Guia 5

May 2024

1. Ejercicio 1

- Sin Repetidos:
- `pred invRep` (c:ConjAcotadoArr< T >) {

  $0 \leq c.largo \leq c.datos.length \ \wedge \ (\forall i, j : \mathbb{Z}) \ (0 \leq i, j < c.length \ \wedge \ i \neq j \longrightarrow_L c.datos[i] \neq c.datos[j])$

  }
- `pred Abs` (c:ConjAcotadoArr< T >, c":ConjAcotado< T >) {

  $c''.cap = c.datos.length \ \wedge \ (\forall t : T) \ (t \in c''.elems \iff t \in subseq(c.datos, 0, largo))$

  }
- `proc agregar` (inout c:ConjAcotadoArr< T >, in e:T)

```
1 c.datos[c.largo] := e
2 c.largo := c.largo + 1
```

- `proc sacar` (inout c:ConjAcotadoArr< T >, in e:T)

```
1 i := 0
2 while (c.datos[i] /= e) do
3         i := i + 1
4 endwhile
5 c.datos[i] := c.datos[c.largo − 1]
6 c.largo := c.largo − 1
```

- Con repetidos:
- `pred invRep` (c:ConjAcotadoArr< T >) {

  $0 \leq c.largo \leq c.datos.length$

  }
- `pred Abs` (c:ConjAcotadoArr< T >, c":ConjAcotado< T >) {

  $c''.cap = c.datos.length \ \wedge \ (\forall t : T) \ (t \in c''.elems \iff t \in subseq(c.datos, 0, largo))$

  }
- Mi idea en agregar con repetidos es guardar el repetido en la ultima posicion e ir reemplazandolo por los siguientes repetidos que quiera agregar.En caso de querer agregar un no repetido hago lo mismo que con sin repetidos.

- proc agregar (inout c:ConjAcotadoArr< T >, in e:T)

```
1  res := false
2  i := 0
3  while (i < c.largo) do
4          if (c.datos[i] = e) do
5              res := true
6          else
7              skip
8          endif
9          i := i + 1
10 endwhile
11 if (res == true) do
12     c.datos[c.length − 1] := e
13 else
14     c.datos[c.largo] := e
15     c.largo = c.largo + 1
16 endif
```

- proc sacar (inout c:ConjAcotadoArr< T >, in e:T)

```
1  i := 0
2  while (c.datos[i] /= e) do
3          i := i + 1
4  endwhile
5  c.datos[i] := c.datos[c.largo − 1]
6  c.largo := c.largo − 1
```

2. Ejercicio 2

Modulo PilaArr< T > implementa Pila< T >
elems : Array< T >
largo : $\mathbb{Z}$

pred invRep (p:PilaArr< T >) {
$0 \leq p.largo \leq p.elems.length$
}
pred Abs (p:PilaArr< T >, p":Pila< T >) {
$p.largo = |p''.s| \ \wedge \ (\forall i : \mathbb{Z})\ (0 \leq i < |p''.s| \longrightarrow_L p''.s[i] = subseq(p.elems, 0, largo)[i])$
}

proc apilar (inout p:PilaArr< T >, in e:T)

```
1  if (p.largo < p.elems.length) do
2      p.elems[p.largo] := e
3      p.largo := p.largo + 1
4  else
5      a := new Array<T>(p.length)
6      a := p.elems
7      i := 0
8      p.elems := new Array<T>(p.length∗2)
```

```
9       while (i < a.length) do
10              p.elems[i] := a[i]
11              i := i+1
12      endwhile
13      p.elems[largo] := e
14      p.largo := p.largo + 1
15  endif
```

`proc desapilar` (inout p:PilaArr< T >) : T

```
1  res := p.elems[p.largo − 1]
2  p.largo := p.largo − 1
3  return res
```

3. Ejercicio 3

Modulo PilaConElimArr< T > implementa PilaConElim< T >
elems : Array< T >
largo : $\mathbb{Z}$ $\longrightarrow_L$ me va a indicar en donde esta el tope de la pila
enUso : Array< $Bool$ >

