Algoritmos y Estructuras de Datos II

Primer Cuatrimestre de 2018

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 2

Diseño

Grupo:

Integrante	LU	Correo electrónico
Capdevielle, Tomás	245/17	tomas.capdevielle@gmail.com
Jiménez, Gabriel Gonzalo	407/17	gabrielnezzg@gmail.com
Martino, Maximiliano	123/17	maxii.martino@gmail.com
Soltz, Lucas Martín	205/17	lucas.m.soltz@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

1. Informe

Decisiones tomadas

1 Suponemos que no se puede hacer ninguna operación sobre una variable creada por el usuario (usando la operación *AsignarVariable*) durante la ejecución del programa.

2 Si el instante en el que asigna un valor a una variable ya existente es el mismo en que ocurre la última modificación de esa variable en *historialCompleto*, entonces sobrescribimos el valor tanto en *historialCompleto* como en su ventana.

3 Suponemos que no se puede pasar por parámetro una variable que no este entre las de la calculadora (es decir, las variables que se pasen como parámetro en las funciones *ValorVariableActual* y *ValorVariable* deben existir)

2. Módulo Calculadora

Interfaz

parámetros formales
géneros programa, rutina, nat

se explica con: CALCULADORA

géneros: `calculadora(p: programa, r: rutina, w:nat)`

usa: `instrucción`, `programa`

Operaciones básicas de *calculadora*

NUEVACALCULADORA(**in** p: `programa`, **in** r: `rutina`, **in** W : `nat`) $\rightarrow$ res : `calculadora`
Pre $\equiv$ $\{r \in \text{rutinas}(p)\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{nuevaCalculadora}(p, r)\}$
Complejidad: $O(\#p \cdot (|V| + |R|) + W \cdot \#V)$, donde:
- $\#p$ es el tamaño del programa p, medido en cantidad total de instrucciones,
- $|V|$ es la longitud del nombre más largo de alguna de las variables que aparecen en el programa,
- $|R|$ es la longitud del nombre más largo de alguna de las rutinas que aparecen en el programa,

incluyendo la rutina r,
- $\#V$ es la cantidad total de variables distintas que aparecen en el programa,
- W es la capacidad de ventana.

Descripción: Dado un programa p, una rutina r y una capacidad de ventana W, construye una calculadora inicializada para ejecutar la rutina r del programa p.
Aliasing: No produce Aliasing

FINALIZÓ?(**in** c: `calculadora`) $\rightarrow$ res : `bool`
Pre $\equiv$ $\{\text{true}\}$
Post $\equiv$ $\{res =_{\text{obs}} \neg \text{ ejecutando?}(c)\}$
Complejidad: $O(1)$
Descripción: Indica si finalizó la ejecución del programa cargado en la calculadora c.
Aliasing: No produce Aliasing

EJECUTARUNPASO(**in/out** c: `calculadora`)
Pre $\equiv$ $\{\text{ejecutando?}(c) \wedge c = c_0\}$
Post $\equiv$ $\{c =_{\text{obs}} \text{ejecutarUnPaso}(c_0)\}$
Complejidad: $O(1)$
Descripción: Ejecuta el siguiente paso de la rutina actual en el programa cargado en la calculadora c, efectuando una de las 8 operaciones posibles (ADD, SUB, MUL, PUSH, READ, WRITE, JUMP, JUMPZ), todas ellas en complejidad $O(1)$.
Aliasing: Dependiendo de la instrucción se podría llegar a modificar alguna variable de la calculadora.

ASIGNARVARIABLE(**in/out** c: `calculadora`, **in** x: `variable`, **in** n: `int`)
Pre $\equiv$ $\{c = c_0\}$
Post $\equiv$ $\{c =_{\text{obs}} \text{asignarVariable}(c_0, x, n)\}$
Complejidad: $O(|x|)$, donde $|x|$ es la longitud del nombre de x.
Descripción: Asigna el valor n a la variable en memoria x; en el caso de no existir dicha variable, la crea y la inicializa con el valor n. A diferencia de la instrucción WRITE, esta operación no incrementa el instante actual de la ejecución del programa.
Aliasing: Se modifica la variable v, o se crea.

INSTANTEACTUAL(**in** c : `calculadora`) $\rightarrow$ res : `nat`
Pre $\equiv$ $\{\text{ejecutando?}(c)\}$
Post $\equiv$ $\{res =_{\text{obs}} \text{instanteActual}(c)\}$
Complejidad: $O(1)$
Descripción: Dada una calculadora c, indica el instante de la ejecución en el que se encuentra. Los instantes son los $t \in nat$ tales que $t \in \{0, 1, 2, ...,$instante_actual $\}$.
Aliasing: No produce Aliasing.

RutinaActual(**in** c: `calculadora`) $\rightarrow$ res : `rutina`
Pre $\equiv$ {ejecutando?(c)}
Post $\equiv$ {$res =_{\text{obs}}$ rutinaActual(c)}
Complejidad: $O(1)$
Descripción: Dada una calculadora c, indica el nombre de la rutina que se esté ejecutando.
Aliasing: No produce Aliasing.

IndiceInstruccionActual(**in** c: `calculadora`) $\rightarrow$ res : `nat`
Pre $\equiv$ {ejecutando?(c)}
Post $\equiv$ {$res =_{\text{obs}}$ índiceInstrucciónActual(c)}
Complejidad: $O(1)$
Descripción: Dada una calculadora c, indica el índice (número natural) de instrucción actual dentro de la rutina que se esté ejecutando.
Aliasing: No produce Aliasing.

ValorVariableATiempoDado(**in** c: `calculadora`, **in** x: `variable`, **in** t: `nat`, **in** W : `int`) $\rightarrow$ res : `int`
Pre $\equiv$ {$t \leq$ instanteActual(c)}
Post $\equiv$ {$res =_{\text{obs}}$ valorHistóricoVariable(c, x, t)}
Complejidad: determinada según el siguiente criterio:

• Si el valor consultado es de Acceso Reciente (el instante t consultado está dentro del intervalo de tamaño W que llega hasta el instante actual, i.e. $t \geq$ instante_actual $- W$), y además la variable x es una Variable Relevante (aparece en el código fuente del programa), entonces la complejidad temporal en peor caso es de $O(|x| + log(W))$, donde $|x|$ es la longitud del nombre de x.

• Si no se cumple ninguna de las dos condiciones descritas, la complejidad temporal en peor caso es de $O(|x|+n)$, donde n es la cantidad de veces que cambió de valor la variable x.

Descripción: Dada una calculadora c, una variable arbitraria x y un instante t, devuelve el valor de dicha variable en ese instante, donde $t \in$ nat, tal que $t \in \{0, 1, 2, ...,$instante_actual $\}$
Aliasing: No produce Aliasing.

ValorVariableActual(**in** c: `calculadora`, **in** x: `variable`) $\rightarrow$ res : `int`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ valorActualVariable(c, x)}
Complejidad: $O(|x|)$, donde $|x|$ es la longitud del nombre de x.
Descripción: Dada una calculadora c y una variable arbitraria x, devuelve el valor actual de dicha variable.
Aliasing: No produce Aliasing.

Pila(**in** c: `calculadora`) $\rightarrow$ res : `pila`$\langle$`int`$\rangle$
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ pila(c)}
Complejidad: $O(1)$
Descripción: Indica el estado actual de la pila de la calculadora c.
Aliasing: No produce Aliasing.

Representación

Representación de la Calculadora

El objetivo de este módulo es implementar un diccionario de variables con sus respectivos datos, un diccionario de rutinas con sus respectivos datos, una pila de enteros y una lista con datos acerca del estado actual de la calculadora.

