
El TAD Coord contiene las siguientes operaciones:

•
PROCEDURE CrearCoord (X, Y : INTEGER): Coord;
(* Crea una coordenada con el par (X, Y) *)

•
PROCEDURE CoordX (C : Coord): INTEGER;
(* Retorna el primer elemento de C. *)

•
PROCEDURE CoordY (C : Coord): INTEGER;
(* Retorna el último elemento de C. *)

•
PROCEDURE DestruirCoord (VAR C : Coord);
(* Destruye la coordenada C *)

a) Se debe especificar el TAD ConjCoord (Conjunto de coordenadas) con las operaciones usuales de conjuntos
más las siguientes operaciones:

•
Dado un entero *X,* y un conjunto de coordenadas A se debe retornar un conjunto de coordenadas con todas
las coordenadas de A que contienen a *X* como primer elemento del par.

•
Dado un entero *Y,* y un conjunto de coordenadas A se debe retornar un conjunto de coordenadas con todas
las coordenadas de A que contienen a *Y* como último elemento del par.

b) Suponga que el tipo de datos ConjCoord tiene la siguiente representación:

```
TYPE
ConjCoord = POINTER TO RECORD
c : Coord;
sig : ConjCoord
END;
```

Implemente la operación que retorna la diferencia entre dos conjuntos de coordenadas. Si usa funciones o procedimientos auxiliares, deberá implementarlos.

c) Utilizando las operaciones del TAD ConjCoord implemente la operación:
PROCEDURE CoordenadasInternas(C1, C2 : Coord; Conj : ConjCoord): ConjCoord;
(* Retorna todas las coordenadas de Conj que se encuentran dentro del rectángulo formado por los puntos

(C1.X, C1.Y), (C2.X, C1.Y), (C2.X, C2.Y), (C1.X, C2.Y). Si hay elementos en el borde del cuadrado también
deben aparecer en el conjunto resultado. *)
La implementación NO deberá iterar explícitamente sobre todas las coordenadas del rectángulo determinado por
las coordenadas C1 y C2.
Por ejemplo:
Conj
C1
C2
Resultado
[(1,2), (3,5), (1,1)]
(0,0)
(3,5)
[(1,2), (3,5), (1,1)]
[(1,-2), (3,-5), (-1,-1)]
(0,0)
(3,5)
[]
[(1,-2), (3,-5), (-1,-1)]
(1,0)
(3,-2)
[(1,-2)]

(********** Constructoras **********)
PROCEDURE **Vacio** () : ConjCoord;
(* retorna el conjunto vacío *)
PROCEDURE **Insertar** (C : Coord; A : ConjCoord) : ConjCoord;
(* retorna un nuevo conjunto que consta de todos las coordenadas de A más
la coordenada C, si es que C no era ya un elemento de A *)
(********** Predicados **********)
PROCEDURE **EsVacio** (A : ConjCoord) : BOOLEAN;
(* retorna TRUE si A es vacío *)
PROCEDURE **Pertenece** (C : Coord; A : ConjCoord) :BOOLEAN;
(* retorna TRUE si C es una coordenada de A *)
(********** Auxiliares **********)
PROCEDURE **Union** (A, B : ConjCoord) : ConjCoord;
(* retorna el conjunto de las coordenadas que pertenecen a A, o B o a ambos *)
PROCEDURE **Intersec** (A, B : ConjCoord) : ConjCoord;
(* retorna el conjunto de las coordenadas que pertenecen a A y a B *)
PROCEDURE **Dif** (A, B : ConjCoord) : ConjCoord;
(* retorna el conjunto de las coordenadas que pertenecen a A y no pertenecen a B *)
PROCEDURE **SubconjX** (X : INTEGER; A : ConjCoord) :ConjCoord;
(* retorna un conjunto de coordenadas con todas las coordenadas de A que contienen
a X como primer elemento del par *)
PROCEDURE **SubconjY** (Y : INTEGER; A : ConjCoord) :ConjCoord;
(* retorna un conjunto de coordenadas con todas las coordenadas de A que contienen
a Y como último elemento del par *)
(********** Destructoras **********)
PROCEDURE **DestruirConjCoord** (VAR A : ConjCoord);
(* Destruye el conjunto de coordenadas A *)

