DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Practico 2

Calculadora Programable

Algoritmos y Estructuras de Datos II

Integrante	LU	Correo electrónico
Salvia, Daniel	068/17	danmats140@gmail.com
Ampuero, Jose	645/16	ampuero.jose96@gmail.com
Cabrera, Martin	762/16	martincabrera98@gmail.com
Segura, Mariano	235/17	marianosegura90@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

1. Módulo Calculadora

Interfaz

se explica con: CALCULADORA, PROGRAMA, INSTRUCCION, DICCIONARIO(κ), VENTANA(σ), RUTINA, INT, NAT,
géneros: `Calc`.

El módulo calculadora provee una calculadora en la cual se puede ejecutar una rutina dada, asignar u obtener valores de una determinada variable en el momento actual o en algun otro momento anterior. Ademas se puede obtener el nombre de la rutina que se esta ejecutando, el indice de la instruccion que se esta ejecutando, la pila con los valores de la calculadora y el instante actual de la calculadora.

La calculadora se inicializa preparada para realizar la rutina. Se ejecutara una instrucción de la rutina cada vez que se ejecute la funcion EjecutarUnPaso.
Operaciones:

NUEVACALCULADORA(**in** $Prog$: `Programa`, **in** r : `Rutina` ,**in** W : `Nat`) $\rightarrow$ res : `Calc`
Pre $\equiv$ {W $>$ 0}
Post $\equiv$ {$res =_{\text{obs}} nuevaCalculadora(\widehat{Prog}, \widehat{r})$}
Complejidad: $\mathcal{O}(\#p.(|V| + |R|) + W.\#V)$
Descripción: Crea una calculadora con el programa indicado y preparado para ejecutar la rutina enviados por parametro.
Aliasing: Borrar las ventanas que genera NuevaCalculadora puede invalidar iteradores

EJECUTANDO?(**in** c: `Calc`) $\rightarrow$ res : `Bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ Ejecutando?($\widehat{c}$)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve true si aun hay instrucciones de la rutina que no se ejecutaron. En caso contrario devuelve false.

EJECUTARUNPASO(**in/out** c: `Calc`)
Pre $\equiv$ {ejecutando?($\widehat{c}$) $\wedge$ $c = c_0$}
Post $\equiv$ {$c =_{\text{obs}}$ ejecutarUnPaso($\widehat{c_0}$)}
Complejidad: $\mathcal{O}(1)$
Descripción: Ejecuta la instrucción de la rutina actual indicada por el indice de instruccion actual.

ASIGNARVARIABLE(**in/out** c: `Calc`, **in** v: `Variable`, **in** i: `int`)
Pre $\equiv$ {$c = c_0$}
Post $\equiv$ {$c =_{\text{obs}}$ asignarVariable($\widehat{c_0}, \widehat{v}, \widehat{i}$)}
Complejidad: $\mathcal{O}(|v|)$
Descripción: Asigna el valor i a la variable v dentro de la calculadora.

VALORHISTORICOVARIABLE(**in** c: `Calc`, **in** v: `Variable` ,**in** t: `Nat`) $\rightarrow$ res : `Int`
Pre $\equiv$ {$\widehat{t}$ <=instanteActual($\widehat{c}$)}
Post $\equiv$ {$res =_{\text{obs}}$ ValorHistoricoVariable($\widehat{c}, \widehat{v}, \widehat{t}$)}
Complejidad: $\mathcal{O}(|v| + log(W))$
Descripción: Devuelve el valor de la variable v en el instante t.

VALORACTUALVARIABLE(**in** c: `Calc`, **in** v: `Variable`) $\rightarrow$ res : `Int`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ ValorActualVariable($\widehat{c}, \widehat{v}$)}
Complejidad: $\mathcal{O}(|v|)$
Descripción: Devuelve el valor actual de la variable v.

IndiceInstrucciónActual(**in** c: `Calc`) $\rightarrow$ res : `Nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ IndiceInstrucciónActual($\widehat{c}$)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el indice de la instruccion actual de la rutina que se esta ejecutando en el momento en la calculadora.