Estructura

`calculadora` **se representa con** `calc`

donde `calc` es `tupla`(*variables*: `diccString(datosVariable)`,
rutinas: `diccString(tupla( pPal: puntero(rutina), empieza: nat, termina: nat))`,
estadoActual: `tupla( rutActual: puntero(rutina), terminaRut: nat, instanteActual: nat, finalizó: bool)`,
instrucciones: `vector( tupla( instrucción: instrucción, condición: bool, pRut: puntero (rutina), pVar: puntero(datosVariable), próxRut: int) )`,
índiceInstrucActual: `nat`,
índicePróxInstruc: `nat`,
pila: `pila(int)`)

donde `datosVariable` es `tupla`(*pVent*: `puntero(vent)` ,
historialCompleto: `lista( tupla (instante: nat, valor: int))`)

donde `vent` es `ventana`(tupla(instante: nat, valor: int))

Observación: los tipos de datos VARIABLE y RUTINA son representado como *strings*, acorde a lo indicado en la especificación de dichos TADs en el enunciado de este trabajo.

Invariante de representación:

Rep : calc $\longrightarrow$ bool
Rep(c) $\equiv$ true $\Longleftrightarrow$

1 Las claves del `diccstring` *rutinas* son las rutinas del programa cargado en la calculadora.

2 Las claves del `diccstring` *variables* son las variables en la memoria de la calculadora.

3 Para toda instrucción en el vector *instrucciones* (primer campo de la tupla), vale que: si al hacerle Op() esta fuera READ o WRITE, entonces la variable asociada a esa instrucción deberá estar definida en el `diccstring` *variables*.

4 Para toda instrucción en el vector *instrucciones* (primer campo de la tupla), vale que: si al hacerle Op() esta fuera JUMP o JUMPZ, entonces el valor booleano (segunda posición de la tupla) es Verdadero si y solo si la rutina a la que quisiera saltar pertenece al conjunto de claves de *rutinas* y Falso en caso contrario.

5 Para toda instrucción en el vector *instrucciones* (primer campo de la tupla), vale que: si al hacerle Op() esta fuera JUMP o JUMPZ, entonces el puntero *pRut* apunta al significado del diccionario *rutinas* de la rutina a la que saltaría si el booleano de la segunda posición de la tupla es Verdadero.

6 Para toda instrucción en el vector *instrucciones* (primer campo de la tupla), vale que: si al hacerle Op() esta fuera READ o WRITE, entonces la variable a la que se le quiere aplicar la instrucción esta definida en el diccionario *variables* y *pVar* apunta al significado del diccionario *variables* de la variable a la que se quiere hacer un READ o WRITE.

7 Para toda variable definida en el diccionario *variables*, existe una instrucción en el vector instrucciones tal que su operación es READ o WRITE sobre esa variable.

8 Para toda instrucción en el vector instrucciones (primer campo de la tupla), se cumple: si al hacerle Op() esta fuera JUMP o JUMPZ, entonces, si el booleano de la segunda posición es Verdadero, *próxRut* contiene el índice de la instrucción en el vector *instrucciones* (que cumple que $0 \leq$ próxRut $<$ Longitud(e.instrucciones)) que sería la próxima en ejecutarse al hacerse el salto.

9 $0 \leq$ c.índicePróxRut $<$ Longitud(c.instrucciones)

10 c.estadoActual.finalizó = true $\rightarrow$ c.estadoActual.rutActual = NULL

11 c.índiceInstrucActual = c.índicePróxInstruc $-$ Obtener(c.estadoActual.*(rutActual)).empieza

12 c.estadoActual.rutActual $\neq$ NULL $\rightarrow$ (c.estadoActual.rutActual $\rightarrow$ Def?(res.rutinas, *(c.estadoActual.rutActual)))

13 Los significados de las variables definidas en el `diccstring` *variables* es una tupla donde *pVent* apunta a la ventana de la variable en el caso de ser una variable del código fuente; si fue agregada por el usuario, entonces no tiene ventana y *pVent* apunta a NULL. Además, *historialCompleto* es una lista enlazada de tuplas donde 0 $\leq$ instante $\leq$ instanteActual y valor es aquel que obtuvo esa variable en ese intante. Las mismas condiciones se aplican para los campos de ventana.

Función de Abstracción:

Abs : calc c $\longrightarrow$ calculadora {Rep(c)}

Abs(c) $\equiv$ c.pila $=_{\text{obs}}$ pila(calc) $\wedge$
c.estadoActual.instanteActual $=_{\text{obs}}$ instanteActual(calc) $\wedge$
c.estadoActual.rutActual $=_{\text{obs}}$ rutinaActual(calc) $\wedge$
c.índiceInstrucActual $=_{\text{obs}}$ índiceInstrucciónActual(calc) $\wedge$
armarVec(c.instrucciones) $=_{\text{obs}}$ programa(calc) $\wedge$
$(\forall\ t$: nat$)(t \leq$ c.instanteActual $\rightarrow_L$ $(\forall\ x$: variable)(Def?(c.variables, x) $\rightarrow_L$ Obtener(c.variables, x).historialCompleto[t] $=_{\text{obs}}$ valorHistóricoVariable(calc, v, t))

Función Auxiliar armarVec

armarVec: vector(tupla(instrucción, bool, puntero(rutina), puntero(datosVariable), int)) $\rightarrow$ res: vector(instrucciones)
armarVec(a) $\equiv$ **if**(Vacio?(a)) **then** <> **else** π_1(prim(a)) $\bullet$ armarVec(fin(a)) **fi**

