Ejercicio 1

```
proc agregarAtras (inout l:listaEnlazada<T>, in t: T):  {
    nodo := new NodoLista<T>
    nodo.valor := t
    nodo.siguiente := null
    if (l.longitud==0) then
        l.primero := nodo
        l.ultimo := nodo
    else
        l.ultimo.siguiente := nodo
        l.ultimo := nodo
    endif
    l.longitud++        }

proc obtener (in l: listaEnlazada<T>, in i: ℤ): T {
    j := 0
    while (j < i) do
        j++
    endwhile
    return l[j]
}

proc concatenar (inout l1: listaEnlazada<T>, in l2: listaEnlazada<T>):  {
    if (l1.longitud! = 0 and l2 == 0) then
        l1 := l1
    ElseIf (l1.longitud == 0 and l2.longitud! = 0) then
        l1.primero := l2.primero
        l1.ultimo := l2.ultimo
        puntero := l2.primero.siguiente
        while (puntero!=null) do
            agregarAtras(l1,puntero)
            puntero := puntero.siguiente
        endwhile
    else
        l1.ultimo := l2.ultimo
        puntero := l2.primero
        while (puntero!=null) do
            agregarAtras(l1,puntero)
            puntero := puntero.siguiente
        endwhile
    endif
    l1.longitud := l1.longitud+l2.longitud
}
```

$agregarAtras \in O(1)$
$obtener \in O(n)$
$concatenar \in O(n)$

El invariante de representacion debe indicar que el largo de la secuencia se corresponde con la cantidad de elementos que tiene la misma y que 'primero' y 'ultimo' pertenecen a la secuencia (esta mal)

Ejercicio 2

`pred` invRep (c:$conjArr < T >$){$c.tamano \geq c.datos.length \land noRepetidos(c)$}

`pred` noRepetidos (c:$conjArr < T >$){$(\forall i : \mathbb{Z})(0 \leq i < c.tamano \rightarrow \neg(\exists j : \mathbb{Z})(0 \leq j < c.tamano \land j \neq i \land c[j] = c[i])$}

`pred` abs (c:$conjArr < T >$, c': $ConjuntoAcotado < T >$){$c.tamano = c'.capacidad \land (\forall i : \mathbb{Z})(0 \leq i \leq c.tamano \rightarrow c[i] \in c'.elems)$}

Ejercicio 3

`pred` invRep (c: $conjuntoLista < T >$){$c.tamano = c.datos.longitud$}

`pred` abs (c: $conjuntoLista < T >$, c': $conjunto < T >$){$c.tamano = |c'.elems| \land (\forall e : T)(e \in c.datos \leftrightarrow e \in c'.elems)$}

`modulo` $conjuntoLista < T >$ `implementa` $conjunto < T >$ {

```
proc conjVacio (): ConjuntoLista<T> {
    res.datos := new listavacia()
    res.tamaño := 0
    return res
}

proc pertenece (in c:conjuntoLista<T>, in e:T): Bool {
    res := False
    puntero := c.datos.cabeza
    while (puntero! = null ∧ res = False) do
        if (puntero.val==e) then
            res := True
        else
            puntero := puntero.siguiente
        endif
    endwhile
    return res
}
```

```
proc agregar (inout c:conjuntoLista<T>,in e:T):  {
    if (pertenece(c,e)) then
        newUlt := new nodoLista<T>
        newUlt.sig := null
        newUlt.ant := c.datos.ultimo
        c.datos.ultimo.siguiente := newUlt
        c.datos.ultimo := newUlt
    endif
}
```

(`agregarRapido` asume que el elemento se encuentra en la lista)

```
proc agregarRapido (inout c:conjuntoLista<T>,in e:T):  {
    newUlt.sig := null
    newUlt := new nodoLista<T>
    newUlt.ant := c.datos.ultimo
    c.datos.ultimo.siguiente := newUlt
    c.datos.ultimo := newUlt
}

proc unir (inout c1:conjuntoLista<T>,in c2:conjuntoLista<T>):  {
    if (c2.tamaño!=0 and c1.tamaño!=0) then
        puntero := c2.datos.cabeza
        c1.ultimo.siguiente := puntero
        while (puntero!=null) do
            if (pertenece(c1, puntero) == False) then
                agregar(c1,puntero)
                if (puntero.siguiente == null) then
                    c1.datos.ultimo := puntero
                endif
            endif
            puntero := puntero.siguiente
        endwhile
    ElseIf (c1.tamaño=0) then
        c1 := c2
    endif
}
```