InstanteActual(**in** c: `Calc`) $\rightarrow$ res : `Nat`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ InstanteActual($\widehat{c}$)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el instante actual de la calculadora.

RutinaActual(**in** c: `Calc`) $\rightarrow$ res : `Rutina`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ RutinaActual($\widehat{c}$)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el nombre de la rutina que se esta ejecutando en el momento en la calculadora.

Pila(**in** c: `Calc`) $\rightarrow$ res : `Pila`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ pila($\widehat{c}$)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve la pila de la calculadora.

Representación

1.1. Representación de la calculadora

`Calculadora` **se representa con** `Calc`

donde `Calc` es `tupla`(*InstanteActual*: `Nat`
, *CapacidadVentana*: `Nat`
, *IndiceRutinaActual*: `Nat`
, *IndiceInstrucción*: `Nat`
, *Pila*: `Pila(Int)`
, *ProgCalc*: `Array⟨Tupla⟨rut:Rutina,Instrucciones:Lista⟨Instruccion:instr,`
`indiceRutaSaltar:Nat,punteroAVentana:Puntero(Ventana⟨Tupla⟨Instante:int,Valor:Nat⟩⟩),`
`cantInstrucciones:Nat⟩⟩⟩`
, *Asignaciones*: `Lista⟨Tupla⟨Nat,Variable,Int⟩⟩`
, *CantAsignaciones*: `Nat` , *VarVentana*: `diccTrie(Variable,Ventana⟨Tupla⟨int,Nat⟩⟩)`
, *VarAsignacionActual*: `diccTrie(Variable,Int)`
, *Inicio*: `Tupla⟨Programa,Nat⟩`
, *ITinstrucción*: `itLista⟨Instruccion:instr,`
`indiceRutaSaltar:Nat,punteroAVentana:Puntero(Ventana⟨Tupla⟨Instante:int,Valor:Nat⟩⟩)`
)

1.2. Invariante de representación

1.InstanteActual es mayor o igual al segundo elemento de la tupla perteneciente al tope de cada ventana en VarVentana.
2.El tamaño de las ventanas es igual a CapacidadVentana.
3.Todas las ventanas de VarVentana estan ordenadas de menor a mayor segun el segundo elemento de la tupla.
4.Todas las claves de VarVentana aparecen al menos una vez en alguna lista de instrucciones de ProgCalc.
5.Los iteradores en ProgCalc apuntan a las ventanas en varVentana.
6.En ProgCalc no hay 2 o mas rutinas con el mismo nombre.
7.IndiceRutinaActual es mayor o igual a cero y menor a la longitud de ProgCalc
8.IndiceInstruccion es mayor o igual a cero y menor o igual a la longitud de la lista de instruccion en la rutina indicada

por el IndiceRutinaActual.
9.Si OP(instr) es OJUMP o OJUMPZ, el nat dentro de vector⟨instr,Nat,itdicctrie¡⟩ es igual al indice en donde esta la rutina a la que tiene que saltar indicada por nombreRutina(instr).
10. Si OP(instr) es OREAD o OWRITE, itdicctrie dentro del vector⟨instr,Nat,itdicctrie⟩ apunta a la ventana de variable indicada por nombreVariable(instr).
11. El tercer elemento dentro de la tupla en ProgCalc es la longitud del vector⟨instr,Nat,ItDiccTrie(Variable,Ventana)⟩.
12. La variable dentro de la tupla de asignaciones esta en las claves de VarAsignacionActual o en las claves de VarVentana.
13. Para toda clave de VarAsignacionActual existe una tupla en Asignaciones donde el segundo elemento es igual a la clave.
14. Para toda variable de Asignaciones que pertenezca a VarVentana, si existe una tupla en la ventana de dicha variable tal que el Nat es igual al instanteActual o no existe una tupla cuyo nat es mayor a dicha a tupla, entonces el primer elemento de la ventana de esa clave esta la tupla que consiste en el instante actual y el Int de la tupla de asignaciones.
15. Para toda clave en varAsignacionActual, existe una tupla de Asignaciones tal que la clave=variable y el int de varAsignacionActual es igual al significado de la clave.
16. El tamaño de Asignaciones es igual a CantAsignaciones.
17. En inicio: Progama es igual a ProgCalc excepto que Programa no tiene ItDiccTrie(Variable,Ventana) y ProgCalc no tiene guardado la cantidad de rutinas que contiene.
18. El nat en inicio es mayor o igual a 0 y menor a la cantidad de rutinas en ProgCalc.
19. El iterador Instruccion esta apuntando a la instruccion de indice indicado por indiceInstruccion a la rutina indicado por indiceRutina.