Algoritmos

iNuevaCalculadora(**in** p : `programa`, **in** r : `rutina`, **in** W : `nat`) $\rightarrow$ res : calculadora

```
1:  res.estadoActual.instanteActual ← 0                                                          ▷ O(1)
2:  res.estadoActual.finalizó ← false                                                            ▷ O(1)
3:  res.pila ← Vacía()                                                                           ▷ O(1)
4:  res.variables ← Vacío()                                                                      ▷ O(1)
5:  res.rutinas gets Vacío()                                                                     ▷ O(1)
6:  res.índiceInstrucActual ← 0                                                                  ▷ O(1)
7:  lista(tupla(nat, variable)) vars ← Vacía()                                                   ▷ O(1)

8:  nat i ← 0                                                                                    ▷ O(1)
9:  nat j ← 0                                                                                    ▷ O(1)
10: nat empieza ← 0                                                                              ▷ O(1)

11: while i < CantidadRutinas(p) do                                                              ▷ O(#p)
12:     string RutCompleta ← RutinaI(p, i)                                                       ▷ O(|R|)
13:     puntero(string) punt ← &RutCompleta                                                      ▷ O(1)
14:     tupla(puntero(string), nat, nat) cont ← ⟨ punt, empieza, empieza + #InstruccionesRutinaI(p, i) − 1 ⟩ ▷ O(1)
15:     Definir(res.rutinas, RutinaI(p, i, cont)                                                 ▷ O(|V|)

16:     if RutinaI(p, i) = r then                                                                ▷ O(|R|)
17:         res.índicePróxInstruc ← empieza                                                      ▷ O(1)
18:         res.estadoActual.rutActual ← &obtener(res.rutinas, r).pPal                           ▷ O(|R|)
19:         res.estadoActual.terminaRut ← empieza + #InstruccionesRutinaI(p, i) − 1              ▷ O(1)
20:     else
21:         done
22:     end if

23:     empieza ← empieza + #InstruccionesRutinaI(p, i)                                          ▷ O(1)
24:     i++                                                                                      ▷ O(1)
25: end while

26: i ← 0                                                                                        ▷ O(1)
27: while i < TotalInstrucciones(p) do                                                           ▷ O(#p)
28:     if Op(InstrucciónI(p, i)) = JUMP ∨_L Op(InstrucciónI(p, i)) = JUMPZ then                 ▷ O(1)
29:         if Def?(res.rutinas, NombreRutina(InstrucciónI(p, i))) then                          ▷ O(|R|)
30:             punt ← &(obtener(res.rutinas, nombreRutina(InstrucciónI(p, i))))             ▷ O(|R|)
31:             AgregarAtrás(res.instrucciones, ⟨ InstrucciónI(p, i), true, punt, NULL, 0 ⟩  ▷ O(1)
32:         else
33:             AgregarAtrás(res.instrucciones, ⟨ InstrucciónI(p, i), false, NULL, NULL, 0 ⟩ ▷ O(1)
34:         end if
35:     else if Op(InstrucciónI(p, i)) = READ ∨_L Op(InstrucciónI(p, i)) = WRITE then            ▷ O(1)
36:         AgregarAtrás(vars, ⟨ i, NombreVariable(InstrucciónI(p, i))                           ▷ O(1)
37:         AgregarAtrás(res.instrucciones, ⟨ InstrucciónI(p, i), true, NULL, NULL, 0 ⟩      ▷ O(1)
38:     else
39:         AgregarAtrás(res.instrucciones, ⟨ InstrucciónI(p, i), true, NULL, NULL, 0 ⟩      ▷ O(1)
40:     end if
41:     i++                                                                                      ▷ O(1)
42: end while

43: i ← 0                                                                                        ▷ O(1)
44: while Longitud(vars) > 0 do                                                                  ▷ O(#V)
45:     ventana(int) vent ← NuevaVentana(W)                                                      ▷ O(W)
46:     puntero(ventana(int)) punV ← &(vent)                                                     ▷ O(1)
47:     lista(tupla(nat, int)) list ← Vacía()                                                    ▷ O(1)
48:     tupla(puntero(ventana), lista(tupla(nat,int))) tup ← ⟨ punV, list ⟩                      ▷ O(1)
49:     Definir(res.variables, Primero(vars).campo2, tup)                                        ▷ O(|V|)
50:     puntero(datosVariable) punt ← &(Obtener(res.variables, Primero(vars).campo2))            ▷ O(|V|)
```

```
51:     res.instrucciones[p, Primero(vars).campo1].pVar ← punt                    ▷ O(1)
52:     Fin(vars)                                                                 ▷ O(1)
53: end while

54: i ← 0                                                                         ▷ O(1)
55: while i < TotalInstrucciones(p) do                                            ▷ O(#p)
56:     if ((Op(InstrucciónI(p, i)) = JUMP ∨_L Op(InstrucciónI(p, i)) = JUMPZ) ∧_L res.instrucciones[i].condición =
    true then                                                                     ▷ O(1)
57:         nat pos ← Obtener(res.rutinas, NombreRutina(InstrucciónI(p, i))).comienza   ▷ O(1)
58:         res.instrucciones[i].próxRut ← pos                                    ▷ O(1)
59:     else
60:         res.instrucciones[i].próxRut ← i + 1                                  ▷ O(1)
61:     end if
62:     i++                                                                       ▷ O(1)
63: end while
```

Complejidad: $O(\#p \cdot (|V| + |R|) + W \cdot \#V)$ (ver Interfaz)
Justificación: Crear el vector de variables cuesta $O(\#p \cdot |V|)$. Y
$O(\#V \cdot (|V| + W) = \#v \cdot |V| + \#v \cdot W \leq \#p \cdot |v| + \#v \cdot W$, pues $\#v \leq \#p$. Por lo tanto, la complejidad del algoritmo en peor caso es $O(\#p \cdot (|V| + |R|) + W \cdot \#V)$.

---

iFinalizó?(**in** c: `calculadora`) → res : bool

```
1: res ← c.estadoActual.finalizó                                                  ▷ O(1)
```

Complejidad: $O(1)$
Justificación: El acceso a componente de una tupla y la asignación son operaciones elementales.

---

iEjecutarUnPaso(**in/out** c: `calculadora`)

```
1: nat i ← c.instrucciones[c.índicePróxInstruc]                                   ▷ O(1)

2: if Op(i.instrucción) = ADD then                                                ▷ O(1)
3:     if EsVacía?(c.pila) then                                                   ▷ O(1)
4:         Apilar(c.pila, 0)                                                      ▷ O(1)
5:     else if Tamaño(c.pila) = 1 then                                            ▷ O(1)
6:         done                                                                   ▷ No hacer nada
7:     else
8:         int a ← Desapilar(c.pila)                                              ▷ O(1)
9:         int b ← Desapilar(c.pila)                                              ▷ O(1)
10:        Apilar(a + b)                                                          ▷ O(1)
11:    end if
12:    c.índicePróxInstruc++                                                      ▷ O(1)
13:    c.índiceInstrucActual++                                                    ▷ O(1)

14: else if Op(i.instrucción) = SUB then                                          ▷ O(1)
15:    if EsVacía?(c.pila) then                                                   ▷ O(1)
16:        Apilar(c.pila, 0)                                                      ▷ O(1)
17:    else if Tamaño(c.pila) = 1 then                                            ▷ O(1)
18:        int a ← Desapilar(c.pila)                                              ▷ O(1)
19:        Apilar(−a)                                                             ▷ O(1)
20:    else
21:        int b ← Desapilar(c.pila)                                              ▷ O(1)
22:        int c ← Desapilar(c.pila)                                              ▷ O(1)
23:        Apilar(c − b)                                                          ▷ O(1)
24:    end if
25:    c.índicePróxInstruc++                                                      ▷ O(1)
26:    c.índiceInstrucActual++                                                    ▷ O(1)

27: else if Op(i.instrucción) = MUL then                                          ▷ O(1)
28:    if EsVacía?(c.pila) then                                                   ▷ O(1)
```

```
29:         Apilar(c.pila, 0)                                                                    ▷ O(1)
30:     else if Tamaño(c.pila) = 1 then                                                          ▷ O(1)
31:         int a ← Desapilar(c.pila)                                                            ▷ O(1)
32:         Apilar(0)                                                                            ▷ O(1)
33:     else
34:         int b ← Desapilar(c.pila)                                                            ▷ O(1)
35:         int c ← Desapilar(c.pila)                                                            ▷ O(1)
36:         Apilar(c · b)                                                                        ▷ O(1)
37:     end if
38:     c.índicePróxInstruc++                                                                    ▷ O(1)
39:     c.índiceInstrucActual++                                                                  ▷ O(1)
40: else if Op(i.instrucción) = PUSH then                                                        ▷ O(1)
41:     int a ← ValorPush(i.instrucción)                                                         ▷ O(1)
42:     Apilar(c.pila, a)                                                                        ▷ O(1)
43:     c.índicePróxInstruc++                                                                    ▷ O(1)
44:     c.índiceInstrucActual++                                                                  ▷ O(1)
45: else if Op(i.instrucción) = READ then                                                        ▷ O(1)
46:     if i.*(pVar).*(pVent) = NULL then                                                        ▷ O(1)
47:         Apilar(c.pila, 0)                                                                    ▷ O(1)
48:     else
49:         int a ← i.*(pVar).*(pVent)[Tam(i.*(pVar).*(pVent)) − 1].valor                        ▷ O(1)
50:         Apilar(c.pila, a)                                                                    ▷ O(1)
51:     end if
52:     c.índicePróxInstruc++                                                                    ▷ O(1)
53:     c.índiceInstrucActual++                                                                  ▷ O(1)
54: else if Op(i.instrucción) = WRITE then                                                       ▷ O(1)
55:     if EsVacía?(c.pila) then                                                                 ▷ O(1)
56:         tupla(instante: nat, valor: int) a ← ⟨ c.estadoActual.instanteActual, 0 ⟩            ▷ O(1)
57:         if EsVacía?(i.*(pVar).historialCompleto) ∨_L Último(i.*(pVar).historialCompleto).instante ≠
    c.estadoActual.instanteActual then                                                           ▷ O(1)
58:             Registrar(i.pVar.*pVent, a)                                                      ▷ O(1)
59:             AgregarAtrás(i.*(pVar).historialCompleto, a)                                     ▷ O(1)
60:         else
61:             Último(i.*(pVar).historialCompleto).valor ← 0                                    ▷ O(1)
62:             ventana(int) v ← i.*(pVar).*(pVent)                                              ▷ O(1)
63:             v[Tam(v) − 1].valor ← 0                                                          ▷ O(1)
64:         end if
65:         AgregarAtrás(i.pVar.historialCompleto, a)                                            ▷ O(1)
66:     else
67:         int b ← Desapilar(c.pila)                                                            ▷ O(1)
68:         tupla(instante: nat, valor: int) a ← ⟨ c.estadoActual.instanteActual, b ⟩             ▷ O(1)
69:         Registrar(i.pVar.pVent, a)                                                           ▷ O(1)
70:         AgregarAtrás(i.pVar.historialCompleto, a)                                            ▷ O(1)
71:     end if
72:     c.índicePróxInstruc++                                                                    ▷ O(1)
73:     c.índiceInstrucActual++                                                                  ▷ O(1)
74: else if Op(i.instrucción) = JUMP then                                                        ▷ O(1)
75:     if i.condición = false then                                                              ▷ O(1)
76:         c.estadoActual.finalizó ← true                                                       ▷ O(1)
77:         c.estadoActual.rutActual ← NULL                                                      ▷ O(1)
78:     else
79:         c.estadoActual.rutActual ← i.pRut                                                    ▷ O(1)
80:         c.índicePróxInstruc ← i.próxRut                                                      ▷ O(1)
81:         c.estadoActual.terminaRut ← *(i.pRut).termina                                        ▷ O(1)
82:         c.índiceInstrucActual ← 0                                                            ▷ O(1)
83:     end if
```

```
84: else if Op(i.instrucción) = JUMPZ then                                  ▷ O(1)
85:     if EsVacía?(c.pila) ∨ tope(c.pila) = 0 then                         ▷ O(1)
86:         if i.condición = false then                                     ▷ O(1)
87:             c.estadoActual.finalizó ← true                              ▷ O(1)
88:             c.estadoActual.rutActual ← NULL                             ▷ O(1)
89:         else
90:             c.estadoActual.rutActual ← i.pRut                           ▷ O(1)
91:             c.índicePróxInstruc ← i.próxRut                             ▷ O(1)
92:             c.estadoActual.terminaRut ← *(i.pRut).termina               ▷ O(1)
93:             c.índiceInstrucActual ← 0                                   ▷ O(1)
94:         end if
95:     else
96:         c.índicePróxInstruc++                                           ▷ O(1)
97:         c.índiceInstrucActual++                                         ▷ O(1)
98:     end if
99: end if

100: c.estadoActual.instanteActual++                                        ▷ O(1)
101: if c.índiceInstrucActual − 1 = c.estadoActual.terminaRut then          ▷ O(1)
102:     c.estadoActual.finalizó = true                                     ▷ O(1)
103:     c.estadoActual.rutActual ← NULL                                    ▷ O(1)
104: else
105:     done
106: end if
```