}

- $conjVacio \in O(1)$
- $pertenece \in O(n)$
- $agregar \in O(n)$
- $agregarRapido \in O(1)$
- $unir \in O(n^2)$

Ejercicio 4

`modulo` *indice* `implementa` $Conjunto\langle tupla\langle \mathbb{Z}, \mathbb{Z}, \mathbb{Z}\rangle\rangle$ {

`var` data : $array\langle tupla\langle \mathbb{Z}, \mathbb{Z}, \mathbb{Z}\rangle\rangle$

`var` indices : $array\langle array\langle \mathbb{Z}\rangle\rangle$

}

`pred` invRep (i:indice){$(\forall a : array\langle \mathbb{Z}\rangle)(a \in i.indices \rightarrow a.length = i.data.longitud) \wedge$
$(\forall j : \mathbb{Z})(0 \leq j \leq 2 \rightarrow_L (\forall k : \mathbb{Z})(1 \leq k < i[j].length \rightarrow_L i.data[k]_j \geq i.data[k-1]_j)) \wedge i.indices.length = 3$}

`pred` abs (i: indice,c': $Conjunto\langle tupla\langle \mathbb{Z}, \mathbb{Z}, \mathbb{Z}\rangle\rangle$){$(\forall e : tupla\langle \mathbb{Z}, \mathbb{Z}, \mathbb{Z}\rangle)(e \in i.data \leftrightarrow e \in c.elems)$}

`proc` buscarPor ($i : indice, comp : \mathbb{Z}, t : tupla\langle \mathbb{Z}, \mathbb{Z}, \mathbb{Z}\rangle$): Bool {
    res := false
    int j := 0
    pos := i.indices[comp][j]
    `while` ($j < i.data.length$) `do`
        `if` ($t == i.data[pos]$) `then`
            res := true
        `endif`
    return res
}

`proc` agregar (*inout* $i : indice$, *in* $t : tupla\langle \mathbb{Z}, \mathbb{Z}, \mathbb{Z}\rangle$, *in* $comp : \mathbb{Z}$): {
    `if` (buscarPor(i,t,comp)) `then`
        i.data.length++
        i.data[i.data.length-1] := t
        int j := 0
        `while` ($j < 3$) `do`
            agregarAIndice(i,j,t)
        `endwhile`
}

`proc` agregarAIndice (*inout* $i : indice$, *in* $j : \mathbb{Z}$, *in* $t : tupla\langle \mathbb{Z}, \mathbb{Z}, \mathbb{Z}\rangle$, *in* $comp : \mathbb{Z}$): {
    j := 0
    newIndice := **new** $array\langle \mathbb{Z}\rangle(i.data.length)$
    noSeAgrego := True
    `while` ($j < i.data.length - 1$) `do`
        pos := i.indices[comp][j]
        `if` ($i.data[pos]_{comp} \geq t_{comp}$ and noSeAgrego) `then`
            newArray[j] := i.data.length-1
            noSeAgrego := False
            newArray[j+1] := pos
        `else`
            newArray[j] := pos
        `endif`
    `endwhile`

```
    if (noSeAgrego) then
        newArray[i.data.length-1] := i.data.length-1
    endif
    i.indices[comp] := newIndice
}

proc sacar (inout i : indice, in j : ℤ, in t : tupla⟨ℤ, ℤ, ℤ⟩, in comp : ℤ):  {
    if (buscarPor(i,t,comp)) then
    newData := newarray⟨tupla⟨ℤ, ℤ, ℤ⟩⟩(i.data.length − 1)
    j := 0
    while (j < i.data.length) do
        if (i.data[j]!=t) then
            newArray[j] := i.data[j]
        else
            posTupla := j
        endif
    endwhile
    k := 0
    newindex0 := new array⟨ℤ⟩(i.data.length − 1)
    newindex1 := new array⟨ℤ⟩(i.data.length − 1)
    newindex2 := new array⟨ℤ⟩(i.data.length − 1)
    while (k < i.data.length) do
        if (i.indice[k]!=posTupla) then
            newindex0[k] := i.indice[0][k]
            newindex1[k] := i.indice[1][k]
            newindex2[k] := i.indice[2][k]
        endif
    endwhile
    i.indice[0] := newindex0
    i.indice[1] := newindex1
    i.indice[2] := newindex2
    i.data := newData
}
```