1.3. Función de Abstracción

Como precondición, la calculadora c debe cumplir el invariante de representación.
La abstracción de c es C.
1.El primer elemento de la tupla inicio es igual a Programa(C).
2.El primer elemento de c.Progcalc[indiceRutinaActual] es igual a RutinaActual(C).
3.El indiceInstruccion es igual a IndiceInstrucciónActual(C).
4.La pila de c es igual a Pila(C).
5.El instanteActual de c es igual a InstanteActual(C).
6.Para toda clave v perteneciente a claves(varAsignacionesActual) existe una tupla t1 en Asignaciones donde v es igual al segundo elemento de t1, el significado de v en varActual es igual al tercer elemento de t1 y todas las demas tuplas t2 en Asignacionesque cumplen que v es igual al segundo elemento de t2 y el primer elemento de t1 es mayor o igual al primer elemento de t2 y por ultimo el tercer elemento de t1 es igual a ValorHistoricoVariable(c,v,t) o, si no existe la tupla t1 que cumpla todas las condiciones anteriores entonces ValorHistoricoVariable(c,v,t)=0 siendo t un instante.
7.Para toda clave v perteneciente a claves(varVentana), se puede obtener el valor de la variable en un instante n rehaciendo los mismos pasos que hizo la calculadora (ejecutar pasos hasta que este en el momento que se quiere saber el valor de la variable y asignando) y este valor es igual a ValorHistoricoVariable(c,v,n)

1.4. Algoritmos

InuevaCalculadora(**in** $Prog$: `Programa`, **in** r : `Rutina`, **in** w : `Nat`) → res : calc

1: $i \leftarrow 0$
2: $rutinasYinstrucciones \leftarrow$ IparaCalculadora(Prog)
3: $IteradorparaRutina \leftarrow$ CrearIt(rutinaYinstrucciones)
4: $indiceRutina \leftarrow 0$:
5: $ProgCalc \leftarrow \langle$ array<lista::Vacia()>[rutinas.size()] $\rangle$
6: **while** haySiguiente(IteradorparaRutina)==True **do**
7: $v \leftarrow \langle$ lista::Vacia() $\rangle$
8: $IteradorparaInstruccion \leftarrow$ CrearIT(IteradorparaRutina.siguiente)
9: $cantinstrucciones \leftarrow$ Siguiente(IteradorparaRutina).tamanioRutina;
10: **while** haySiguiente(IteradorParaInstruccion)==True **do**
11: **if** si la operacion de la instrucción no es OWRITE o OREAD o OJUMP o OJUMPZ **then**
12: agregarAtras(v,⟨ siguiente(IteradorParaInstruccion).Instruccion,0,NULL⟩);
13: **else**
14: **if** si la instruccion es un OREAD o OWRITE y no esta definido la variable a la que se le aplica OREAD o OWRITE en varVentana **then**
15: $ventana \leftarrow$ ventana::nuevaVentana;
16: $c \leftarrow \langle$ 0 $\rangle$
17: **while** $c < W$ **do**
18: registrar($ventana$,⟨0,0⟩);
19: c++;
20: **end while**
21: TDefinir(Varventanas,nombreVariable(siguiente(IteradorParaInstruccion).Instruccion),ventana);
22: $ITVAR \leftarrow$ ⟨Tsignificado(Varventana, NombreVariable(siguiente(IteradorParaInstruccion).Instruccion)).Crear
23: agregarAtras(v,⟨ siguiente(IteradorParaInstruccion).Instruccion,0,ITVAR⟩);
24: **else**
25: agregarAtras(v,⟨ siguiente(IteradorParaInstruccion).Instruccion,siguiente(IteradorParaInstruccion).Indice, NULL⟩);
26: **end if**
27: **end if**
28: avanzar(IteradorparaInstruccion)
29: **end while**
30: ProgCalc[i] ← v);
31: **if** $r = Ultimo(ProgCalc).Rut$ **then**
32: $indiceRutina \leftarrow$ i
33: $iterador \leftarrow$ CrearIT((ProgCalc[i]).Instrucciones);
34: **end if**
35: avanzar(IteradorParaRutina)
36: $i++$
37: **end while**
38: $Pila \leftarrow \langle$ Pila::Vacia() $\rangle$;
39: $IndiceInstruccion \leftarrow 0$;
40: $InstanteActual \leftarrow 0$;
41: $cantAsignaciones \leftarrow 0$;
42: $VarActual \leftarrow \langle$ diccTrie::TVacio $\rangle$;
43: $CapacidadVentana \leftarrow$W;
44: $Inicio \leftarrow \langle Programa, r \rangle$;
45: $Asignaciones \leftarrow \langle$ secu::Vacia() $\rangle$;
46: $res \leftarrow \langle$ InstanteActual,capacidadVentana,IndiceRutinaActual,IndiceInstrucción,Pila,ProgCalc,Asignaciones,cantAsignacion VarVentana,VarActual,Inicio,iterador $\rangle$