Complejidad: $O(1)$
Justificación: Todas las operaciones utilizadas tienen complejidad constante.

---

iAsignarVariable(**in/out** c: `calculadora`, **in** x: `variable`, **in** n: `int`)

```
1: tupla(instante: nat, valor:int) tup ← ⟨ c.estadoActual.instanteActual, n ⟩
2: if c.variables.Def?(c.variables, x) then
3:     int d ← Obtener(c.variables, x)                                      ▷ O(|x|)
4:     if Último(d.historialCompleto).instante = c.estadoActual.instanteActual then
5:         Último(d.historialCompleto).valor ← n
6:     else
7:         AgregarAtrás(d.historialCompleto, tup)
8:     end if
9:     puntero(int) t ← &(obtener(c.variables, x))                          ▷ O(|x|)
10:    ventana(int) v ← *(t).*(pVent)                                       ▷ O(1)
11:    if v[Tam(v) − 1].instante = c.estadoActual.instanteActual then       ▷ O(1)
12:        v[Tam(v) − 1].valor ← n                                          ▷ O(1)
13:    else
14:        Registrar(v, tup)                                                ▷ O(1)
15:    end if
16: else
17:    puntero(ventana(int)) punt ← NULL                                    ▷ O(1)
18:    lista(int) list ← Vacía()                                            ▷ O(1)
19:    AgregarAtrás(list, tup)                                              ▷ O(1)
20:    tupla(puntero(ventana(int)), lista(int)) nueva ← ⟨ punt, list ⟩      ▷ O(1)
21:    c.variables.Definir(c.variables, x, nueva)                           ▷ O(|x|)
22: end if
```

Complejidad: $O(|x|)$
Justificación: Definir y obtener en la estructura elegida para almacenar las variables (el diccString) tiene complejidad $O(|x|)$, donde $|x|$ es la longitud de la variable más larga.

---

iInstanteActual(**in** c : `calculadora`) $\rightarrow$ res : nat

1: $res \leftarrow$ c.estadoActual.instanteActual ▷ $O(1)$

Complejidad: $O(1)$
Justificación: El acceso a componente de una tupla y la asignación son operaciones elementales.

iRutinaActual(**in** c : `calculadora`) $\rightarrow$ res : rutina

1: $res \leftarrow$ *c.estadoActual.rutActual ▷ *Desreferencia* del puntero

Complejidad: $O(1)$
Justificación: Los accesos a componentes de una tupla, las asignaciones y las *desreferencias* de punteros tienen complejidad $O(1)$

iIndiceInstrucciónActual(**in** c : `calculadora`) $\rightarrow$ res : nat

1: $res \leftarrow$ c.índiceInstrucActual ▷ $O(1)$

Complejidad: $O(1)$
Justificación: El acceso a componente de una tupla y la asignación son operaciones elementales.

iValorVariableATiempoDado(**in** c : `calculadora`, **in** x : `variable`, **in** t : `nat`, **in** W : `int`) $\rightarrow$ res : int

1: **if** c.estadoActual.instanteActual $- W \leq t \wedge_L$ Obtener(c.variables, x).pVent $\neq$ NULL **then**
2: $res \leftarrow$ BinaryT(*(Obtener(c.variables, x).pVent), t) ▷ $O(|x| + log(W))$
3: **else**
4: itLista(tupla) $p \leftarrow$ CrearItUlt(Obtener(c.variables, x).historialCompleto) ▷ $O(|x|)$
5: **while** Anterior(p).valor $\neq x \wedge_L$ Anterior(p).instante $> t$ **do** ▷ $O(n)$
6: Retroceder(p) ▷ $O(1)$
7: **end while**
8: $res \leftarrow$ Anterior(p).valor ▷ $O(1)$
9: **end if**

Complejidad: $O(|x| + log(W))$ o bien $O(|x| + n)$, donde n es el largo de la lista enlazada *historialCompleto* de la estructura. (ver interfaz)
Justificación: En el caso de que se pida un instante de acceso reciente, la complejidad se explica mediante el algoritmo de búsqueda binaria $O(log(W))$, sumado al costo de la operación *obtener* del `diccString` *variables*, que es de $O(|x|)$ en peor caso (ya que dicho diccionario está montado sobre un Trie). Caso contrario, también hay que buscar en el `diccString` de variables, acarreando el correspondiente costo ya mencionado, pero además se debe recorrer, en peor caso, la totalidad de la lista enlazada *historialCompleto*, lo cual conlleva una complejidad adicional de $O(n)$, siendo n la longitud de dicha lista.

iValorVariableActual(**in** c : `calculadora`, **in** x : `variable`) $\rightarrow$ res : int

1: itLista(tupla) $p \leftarrow$ CrearItUlt(Obtener(c.variables, x).historialCompleto) ▷ $O(|x|)$
2: res $\leftarrow$ Anterior(p).valor ▷ $O(1)$

Complejidad: $O(|x|)$
Justificación: La complejidad de crear el iterador es $O(1)$, y la de *obtener* en el diccString es $O(|x|)$

iPila(**in** c : `calculadora`) $\rightarrow$ res : pila(int)

1: $res \leftarrow$ c.pila ▷ $O(1)$

Complejidad: $O(1)$
Justificación: La operación de asignación es elemental.