`pred invRep` (p:PilaConElimArr) {
$|p.elems| = |p.enUso| \ \wedge \ 0 \leq p.largo \leq |p.elems|$
}
`pred Abs` (p:PilaConElimArr< T >, p":PilaConElim< T >) {
$mismaCantidad(|p''.s|, p.enUso) \ \wedge \ mismoOrden(p'', p)$
}
`pred mismaCantidad` (len:$\mathbb{Z}$,p.enUso:Array< $Bool$ >) {
$len = \sum_{i=0}^{|p.enUso|-1}$ if $p.enUso = true$ then 1 else 0 fi
}
`pred mismoOrden` (p":PilaConElim< T >, p:PilaConELimArr< T >) {
$(\forall i : \mathbb{Z}) \ (0 \leq i < |p''.s| \longrightarrow_L ((\exists j : \mathbb{Z}) \ (j \geq i) \ \wedge \ ((\forall k : \mathbb{Z}) \ (i \leq k < j \longrightarrow_L p.enUso[k] = false) \ \wedge$
$p.enUso[j] = true \ \wedge \ p.elems[j] = p''.s[i]))$
}

`proc apilar` (inout p:PilaConElimArr< T >, e:T)

```
1  if (p.largo < p.elems.length) do
2      p.elems[p.largo] := e
3      p.enUso[p.largo] = true
4      p.largo := p.largo + 1
5
6  else
7      a := new Array<T>(p.elems.length)
8      a := p.elems
9      a'' := new Array<Bool>(p.enUso.length)
10     a'' := p.enUso
11     i := 0
12     p.elems := new Array<T>(p.elems.length∗2)
13     p.enUso := new Array<Bool>(p.enUso.length∗2)
```

```
14      while (i < a.length) do
15              p.elems[i] := a[i]
16              p.enUso[i] := a''[i]
17              i := i+1
18      endwhile
19      p.elems[largo] := e
20      p.enUso[p.largo] := true
21      p.largo := p.largo + 1
22
23  endif
```

`proc desapilar` (inout p:PilaConElimArr< T >) : T

```
1  res := p.elems[p.largo − 1]
2  p.enUso[p.largo − 1] := false
3  p.largo := p.largo − 1
4  i := p.largo
5  while (p.elems[i] /= true && i >= 0) do
6          i := i − 1
7  endwhile
8  p.largo := i + 1
```

`proc eliminar` (inout p:PilaConElimArr< T >, in e:T)

```
1  i := 0
2  while (p.elems[i] /= e) do
3          i := i + 1
4  endwhile
5  p.enUso[i] := false
```

4. Ejercicio 4

Modulo ColaAcotadaImpl< T > implementa ColaAcotada< T >
datos : Array< T >
inicio : $\mathbb{Z}$
fin : $\mathbb{Z}$

`pred invRep` (c:ColaAcotadaImpl< T >) {
$0 \leq c.inicio, c.fin < c.elems.length$
}
`pred Abs` (c:ColaAcotadaImpl< T >, c":ColaAcotada< T >) {
$c''.cap = |c.datos| \wedge (inicioAfin(c.datos, c.inicio, c.fin, c''.s) \vee finAinicio((c.datos, c.inicio, c.fin, c''.s))$
}
`pred inicioAfin` (c.datos:Array< T >,inicio,fin:$\mathbb{Z}$,s:$seq\langle T\rangle$) {
$inicio \leq fin \ \wedge \ (\forall i : \mathbb{Z})\ (0 \leq i < |s| \longrightarrow_L s[i] = subseq(c.datos, inicio, fin)[i])$
}
`pred finAinicio` (c.datos:Array< T >,inicio,fin:$\mathbb{Z}$,s:$seq\langle T\rangle$) {
$inicio > fin \ \wedge \ (\forall i : \mathbb{Z})\ (0 \leq i < |s| \longrightarrow_L$
$s[i] = concat(subseq(c.datos, inicio, |c.datos|), subseq(c.datos, 0, fin))[i])$

}

`proc encolar` (inout c:ColaAcotadaImpl, in e:T)

```
1 | c.datos[c.fin] := e
2 | c.fin := (c.fin + 1) mod c.datos.length
```

`proc desencolar` (inout c:ColaAcotadaImpl) : T

```
1 | res := c.datos[c.inicio]
2 | c.inicio := (c.inicio + 1) mod c.datos.length
3 | return res
```

- No tiene sentido usarlo en una pila por que inicio y fin apuntarian siempre al mismo espacio.

5. Ejercicio 5

- Voy a intentar implementar un conjnto de tuplas $< (String, \mathbb{Z}) >$ donde string es el nombre de un alumno e int es su lu (libreta universitaria) usando un conjunto acotado, simulando la cantidad de alumnos que puedo tener en una clase.
- Voy a usar dos indices, uno para ordenarlos alfabeticamente y otro para ordenarlos por lu.
- Uso conjuntos por que no quiero tener repetidos ya que nombre y lu son datos unicos , no puedo tener 2 o mas alumnos con el mismo nombre y la misma libreta.