PROCEDURE **Vacio** () : ConjCoord;
BEGIN
RETURN NIL;
END Vacio;
PROCEDURE **EsVacio** (A : ConjCoord) : BOOLEAN;
BEGIN
RETURN A = NIL;
END EsVacio;
PROCEDURE **Pertenece** (C : Coord; A : ConjCoord) :BOOLEAN;
VAR res: BOOLEAN;
BEGIN
IF EsVacio(A) THEN
res := FALSE;
ELSE

```
res := (CoordX(A^.c) = CoordX(C)) AND (CoordY(A^.c) = CoordY(C))
OR Pertenece(C, A^.sig);
END;
RETURN res;
END Pertenece;
PROCEDURE Insertar (C : Coord; A : ConjCoord) :ConjCoord;
VAR res: ConjCoord;
BEGIN
IF NOT Pertenece(C, A) THEN
NEW(res);
res^.c := C;
res^.sig := A
ELSE
res := A
END;
RETURN res;
END Insertar;
PROCEDURE Dif (A, B : ConjCoord) : ConjCoord;
VAR
res, iterador: ConjCoord;
BEGIN
res := Vacio();
iterador := A;
WHILE NOT EsVacio(iterador) DO
IF NOT Pertenece(iterador^.c, B) THEN
res := Insertar(iterador^.c, res);
END;
Iterador := iterador^.sig;
END;
RETURN res;
END Dif;
```

```
PROCEDURE CoordenadasInternas(C1, C2 : Coord; Conj : ConjCoord): ConjCoord;
VAR
i, incremento: INTEGER;
res1, res2: ConjCoord;
BEGIN
(* Se calcula en res1 el conjunto de coordenadas con X dentro del rango *)
incremento := (CoordX(C2) - CoordX(C1)) / ABS(CoordX(C2) - CoordX(C1));
i := CoordX(C1)-incremento;
res1 := Vacio();
REPEAT
i := i + incremento;
res1 := Union(res1, SubconjX(i, Conj));
UNTIL i = CoordX(C2);
(* Se calcula en res2 el subconjunto de res1 con Y dentro del rango *)
incremento := (CoordY(C2) - CoordY(C1)) / ABS(CoordY(C2) - CoordY(C1));
i := CoordY(C1)-incremento;
res2 := Vacio();
REPEAT
i := i + incremento;
res2 := Union(res2, SubconjY(i, res1));
UNTIL i = CoordY(C2);
RETURN res2;
END CoordenadasInternas;
```

Ejercicio 2 (30 puntos)

Se considera el tipo abstracto SetNat que representa conjuntos de números naturales. Las operaciones de este TAD son:

ConjVacio //Retorna el conjunto vacío.

Agregar //Agrega un elemento a un conjunto.

Union //Retorna la unión de 2 conjuntos.

Interseccion //Retorna la intersección de 2 conjuntos.

Inclusión //Recibe 2 conjuntos y retorna TRUE, si y solo si el primer conjunto está incluido en el segundo.

Diferencia //Retorna la diferencia de 2 conjuntos. La diferencia (A - B) se define como el conjunto de todos los elementos que estan en A pero no estan en B.

Pertenencia //Dado un natural y un conjunto, retorna TRUE si el natural pertenece al conjunto.

Maximo //Retorna el máximo de un conjunto.

Minimo //Retorna el mínimo de un conjunto.

EsVacio //Retorna verdadero TRUE si y solo si el conjunto es vacío.

a) Escribir un módulo de definición del TAD SetNat con las operaciones descriptas arriba. La especificación debe ser
puramente funcional.
b) Escribir un módulo de implementación correspondiente al módulo de definición dado en la parte a).

Solución:

Solución Parte a

```
typedef  NodoSetNat* SetNat;


//Retorna el conjunto vacío.
SetNat ConjVacio();

//Agrega un elemento a un conjunto.
//Pre: !Pertenencia(n,s)
SetNat Agregar(int n, SetNat & s);

//Retorna la unión de 2 conjuntos.
SetNat Union(SetNat s, SetNat t);

//Retorna la intersección de 2 conjuntos.
SetNat Interseccion( SetNat s , SetNat t);

//Recibe 2 conjuntos y retorna TRUE, si y solo si el primer conjunto está incluido en el
segundo.
bool Inclusión( SetNat s , SetNat t);

//Retorna la diferencia de 2 conjuntos. La diferencia (A - B) se define como el conjunto
de todos los elementos que estan en A pero no estan en B.
SetNat Diferencia( SetNat s , SetNat t);

//Dado un natural y un conjunto, retorna TRUE si el natural pertenece al conjunto.
```

```
bool Pertenencia(int n, SetNat s);

//Retorna el máximo de un conjunto.
//Pre: !EsVacio(s)
int Maximo(SetNat s);

//Retorna el mínimo de un conjunto.
//Pre: !EsVacio(s)
int Minimo(SetNat s);

//Retorna verdadero TRUE si y solo si el conjunto es vacío.
bool EsVacio(SetNat s);
```