Función Auxiliar de búsqueda binaria

Pre ≡ La ventana está ordenada y no es vacía
Post ≡ *res* es el valor asociado al instante t en la ventana w.
Descripción: Esta función busca un cierto instante dado en la ventana de una variable, y devuelve el valor que tiene dicha variable en dicho instante.

BinaryT (**in** w : `ventana(`α`)`, **in** t : `nat`) → res : int

```
1:  if Tam(w) = 1 then                                                  ▷ O(1)
2:      res ← w[0].valor                                                ▷ O(1)
3:  else if w[Tam(w) − 1].instante ≤ t then                             ▷ O(1)
4:      res ← w[Tam(w) − 1].valor                                       ▷ O(1)
5:  else
6:      nat low ← 0                                                     ▷ O(1)
7:      nat high ← Tam(w) − 1                                           ▷ O(1)
8:      while low + 1 < high ∧_L w[low].instante ≠ t do                 ▷ O(log(Tam(w)))
9:          nat mid ← low + (high − low)/2                              ▷ O(1)
10:         if w[mid].instante ≤ t then                                 ▷ O(1)
11:             low ← mid                                               ▷ O(1)
12:         else
13:             high ← mid                                              ▷ O(1)
14:         end if
15:     end while
16:     res ← w[low].valor                                              ▷ O(1)
17: end if
```

Complejidad: $O(log(W))$, donde W es el tamaño de la ventana.
Justificación: Al ser un algoritmo de busqueda binaria sobre un arreglo ordenado, por lo visto en clase, se sabe que la complejidad en peor caso es de $O(log(W))$.

Servicios Usados

De Vector
- AgregarAtrás debe ser O(1)

De Puntero(α)
- $*\bullet$ debe ser O(1)
- $\&\bullet$ debe ser O(1)

3. Módulo Diccionario_String

Interfaz

parámetros formales
géneros α

se explica con: DICC(STRING, α), CONJ(STRING).

géneros: `diccString(`α`)`.

Operaciones básicas del diccionario

VACIO() $\rightarrow$ *res* : `diccString(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ vacío}
Complejidad: $O(1)$
Descripción: genera un diccionario vacío.

DEFINIR(**in/out** d: `diccString(`α`)`, **in** s: `string`, **in** a: α)
Pre $\equiv$ {$d =_{\text{obs}} d_0$}
Post $\equiv$ {$d =_{\text{obs}}$ definir(p, a, d)}
Complejidad: $O(|P| + copiar(a))$ siendo P la clave mas larga.
Descripción: define a en d con la clave s.
Aliasing: el elemento a se define por copia.

DEF?(**in** p: `string`, **in** d: `diccString(`α`)`) $\rightarrow$ *res* : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ def?(p, d)}
Complejidad: $O(|P|)$ siendo P la clave mas larga.
Descripción: devuelve `true` si y sólo si la p tiene una definicion en d.

OBTENER(**in** p: `string`, **in** d: `diccString(`α`)`) $\rightarrow$ *res* : α
Pre $\equiv$ {def?(s, d)}
Post $\equiv$ {alias(*res* $=_{\text{obs}}$ obtener(p, d))}
Complejidad: $O(|P|)$ siendo P la clave mas larga.
Descripción: devuelve el significado de la la clave p en d.
Aliasing: *res* es modificable si y sólo si d es modificable.

CLAVES(**in** d: `diccString(`α`)`) $\rightarrow$ *res* : `conj(string)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(*res* $=_{\text{obs}}$ claves(d))}
Complejidad: $O(1)$
Descripción: Devuelve del conjunto de claves de d. Aliasing: *res* se modifica sii claves(d) se modifica

Representación

Representación del Diccionario_String

En este módulo se utiliza un trie (estructura de datos vista en la materia) para definir un diccionario cuyas claves son string. La idea tras esta decisión consiste en que la complejidad temporal en peor caso de las operaciones *definir* y *obtener* no dependa de la cantidad de claves, sino de la longitud de la clave.

Estructura

`diccString(`α`)` **se representa con** `estr`

donde `estr` es `tupla`(*raiz*: `puntero(nodo)`,
claves: `conj(string)`)

donde `nodo` es `tupla`(*definición*: `puntero(`α`)`,
siguientes: `arreglo(puntero(nodo))`,
itClave: `puntero(itConj(string))`)

Invariante de representación en castellano:

- Si raíz es $NULL$ entonces el conjunto de claves es vacío.
- Si la raíz no es $NULL$ entonces el conjunto de claves es no vacío.
- Todos los elementos del conjunto de claves están definidos en el diccString.
- Ningún nodo tiene entre sus siguientes a un nodo que esta "antes" que él.

Función de Abstracción:

Abs : estr d $\longrightarrow$ dicc(string, α) {Rep(d)}
Abs(d) $\equiv$ dic : dicc(string, α) /
claves(dic) $=_{\text{obs}}$ d.claves $\wedge$

($\forall s$: string) ((def?(s, dic) $=_{\text{obs}}$ $s \in d$.claves) $\wedge_{\text{L}}$
(def?(s, dic) $\Rightarrow_{\text{L}}$ obtener(s, dic) $=_{\text{obs}}$ significado de s en la estructura))

Algoritmos

---

iVacio() $\rightarrow$ res : diccString(α)

1: puntero(nodo) $iRaiz \leftarrow NULL$ ▷ $O(1)$
2: conj($string$) $iClaves \leftarrow$ Vacio() ▷ $O(1)$
3: $res \leftarrow \langle\ iRaiz, iClaves\ \rangle$ ▷ $O(1)$
4: Complejidad: $O(1)$
5: Justificación: $O(1) + O(1) + O(1)$

---

iClaves(**in** d: `diccString(`α`)`) $\rightarrow$ res : conj(α)

1: $res \leftarrow d.claves$ ▷ $O(1)$
2: Complejidad: $O(1)$

---

iObtener(**in** p: `string`, **in** d: `diccString(`α`)`) $\rightarrow$ res : α

1: puntero(nodo) $n \leftarrow raiz$ ▷ $O(1)$
2: nat $i \leftarrow 0$ ▷ $O(1)$
3: **while** $i <$ Longitud(p) **do** ▷ Se repite $|p|$ $O(1)$
4: $actual \leftarrow (*actual).siguientes[\text{ord}(p[i])]$ ▷ $O(1)$
5: $i \leftarrow i + 1$ ▷ $O(1)$
6: **end while**
7: $res \leftarrow (*actual).definicion$ ▷ $O(1)$
8: Complejidad: $O(|P|)$
9: Justificación: Siendo $|P|$ el largo de la clave mas larga, sea cual sea p, $|p| \leq |P|$ entonces $O(|p|) = O(|P|)$