Complejidad: $\mathcal{O}(\#p.(|V| + |R|) + W.\#V)$
Justificación: La funcion primero usa la funcion IParaCalculadora que es $\mathcal{O}(\#Instruccionesdeprograma)$. Luego la funcion va a entrar en cada rutina y va a iterar segun la cantidad de instrucciones. Mientras itera en las funciones si encuentra una instruccion que contiene el nombre de una variable o de una rutina, va a copiar el nombre y el indice de la rutina en la que esta la rutina (si es un OJUMP o OJUMPZ) o va a copiar el nombre de la variable y definir (si no se definio antes) en un diccTrie ,usando como clave la variable, una ventana de Tamaño W. Copiar el nombre de la rutina o Variable es $\mathcal{O}(|R|)$ o $\mathcal{O}(|V|)$. Generar la ventana es $\mathcal{O}(W)$ y como depende si la variable ya se definio o no es $\mathcal{O}(W.\#V)$. Por lo tanto la complejidad es $\mathcal{O}(\#p.(|V| + |R|) + W.\#V)$

Iejecutando?(**in** c: `Calc`) $\rightarrow$ res : bool

1: $res \leftarrow$ (c.IndiceInstruccion<(c.ProgCalc[indiceRutinaActual].cantInstrucciones));

Complejidad: $\mathcal{O}(1)$
Justificación: La funcion devuelve true si el IndiceInstruccion es menor a la cantidad de instrucciones que tiene la rutina. Ver que se cumpla una condicion es $\mathcal{O}(1)$

InstruccionOJUMP(**in/out** c: `Calc`)

1: $c.indiceRutinaActual \leftarrow$ siguiente(c.instruccion).nat;
2: $c.indiceInstruccion \leftarrow 0$;
3: $c.instruccion \leftarrow$ crearIT((ProgCalc[indiceRutinaActual]).instrucciones)

Complejidad: $\mathcal{O}(1)$
Justificación: Modifica indiceRutinaActual y indiceInstrucción de tal manera que las proximas veces que se ejecute un paso se ejecuten instrucciones de la rutina indicada en la instrucción. Modificar variables tipo Nat es $\mathcal{O}(1)$.