Ejercicio 5

```
modulo bufferCircular<T> implementa Cola<T> {
        var data : array<T>
        var f : int
        var i : int
        var size : int

    proc encolar (inout b:bufferCircular<T>, in elem: T): {
        if (b.f! = −1 and ((b.i > b.f and b.i − b.f > 1) or (b.i < b.f and b.size − b.f > 0))) then
            b.data[f] := elem
            b.f := b.f+1
        ElseIf (b.f==-1) then
            b.data[b.i] := elem
            b.f := b.i
        endif
    }

    proc desencolar (inout b:bufferCircular<T>): T {
        res := null
        if (i=f) then
            res := b.data[i-1]
            b.data[i-1] := null
            f := -1
        ElseIf (i!=f) then
            res := b.data[f-1]
            b.data[f-1] := null
            if (f==-1) then
                f := b.size
            else
                f := f-1
            endif
        endif
        return res
    }

}
```

pred invRep (b:$bufferCircular<T>$){$0 \leq i \leq b.size \land -1 \leq f \leq b.size \land size = b.data.length$}

pred abs (b:$bufferCircular<T>$,b':$ColaAcotada<T>$){$cantElems(b) = b'.s.length \land$
$(i > f \leftrightarrow b'.s = subseq(b.data, b.i - 1, b.size - 1) + +subseq(b.data, 0, b.f - 1)) \land$
$(i \leq f \leftrightarrow b'.s = subseq(b.data, b.i - 1, b.f - 1)) \land (f = -1 \leftrightarrow b'.s = \langle\rangle)$}

No tiene sentido usar un buffer circular para una pila ya que siempre se agregan o se eliminan elementos en la ultima posicion, por lo tanto no tendria ninguna utilidad reciclar espacio con un buffer ya que, el funcionamiento en si mismo de la pila permite reciclar la ultima posicion.

Ejercicio 6

Diccionario sobre listaEnlazada

`Pares` es $struct < key : K, val : V >$

```
modulo dict<K,V> implementa Diccionario<K,V> {
        var data : listaEnlazada<Pares>

    proc diccionarioVacio (): dict<K,V> {
        res.data.listavacia()
        return res
    }

    proc esta (d:dict<K,V>,clave:K): Bool {
        res := False
        it := d.data.iterador()
        while (it.haySiguiente()) do
            if (it.siguiente().key == k and res! = True) then
                res := True
            endif
        endwhile
        return res
    }

    proc definir (inout d: dict<K,V>, in k: K, in v: V): {
        it := d.data.iterador()
        def := False
        while (it.haySiguiente() and !def) do
            elem := it.siguiente()
            if (elem.key==k) then
                elem.val := v
                def := True
            endif
        endwhile
        if (!def) then
            newElem := new Pares
            newElem.key := k
            newElem.val := v
            d.data.agregarAtras(newElem)
        endif
    }
```

```
    proc definirRapido (inout d: dict < K, V >, in k: K, in v: V): {
        newElem := new Pares
        newElem.key := k
        newElem.val := v
        d.data.agregarAtras(newElem)
    }

    proc obtener (inout d: dict < K, V >, in k: K): V {
        it := d.data.iterador()
        find := False
        while (it.haySiguiente() and !find) do
            elem := it.siguiente()
            if (elem.key==k) then
                res := elem.val
                find := True
            endif
        endwhile
        return res
    }

    proc borrar (inout d: dict < K, V >, in k: K): {
        value := obtener(d,k)
        Elem := new Pares
        Elem.key := k
        Elem.val := value
        d.data.eliminar(elem)       }

    proc tamaño (d:dict < K, V >): int {
        return d.data.longitud()
    }

}
```

- $diccionarioVacio \in O(1)$
- $esta \in O(n)$
- $definir \in O(n)$
- $definirRapido \in O(1)$
- $obtener \in O(n)$
- $borrar \in O(n)$
- $tamano \in O(1)$