Solución Parte b

```
typedef  struct NodoSet{
        int n;
        SetNat sig;
}

SetNat ConjVacio(){
        return NULL;
}

void Agregar(int n, SetNat & s){
        SetNat nuevo= new NodoSetNat;
        nuevo->n = n;
        nuevo -> sig = s;
        s=nuevo;
}

SetNat Union(SetNat s, SetNat t){
        SetNat nuevo=ConjVacio();
        while (!EsVacio(s) ){
                if (!Pertenencia(s->n, nuevo) )
                        Agregar(s->n, nuevo);
                s = s->sig;
        }
        while (!EsVacio(t) ){
                if (!Pertenencia(t->n, nuevo) )
                        Agregar(t->n, nuevo);
                t = t->sig;
        }
        Return nuevo;
}


SetNat Interseccion( SetNat s , SetNat t){
        SetNat nuevo = ConjVacio();
        while (!EsVacio(s) ){
```

```
                if ( Pertenencia(s->n, t)
                        Agregar(s->n, nuevo);
                s = s->sig;
        }
        return nuevo;
}

bool Inclusión( SetNat s , SetNat t){
        bool incluido = true;
        while (!EsVacio(s) ){
                if ( !Pertenencia(s->n, t) )
                        incluido=false;
                s = s->sig;
        }
        return incluido;
}


SetNat Diferencia( SetNat s , SetNat t) {
        SetNat nuevo=ConjVacio();
        while (!EsVacio(s) ){
                if ( !Pertenencia(s->n, t)
                        Agregar(s->n, nuevo);
                s = s->sig;
        }
        return nuevo;
}


bool Pertenencia(int n, SetNat s){
        if(EsVacio(s) )
                return false;
        else if( s->n = = n)
                return true;
        else
                return Pertenencia(n, s->sig);
}

int Maximo(SetNat s){
        int max= s->n;
        while ( ! EsVacio(s) ){
                if( s->n > max)
                        max=s->n;
                s = s->sig;
        }
        return max;
}

int Minimo(SetNat s){
        int min= s->n;
```

```
        while ( ! EsVacio(s) ){
                if( s->n < min)
                        min=s->n;
                s = s->sig;
        }
        return min;
}


bool EsVacio(SetNat s){
        return s = =NULL;
}
```

Ejercicio 3 (25 puntos)

a) Especifique en C/C++ el TAD PILA de elementos de un tipo genérico que permita almacenar a lo sumo MAX elementos. Se pide considerar
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 MAX (a lo sumo) se
consideran.
b) Implemente en C/C++ el TAD especificado en la parte a) en la que las operaciones constructoras, selectoras y predicados se realicen sin recorrer la estructura.

Solución

a)

```
#ifndef PILA_H
#define PILA_H
#include "generico.h"

#define max ...

typedef struct Nodo* Pila;
/*** constructoras ***/
/* construye una estructura (pila) vacia */
Pila vacia();
/* agrega el elemento e a la estructura */
/* si la estrucutura ya contiene K elementos, se elimina el mas viejo
*/
Pila insertar ( generico e, Pila P );
/*** selectoras ***/
/* devuelve el elemento que ha sido insertado en ultima instancia */
/* pre: NOT esta_vacia(P) */
generico tope ( Pila P );
/* devuelve la estructura sin el elemento mas nuevo */
/* pre: NOT esta_vacia(P) */
Pila quitar ( Pila P );
/*** predicados ***/
/* devuelve true sii la estructura esta vacia */
bool esta_vacia ( Pila p );
```

```
/* devuelve true sii la estructura cuenta con K elementos */
bool llena ( Pila p );
#endif
```

b)

```
#include "Pila.h"
struct Nodo{
  short tope, base, cant;
  generico vector[max];
};

Pila vacia(){
Pila p=NEW(nodo);
p->base = 0;
p->tope = MAX-1;
p->cant = 0;
return p;
}



/* agrega el elemento e a la estructura */
/* si la estrucutura ya contiene K elementos, se elimina el mas viejo
*/
Pila insertar ( generico e, Pila P){
if (P->cant < k){
P->tope = (P->tope + 1) % MAX;
P->vector[P->tope] = e;
P->cant = P->cant + 1
}else{
P->tope = P->base;
P->base = (P->base + 1) % MAX;
P->vector[P->tope] = e;
}}

 /*** selectoras ***/
/* devuelve el elemento que ha sido insertado en ultima instancia */
/* pre: NOT esta_vacia(P) */
generico tope ( Pila P){
  return P->vector[P->tope];
}

/* devuelve la estructura sin el elemento mas nuevo */
/* pre: NOT esta_vacia(P) */
Pila quitar ( Pila P){
P->cant = P->cant - 1;
P->tope = (P->tope - 1) % MAX;
}
/*** predicados ***/
/* devuelve true sii la estructura esta vacia */
bool esta_vacia ( Pila P){
  return P->cant == 0;
}

 /* devuelve true sii la estructura cuenta con K elementos */
bool tiene_k (Pila P ){
 return P->cant == MAX;
}
```