Modulo Clase$< (String, \mathbb{Z}) >$ implementa ConjAcotado$< (String, \mathbb{Z}) >$
datos :Array$< (String, \mathbb{Z}) >$
lu : Array$< \mathbb{Z} >$
nombre : Array$< \mathbb{Z} >$
espacio : $\mathbb{Z}$

- invRep: Para que una instancia de clase se considere valida , la longitud de datos debe coincidir con la longitud en indices nombre y lu.
- Ademas los indices guardados en nombre y lu deben estar en rango de datos y deben ser distintos entre si ya que apuntaran a posiciones distintas en el arreglo datos.
- Espacio no debe superar la longitud del arreglo datos y los 3 arreglos tienen la misma cantidad de elementos que dice espacio.
- Por ultimo lu y nombre deben estar ordenados de menor a mayor segun mi implementacion.
- `pred invRep` (c:Clase$< String, \mathbb{Z}) >$) {

  $|c.datos| = |c.lu| = |c.nombre| \land (\forall i : \mathbb{Z})\ (i \in c.lu \land i \in c.nombre \iff 0 \leq i < c.espacio) \land todosDistintos(c.lu, c.datos, c.nombre, c.espacio) \land 0 \leq c.espacio \leq |c.datos| \land estanOrdenados(c.nombre, c.lu)$

  }
- `pred todosDistintos` (a1,a2,a3:Array$< T >$,espacio:$\mathbb{Z}$) {

  $(\forall i, j : \mathbb{Z})\ (0 \leq i, j < espacio \longrightarrow_L a1[i] \neq a1[j] \ \land \ a2[i] \neq a2[j] \ \land \ a3[i] \neq a3[j])$

  }
- `pred estanOrdenados` (a1,a2:Array$< T >$) {

$(\forall i : \mathbb{Z})\ (0 \leq i < |a1| - 1 \longrightarrow_L a1[i] \leq a1[i+1]\ \wedge\ a2[i] \leq a2[i+1])$

}

- Abs:Para que una instancia de Clase se corresponda con una instancia del TAD conjuntoAcotado , la capacidad del TAD debe ser igual a la longitud de los datos de Clase y ambos deben tener los mismos elementos , como es un conjunto no importa el orden.

- `pred Abs` (c:Clase$< (String, \mathbb{Z}) >$,c":ConjAcotado$< (String, \mathbb{Z}) >$) {

$c''.cap = c.datos.length\ \wedge\ (\forall s : (String, \mathbb{Z}))\ (s \in c''.elems\ \iff\ s \in c.datos)$

}

- `proc buscarPorPrimera` (in c:Clase$< (String, \mathbb{Z}) >$, e:String) : (String,$\mathbb{Z}$)

```
\\Algoritmo de busqueda binaria.
inicio := 0
fin := c.datos.length - 1
while (inicio <= fin) do
      medio = (inicio + fin) // 2 --> "//" es la division entera
      if (c.datos[c.nombre[medio]][0] == e)
          return c.datos[medio]
      else
          if (c.datos[c.nombre[medio]][0] < e) do
              inicio := medio + 1
          else
              fin := medio - 1
          endif
      endif
endwhile
res := ("No esta",-1)
return res;
```

- `proc buscarPorSegunda` (in c:Clase$< (String, \mathbb{Z}) >$, e:$\mathbb{Z}$) : (String,$\mathbb{Z}$)

```
inicio := 0
fin := c.datos.length - 1
while (inicio <= fin) do
      medio = (inicio + fin) // 2
      if (c.datos[c.lu[medio]][1] == e)
          return c.datos[medio]
      else
          if (c.datos[c.lu[medio]][1] < e) do
              inicio := medio + 1
          else
              fin := medio - 1
          endif
      endif
endwhile
res := ("No esta",-1)
return res;
```

- `proc agregar` (inout c:Clase$< (String, \mathbb{Z}) >$, in e :(String,$\mathbb{Z}$))

```
\\Inicialmente lo agrego al final de todos los arreglos.
c.datos[largo] := e
c.nombre[largo] := e[0]
c.lu[largo] := e[1]
largo := largo + 1

\\Aca lo ordeno en lu y nombre.
primer := e[0]
segundo := e[1]
i := largo - 2

while (i >= 0 && primer < c.datos[c.nombre[i]][0]) do
        c.nombre[i+1] := c.nombre[i]
        c.nombre[i] := primer
        i := i - 1
endwhile

while (i >= 0 && primer < c.datos[c.nombre[i]][1]) do
        c.lu[i+1] := c.lu[i]
        c.lu[i] := segundo
        i := i-1
endwhile
```