---

iDefinir(**in/out** d: `diccString(`α`)`, **in** p: `string`, **in** a: α)

```
1: Puntero(nodo) actual ← raiz                                              ▷ O(1)
2: Nat i ← 0                                                                ▷ O(1)
3: bool esNueva ← false                                                     ▷ O(1)
4: while i < Longitud(p) do                                                 ▷ Se repite |p| O(1)
5:     if (*actual).siguientes[ord(p[i])] = NULL then                       ▷ O(1)
6:         arreglo(puntero(Nodo)) sig ← CrearArreglo(256)                   ▷ O(1)
7:         for i = 0 to 256 do                                              ▷ se repite siempre 256 veces O(1)
8:             sig[i] ← NULL                                                ▷ O(1)
9:         end for
10:        (*actual).siguientes[ord(p[i])] ← & ⟨ NULL, sig, NULL ⟩          ▷ O(1)
11:        esNueva ← true                                                   ▷ O(1)
12:    end if
13:    actual ← (*actual).siguientes[ord(p[i])]                             ▷ O(1)
14:    i ← i + 1
15: end while
16: if (*actual).definicion ≠ NULL then                                     ▷ O(1)
17:    (*actual).definicion ← NULL                                          ▷ se libera la memoria acupada por definicion O(1)
18: end if
19: (*actual).definicion ← & copiar(a)                                      ▷ O(copiar(α))
20: if esNueva then                                                         ▷ O(1)
21:    (*actual).itClave ← claves.AgregarRapido(s)                          ▷ O(1)
22: end if
```

23: Complejidad: $O(|P| + copiar(\alpha))$
24: Justificación: Siendo $|P|$ el largo de la clave mas larga, sea cual sea p, $|p| \leq |P|$ entonces $O(|p|) = O(|P|)$ mas lo que cueste copiar a.

iDef?(**in** p: `string`) $\rightarrow$ res : $bool$

```
1: Nat i ← 0                                                                ▷ O(1)
2: bool pertenece ← true                                                    ▷ O(1)
3: puntero(nodo) actual ← raiz                                              ▷ O(1)
4: while i < Longitud(p) ∧ pertenece do                                     ▷ Se repite |p| O(1)
5:     if (*actual).siguientes[ord(p[i])] = NULL then                       ▷ O(1)
6:         pertenece ← false                                                ▷ O(1)
7:     end if
8:     actual ← (*actual).siguientes[ord(p[i])]                             ▷ O(1)
9:     i ← i + 1                                                            ▷ O(1)
10: end while
11: if (*actual).significado = NULL then                                    ▷ O(1)
12:    pertenece ← false                                                    ▷ O(1)
13: end if
14: res ← pertenece                                                         ▷ O(1)
```

15: Complejidad: $O(|P|)$
16: Justificación: Siendo $|P|$ el largo de la clave mas larga, sea cual sea p, $|p| \leq |P|$ entonces $O(|p|) = O(|P|)$

Pre $\equiv$ {*desde* no nulo}
Post $\equiv$ {deveulve la cantidad de punteros no nulos en *desde*.siguientes}

---

iCuentaHijos(**in** *desde* : `puntero(nodo)` $\rightarrow$ *res* : *nat*)

```
1: nat i ← 0                                                          ▷ O(1)
2: nat hijos ← 0                                                      ▷ O(1)
3: while i < 256 do                                                   ▷ se repite siempre 256 veces O(1)
4:     if (*actutal).siguiente[i] ≠ NULL then
5:         O(1)
6:         suma ← suma + 1                                            ▷ O(1)
7:         i ← i + 1                                                  ▷ O(1)
8:     end if
9: end while
10: res ← hijos                                                       ▷ O(1)
```

11: Complejidad: $O(1)$
12: Justificación: $O(1) + O(1) + O(256) + O(1) + O(1) + O(1) = O(261) = O(1)$

Servicios Usados

De Conjunto Lineal
- Vacio() debe ser O(1)
- AgregarRapido(conj(α), α α) debe ser O(copy(a))
- EliminarSiguiente(itConj(α)) debe ser O(1)

De String
- Longitud(string) debe ser O(1)

De Char
- ord(char) debe ser O(1)

4. Módulo Instrucción

Interfaz

parámetros formales
géneros int, string

se explica con: INSTRUCCIÓN

géneros: `instrucción`

usa: `Operación`

Operaciones básicas de instrucción

OP(**in** i : `instrucción`) $\rightarrow$ res : `operación`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ op(i)}
Complejidad: $O(1)$
Descripción: Indica qué operación corresponde a la instrucción i.

VALORPUSH(**in** i : `instrucción`) $\rightarrow$ res : `int`
Pre $\equiv$ {op(i) = PUSH}
Post $\equiv$ {$res =_{\text{obs}}$ constanteNumérica(i)}
Complejidad: $O(1)$
Descripción: Indica el valor a *pushear* si la instrucción i es PUSH.

NOMBREVARIABLE(**in** i : `instrucción`) $\rightarrow$ res : `string`
Pre $\equiv$ {op(i) = WRITE $\vee$ op(i) = READ}
Post $\equiv$ {$res =_{\text{obs}}$ nombreVariable(i)}
Complejidad: $O(1)$
Descripción: Indica el nombre de la variable a leer o modificar si la instrucción i es READ o WRITE.

NOMBRERUTINA(**in** i : `instrucción`) $\rightarrow$ res : `rutina`
Pre $\equiv$ {op(i) = JUMP $\vee$ op(i) = JUMPZ}
Post $\equiv$ {$res =_{\text{obs}}$ nombreRutina(i)}
Complejidad: $O(1)$
Descripción: Indica el nombre de la rutina a la que se salta, si la instrucción i es JUMP o JUMPZ.

PUSH(**in** n : `int`) $\rightarrow$ res : `instrucción`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ IPUSH(n)}
Complejidad: $O(1)$
Descripción: Instrucción correspondiente a la operación PUSH

ADD() $\rightarrow$ res : `instrucción`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ IADD}
Complejidad: $O(1)$
Descripción: Instrucción correspondiente a la operación ADD

SUB() $\rightarrow$ res : `instrucción`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ ISUB}
Complejidad: $O(1)$
Descripción: Instrucción correspondiente a la operación SUB

MUL() $\rightarrow$ res : `instrucción`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ IMUL}
Complejidad: $O(1)$

Descripción: Instrucción correspondiente a la operación MUL

READ(**in** x : `variable`) $\rightarrow$ res : `instrucción`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ IREAD(x)}
Complejidad: $O(1)$
Descripción: Instrucción correspondiente a la operación READ

WRITE(**in** x : `variable`) $\rightarrow$ res : `instrucción`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ IWRITE(x)}
Complejidad: $O(1)$
Descripción: Instrucción correspondiente a la operación WRITE

JUMP(**in** r : `rutina`) $\rightarrow$ res : `instrucción`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ IJUMP(r)}
Complejidad: $O(1)$
Descripción: Instrucción correspondiente a la operación JUMP

JUMPZ(**in** r : `rutina`) $\rightarrow$ res : `instrucción`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ IJUMPZ(r)}
Complejidad: $O(1)$
Descripción: Instrucción correspondiente a la operación JUMPZ

Representación

Representación de la Instrucción

Esta estructura corresponde a la representación del TAD Instrucción, con la debida especificación e implementación en pseudo-código de sus observadores y generadores, utilizando 4 campos distintos para la tupla denominada *instruc*: *op*, donde se establece la operación que da lugar a la instrucción construida; *valor*, usada para indicar el entero que sirve de parámetro para la operación PUSH, que coloca el valor en cuestión en el tope de la pila de la calculadora; *rut*, correspondiente al nombre de la rutina que va aparejado a las operaciones JUMP y JUMPZ, y hace referencia a la rutina del programa cargado en la calculadora a la que se va a saltar; y finalmente *var*, que se refiere a la variable (string) que se lee o escribe (en las operaciones READ y WRITE, respectivamente) en la calculadora al ejecutar dichas operaciones.

Estructura

`instrucción` **se representa con** `instruc`

donde `instruc` es `tupla`(*op*: `operación`,
valor: `int`,
rut: `rutina`,
var: `variable`)

Observación: el tipo de datos OPERACIÓN es representado con un *string*.

Invariante de representación:

Rep : instruc $\longrightarrow$ bool
Rep(i) $\equiv$ true $\Longleftrightarrow$

1 Una vez creada una instrucción, el campo i.op será usado y los demás campos se inicializarán siguiendo el esquema:

- El campo i.valor si y sólo si i.op es PUSH;
- El campo i.rut si y sólo si i.op es JUMP o JUMPZ;
- El campo i.var si y sólo si i.op es READ o WRITE;

Función de Abstracción:

Abs : instruc i $\longrightarrow$ instrucción $\{\text{Rep}(i)\}$
Abs(i) $\equiv$ i.op $=_{\text{obs}}$ operación(instrucción) $\wedge$
i.valor $=_{\text{obs}}$ constanteNumérica(instrucción) $\wedge$
i.rut $=_{\text{obs}}$ nombreRutina(instrucción) $\wedge$
i.var $=_{\text{obs}}$ nombreVariable(instrucción)

Algoritmos

iOp(**in** i : `instrucción`) $\rightarrow$ res : operación

1: $res \leftarrow i.op$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Las asignaciones, al ser operaciones elementales, tienen complejidad $O(1)$

iValorPush(**in** i : `instrucción`) $\rightarrow$ res : int

1: $res \leftarrow i.valor$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Las asignaciones, al ser operaciones elementales, tienen complejidad $O(1)$

iNombreVariable(**in** i : `instrucción`) $\rightarrow$ res : variable

1: $res \leftarrow i.var$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Las asignaciones, al ser operaciones elementales, tienen complejidad $O(1)$

iNombreRutina(**in** i : `instrucción`) $\rightarrow$ res : rutina

1: $res \leftarrow i.rut$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Las asignaciones, al ser operaciones elementales, tienen complejidad $O(1)$

iPush(**in** n : `int`) $\rightarrow$ res : instrucción

1: $res.op \leftarrow$PUSH ▷ $O(1)$
2: $res.valor \leftarrow n$ ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Las asignaciones, al ser operaciones elementales, tienen complejidad $O(1)$

iAdd() $\rightarrow$ res : instrucción

1: $res.op \leftarrow$ADD ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Las asignaciones, al ser operaciones elementales, tienen complejidad $O(1)$

iSub() $\rightarrow$ res : instrucción

1: $res.op \leftarrow$SUB ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Las asignaciones, al ser operaciones elementales, tienen complejidad $O(1)$

iMul() $\rightarrow$ *res* : instrucción

1: *res.op* $\leftarrow$MUL ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Las asignaciones, al ser operaciones elementales, tienen complejidad $O(1)$

iRead(**in** *x* : `variable`) $\rightarrow$ *res* : instrucción

1: *res.op* $\leftarrow$READ ▷ $O(1)$
2: *res.var* $\leftarrow$ *x* ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Las asignaciones, al ser operaciones elementales, tienen complejidad $O(1)$

iWrite(**in** *x* : `variable`) $\rightarrow$ *res* : instrucción

1: *res.op* $\leftarrow$WRITE ▷ $O(1)$
2: *res.var* $\leftarrow$ *x* ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Las asignaciones, al ser operaciones elementales, tienen complejidad $O(1)$

iJump(**in** *r* : `rutina`) $\rightarrow$ *res* : instrucción

1: *res.op* $\leftarrow$JUMP ▷ $O(1)$
2: *res.rut* $\leftarrow$ *r* ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Las asignaciones, al ser operaciones elementales, tienen complejidad $O(1)$

iJumpZ(**in** *r* : `rutina`) $\rightarrow$ *res* : instrucción

1: *res.op* $\leftarrow$JUMPZ ▷ $O(1)$
2: *res.rut* $\leftarrow$ *r* ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Las asignaciones, al ser operaciones elementales, tienen complejidad $O(1)$

5. Módulo Programa

Interfaz

parámetros formales
géneros rutina

usa: `Instrucción`

se explica con: PROGRAMA

géneros: `programa`.

Operaciones básicas de programa

NUEVOPROGRAMA() $\rightarrow$ *res* : `programa`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ nuevoPrograma}
Complejidad: $O(1)$
Descripción: Crea una nueva instancia de programa.

AGREGARINSTRUCCIÓN(**in/out** p: `programa`, **in** r: `rutina`, **in** i: `instrucción`)
Pre $\equiv$ {$p = p_0$}
Post $\equiv$ {$p =_{\text{obs}}$ agInstrucción(p_0, r, i) }
Complejidad: O(cantidadRutinas(p)))
Descripción: Agrega la instrucción i a la rutina r. Si esta no existe, la crea.

CANTIDADRUTINAS(**in** p: `programa`) $\rightarrow$ *res* : `nat`
Pre $\equiv$ {true}
Post $\equiv$ {*res* $=_{\text{obs}}$ #rutinas(p)}
Complejidad: $O(1)$
Descripción: Devuelve la cantidad total de rutinas del programa p.

RUTINAI(**in** p: `programa`, **in** i: `nat`) $\rightarrow$ *res* : `rutina`
Pre $\equiv$ {$0 \leq i <$CantidadRutinas(p)}
Post $\equiv$ {*res* $=_{\text{obs}}$ dameUno(rutinas(p))}
Complejidad: $O(1)$
Descripción: Devuelve la rutina asociada al número i en el programa p.

#INSTRUCCIONESRUTINAI(**in** p: `programa`, **in** i: `nat`) $\rightarrow$ *res* : `nat`
Pre $\equiv$ {$0 \leq i <$CantidadRutinas(p)}
Post $\equiv$ {*res* $=_{\text{obs}}$ longitud(RutinaI(p, i))}
Complejidad: $O(1)$
Descripción: Devuelve la cantidad de instrucciones de la rutina asociada al número i.

INSTRUCCIÓNJDERUTINAI(**in** p: `programa`, **in** i: `nat`, **in** j: `nat`) $\rightarrow$ *res* : `instrucción`
Pre $\equiv$ {$0 \leq i <$CantidadRutinas(p) $\wedge_L$ $0 \leq j <$#InstruccionesRutinaI(p, i)}
Post $\equiv$ {*res* $=_{\text{obs}}$ instrucción(p, RutinaI(p, i), j)}
Complejidad: $O(1)$
Descripción: Devuelve la *j-ésima* instrucción de la rutina asociada al indice i

TotalInstrucciones(**in** p : `programa`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv$ $\{\sum_{i=1}^{CantidadRutinas(p)-1} \#InstruccionesRutinaI(p,i)\}$
Complejidad: O(CantidadRutinas(p))
Descripción: Devuelve la cantidad total de instrucciones del programa p.

InstrucciónI(**in** p : `programa`, **in** i : `nat`) $\rightarrow$ res : `instrucción`
Pre $\equiv$ {$0 \leq i <$TotalInstrucciones(p)}
Post $\equiv$ {$(\exists\ r$: nat$)(0 \leq r <$CantidadRutinas(p)}
Complejidad: $O(1)$
Descripción: Devuelve la *j-ésima* instrucción de la rutina asociada al indice i

Representación

Representación del Programa

La siguiente estructura se utiliza para representar el TAD PROGRAMA de esta forma: El vector de instrucciones *prog* contiene todas las instrucciones que componen el programa, sin hacer distinción sobre la rutina en la que se encuentren. Por otro lado, el vector *rutinas* almacena las rutinas del programa a través de tuplas, que contienen 3 componentes, a saber:

- *nombre*: corresponde al nombre de la rutina;
- *empieza*: valor natural que se refiere al índice asociado a la primera instrucción de dicha rutina en el vector de instrucciones *prog*.
- *termina*: de manera similar que *empieza*, este valor natural hace referencia al índice asociado a la última instrucción de la rutina en el vector de instrucciones *prog*.