- `proc sacar` (inout c:Clase< $(String, \mathbb{Z})$ >, in e:(String,$\mathbb{Z}$))

```
\\Busco que posicion esta el que quiero sacar.
\\Podria usar busqueda binaria pero para simplificar lo busco asi lol.
i := 0
while (c.datos[i] != e) do
       i := i + 1
endwhile

\\Reemplazo esa posicion por el ultimo y disminuyo largo.
c.datos[i] := c.datos[largo - 1]
c.largo := c.largo - 1

\\Ahora lo saco de lu y nombre.
\\Busco la posicion del elemento que quiero sacar.
j := 0
while(c.datos[c.nombre[j]] != e[0]) do
      j := j + 1
endwhile

\\Reemplazo esa posicion por el ultimo del arreglo.
c.nombre[j] := c.nombre[c.largo]

\\Con un ciclo vuelvo a mover todos los elementos para que esten ordenados.
k := j
while (k < largo) do
        c.nombre[k] := c.nombre[k+1]
        c.nombre[k+1] := c.nombre[largo+1]
```

```
27        k := k + 1
28  endwhile
29
30  \\idem pero con la lista lu.
31  m := 0
32  while(c.datos[c.nombre[m]] != e[1]) do
33        m := m + 1
34  endwhile
35
36  c.lu[m] := c.lu[c.largo]
37  n := m
38  while (n < largo) do
39        c.lu[n] := c.lu[n+1]
40        c.lu[n+1] := c.lu[largo+1]
41        k := k + 1
42  endwhile
```

5.1. Ejercicio sin numero

- Todos los productos en venta deben estar en ultimoPrecio y totalPorProducto.
- ultimoPrecio debe ser el monto asociado a la ultima fecha.
- totalPorProducto debe verificar la suma de montos por producto en venta.
- `pred invRep` (c:ComercioImpl) {

  $esUltimoPrecio(c.ultimoprecio, c.ventas) \wedge esTotalPorProducto(c.ventas, c.totalPorProducto \wedge (\forall p : String)\ (p \in c.ultimoPrecio \wedge p \in c.totalPorProducto \iff (\exists i, j : \mathbb{Z})\ ((p, i, j) \in c.ventas))$

  }
- `pred esUltimoPrecio` (u:dictImpl$< String, \mathbb{Z} >$, v:$seq\langle(string, \mathbb{Z}, \mathbb{Z})\rangle$) {

  $(\forall p : String)\ (p \in u \longrightarrow_L (\exists i : \mathbb{Z})\ (0 \leq i < |v| \wedge v[i]_0 = p \wedge u[p] \geq v[i]_1))$

  }
- `pred esTotalPorProducto` (v:$seq\langle(string, \mathbb{Z}, \mathbb{Z})\rangle$, t:dictImpl$< String, \mathbb{Z} >$) {

  $(\forall p : String)\ (p \in t \longrightarrow_L t[p] = \sum_{i=0}^{|v|-1} \text{if } v[i]_0 = p \text{ then } v[i]_2 \text{ else } 0 \text{ fi})$

  }
- En la funcion de abstraccion debo asegurarme que todos los productos de Comercio esten en ComercioImpl con su correspondiente fecha y monto , el orden puede ser distinto pero todos deben estar.
- `pred Abs` (c:ComercioImpl,c":Comercio) {

  $(\forall p : Producto)\ (p \in c''.ventasPorProducto \longrightarrow_L ((\forall i : \mathbb{Z})\ (0 \leq i < c''.ventasPorProducto[p] \longrightarrow_L$

  $(p, c''.ventasPorProducto[i]_0, c''.ventasPorProducto[i]_1) \in c.ventas)))$

  }

6. Ejercicio 6

- En el invariante me aseguro que valga la doble implicacion , si alarma esta en Alarmas entonces existe un sensor en Sensores tal que alarma tenga asociado ese sensor , y ese sensor tenga a alarma en su conjunto.
- `pred invRep` (p:PlantaImpl) {

  $(\forall a : Alarma)\ (a \in p.alarmas \iff (\exists s : Sensor)\ (s \in p.sensores \ \wedge \ s \in p.alarmas[a] \ \wedge \ a \in p.Sensores[s]))$

  }
- En la funcion de abstraccion me aseguro que todas las alarmas y sensores de PlantaImpl esten en el TAD Planta.
- `pred Abs` (p:PlantaImpl, p":Planta) {

  $(\forall a : A)\ (a \in p.alarmas \iff (\exists s : Sensor)\ (s \in p.Sensores \ \wedge \ (s, a) \in p''.sensores \ \wedge \ a \in p''.alarmas))$

  }