De esta forma, quedan representadas todas las instrucciones del programa por un lado, y su forma de organización en rutinas, por otro.

Estructura

`programa` **se representa con** `estr`

donde `estr` es `tupla`(*prog*: `vector(instrucción)`,
rutinas: `vector( tupla(nombre: rutina, empieza: nat, termina: nat) )`)

Observación: el tipo de datos RUTINA es representado con un *string*.

Invariante de representación:

Rep : estr $\longrightarrow$ bool
Rep(p) $\equiv$ true $\Longleftrightarrow$ (Longitud(p.rutinas) = 0 $\wedge_L$ Longitud(p.prog) = 0) $\vee_L$
(Longitud(p.prog) = (p.rutinas[Longitud(p.rutinas) − 1].termina + 1))

Función de Abstracción:

Abs : estr p $\longrightarrow$ programa {Rep(p)}
Abs(p) $\equiv$ armarConj(p.rutinas) $=_{\text{obs}}$ rutinas(programa) $\wedge$
((∀ r: rutina)(r $\in$ rutinas(programa) $\rightarrow_L$ p.rutinas[damePos(p.rutinas, r)].termina − p.rutinas[damePos(p.rutinas, r)].empieza + 1 = longitud(programa, r)) $\wedge$
((∀ r: rutina)(∀ n: nat)((r $\in$ rutinas(programa) $\wedge$ 0 $\leq$ n $<$ longitud(programa, r)) $\rightarrow_L$ p.prog[p.rutinas[damePos(p.rutinas, r)].empieza + n] = instrucción(programa, r, n)))

Función Auxiliar armarConj
armarConj: vector(tupla(rutina, nat, nat)) → res: conj(rutina)
armarConj(a) ≡ **if**(Vacio?(a)) **then** vacío **else** Ag(π_1(prim(a), fin(a)) **fi**

Función Auxiliar damePos
damePos: vector(tupla(rutina, nat, nat)) x rutina → res: nat
damePos(a, r) ≡ **if** π_1(prim(a)) = r **then** 0 **else** 1 + damePos(fin(a), r) **fi**

Algoritmos

iNuevoPrograma() → *res* : programa

```
1: res.prog ← vacía()                                   ▷ O(1)
2: res.rutinas ← vacía()                                ▷ O(1)
```

Complejidad: $O(1)$
Justificación: Las asignaciones son operaciones básicas, por lo que su complejidad es $O(1)$, y la operación $Vacia()$ del módulo básico Vector(α) es $\theta(1)$.

---

iAgregarInstrucción(**in/out** p: `programa`, **in** r: `rutina`, **in** i: `instrucción`)

```
1: nat j ← 0                                                                  ▷ O(1)
2: while j < Longitud(p.rutinas) ∧ p.rutinas[j].nombre ≠ r do                 ▷ O (CantidadRutinas(p)
3:     j++                                                                    ▷ O(1)
4: end while
5: if j < Longitud(p.rutinas) then                                            ▷ O(1)
6:     AgregarAtrás(p.prog, i)                                                ▷ O(1)
7:     nat t ← p.rutinas[j].termina + 1                                       ▷ O(1)
8:     while t < Longitud(p.prog) do                                          ▷ O (CantidadRutinas(p)
9:         swap(p.prog, t, Longitud(p.prog) − 1)                              ▷ O(1)
10:        t++                                                                ▷ O(1)
11:    end while
12:    p.rutinas[j].termina++                                                 ▷ O(1)
13: else
14:    AgregarAtrás(p.prog, i)                                                ▷ O(1)
15:    tupla(rutina, nat, nat) nuevaTup ← ⟨ r, Longitud(p.prog) − 1, Longitud(p.prog) − 1 ⟩   ▷ O(1)
16:    AgregarAtrás(p.rutinas, nuevaTup)                                      ▷ O(1)
17: end if
```

Complejidad: $O(\text{cantidadRutinas}(p)))$
Justificación: En ambos ciclos se recorre el vector de rutinas.

---

iCantidadRutinas(**in** p: `programa`) → *res* : nat

```
1: res ← Longitud(p.rutinas)                            ▷ O(1)
```

Complejidad: $O(1)$
Justificación: Las asignaciones son operaciones básicas, por lo que su complejidad es $O(1)$, y la operación $Longitud()$ del módulo básico Vector(α) es $\theta(1)$.

---

iRutinaI(**in** p: `programa`, **in** i: `nat`) → *res* : rutina

```
1: res ← p.rutinas[i].nombre                            ▷ O(1)
```

Complejidad: $O(1)$
Justificación: Las asignaciones son operaciones básicas, por lo que su complejidad es $O(1)$, y la operación $\bullet[\ ]\bullet$ del módulo básico Vector(α) es $\theta(1)$.

---

i#InstruccionesRutinaI(**in** p : `programa`, **in** i : `nat`) $\rightarrow$ res : nat

1: $res \leftarrow$ p.rutinas[i].termina − p.rutinas[i].empieza + 1 ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Las asignaciones son operaciones básicas, por lo que su complejidad es $O(1)$, y la operación •[ ]• del módulo básico Vector(α) es $\theta(1)$.

iInstrucciónJDeRutinaI(**in** p : `programa`, **in** i : `nat`, **in** j : `nat`) $\rightarrow$ res : instrucción

1: $res \leftarrow$ p.prog[p.rutinas[i]+j] ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Las asignaciones son operaciones básicas, por lo que su complejidad es $O(1)$, y la operación •[ ]• del módulo básico Vector(α) es $\theta(1)$.

iTotalInstrucciones(**in** p : `programa`) $\rightarrow$ res : nat

1: nat $i \leftarrow 0$ ▷ $O(1)$
2: $res \leftarrow 0$ ▷ $O(1)$
3: **while** $i <$ CantidadRutinas(p) **do** ▷ O (CantidadRutinas(p)
4: $res \leftarrow res +$ #InstruccionesRutinaI(p, i)
5: i++ ▷ $O(1)$
6: **end while**

Complejidad: O(CantidadRutinas(p))
Justificación: Se recorre una vez el vector de rutinas, y la función #InstruccionesRutinaI es $O(1)$.

iInstrucciónI(**in** p : `programa`, **in** i : `nat`) $\rightarrow$ res : instrucción

1: $res \leftarrow$ p.prog[i] ▷ $O(1)$

Complejidad: $O(1)$
Justificación: Las asignaciones son operaciones básicas, por lo que su complejidad es $O(1)$, y la operación •[ ]• del módulo básico Vector(α) es $\theta(1)$.

Función Auxiliar Swap

Pre $\equiv 0 \leq i <$ Longitud(v) $\wedge\ 0 \leq j <$ Longitud(v) $\wedge\ v = v_0$
Post $\equiv$ Todos los elementos de v siguen iguales salvo en las posiciones i y j, donde ocurre que: $v[i] = v_0[j] \wedge v[j] = v_0[i]$
Descripción: Intercambia los valores asociados a los índices i y j del vector.

Swap (**in/out** v : `vector(`α`)`, **in** i : `nat`, **in** j : `nat`)

1: α a $\leftarrow v[i]$
2: $v[i] \leftarrow v[j]$
3: $v[j] \leftarrow a$

Complejidad: $O(1)$
Justificación: Las asignaciones son operaciones elementales, y la indexación en vectores es $O(1)$.