InstruccionOWRITE(**in/out** c: `Calc`)

1: **if** El tamaño de la pila es 0 **then**
2: registrar(*((siguiente(c.ITinstruccion)).punteroAVentana),0);
3: **else**
4: registrar(*((siguiente(c.ITinstruccion)).punteroAVentana),tope(c.pila));
5: desapilar(c.pila);
6: **end if**

Complejidad: $\mathcal{O}(1)$
Justificación: Si el tamaño de la pila es 0, registra en la ventana de la variable un 0, sino registra el valor que esta al tope de la pila. Las operaciones tope, registrar tienen complejidad $\mathcal{O}(1)$

InstruccionOMUL(**in/out** c: `Calc`)

1: **if** El tamaño de la pila de la calculadora es 0 **then**
2: apilar(c.pila,0);
3: **else**
4: **if** El tamaño de la pila de la calculadora es 1 **then**
5: $c.pila \leftarrow$ apilar(Vacía(),0);
6: **else**
7: $b \leftarrow$ desapilar(c.pila)
8: $a \leftarrow$ desapilar(c.pila)
9: apilar(c.pila,a*b);
10: **end if**
11: **end if**

Complejidad: $\mathcal{O}(1)$
Justificación: Si la pila de la calculadora no tiene elementos apila un 0, si tiene 1 elemento lo saca y apila un 0 y si tiene 2 o mas desapila los 2 que estan en el tope de la pila y apila el resultado de la multiplicación entre ellos. Apilar,desapilar y multiplicar tienen complejidad $\mathcal{O}(1)$.

InstruccionOSUB(**in/out** c : `Calc`)

```
1: if El tamaño de la pila es 0 then
2:     apilar(c.pila,0);
3: else
4:     if El tamaño de la pila de la calculadora es mayor o igual a 2 then
5:         b ← desapilar(c.pila)
6:         a ← desapilar(c.pila)
7:         apilar(c.pila,a-b);
8:     else
9:         b ← desapilar(c.pila)
10:        apilar(c.pila,-b);
11:    end if
12: end if
```

Complejidad: $\mathcal{O}(1)$
Justificación: La funcion apila un 0 en la pila de la calculadora si no hay ningun elemento, si hay 1 elemento en la pila lo multiplica por -1 y si hay 2 o mas desapila los 2 valores que estan en el tope de la pila y apila el resultado de la resta de el segundo elemento - el primer elemento. La complejidad de apilar/desapilar/restar es $\mathcal{O}(1)$

InstruccionOADD(**in/out** c : `Calc`)

```
1: if El tamaño de la pila es 0 then
2:     apilar(c.pila,0);
3: else
4:     if El tamaño de la pila de la calculadora es mayor o igual a 2 then
5:         b ← desapilar(c.pila)
6:         a ← desapilar(c.pila)
7:         apilar(c.pila,a+b);
8:     end if
9:     indiceInstruccion++;
10: end if
```

Complejidad: $\mathcal{O}(1)$
Justificación: La funcion apila un 0 en la pila de la calculadora si no hay ningun elemento, si hay 1 elemento en la pila no hace nada y si hay 2 o mas desapila los 2 primeros y los agrega sumados. La complejidad de apilar/desapilar es $\mathcal{O}(1)$

IEjecutarUnPaso(**in/out** c : `Calc`)

```
1: if OP(*(c.ITinstruccion)) == OADD then
2:     InstruccionOADD(&c);
3:     indiceInstruccion++;
4:     avanzar(c.ITinstruccion)
5: else
6:     if OP(*(c.ITinstruccion)) == OSUB then
7:         InstruccionOSUB(&c);
8:         indiceInstruccion++;
9:         avanzar(c.ITinstruccion)
10:    else
11:        if OP(*(c.ITinstruccion)) == OMUL then
12:            InstruccionOMUL(&c);
13:            indiceInstruccion++;
14:            avanzar(c.ITinstruccion)
15:        else
16:            if OP(*(c.ITinstruccion)) == OPUSH then
17:                apilar(c.pila,constanteNumerica((siguiente(c.ITinstruccion)).Instruccion);
18:                indiceInstruccion++;
19:                avanzar(c.ITinstruccion)
20:            else
21:                if OP(*(c.ITinstruccion)) == OWRITE then
22:                    InstruccionOWRITE(&c);
23:                    indiceInstrucción++;
24:                    avanzar(c.ITinstruccion)
25:                else
26:                    if OP(*(c.ITinstruccion)) == OREAD then
27:                        apilar(c.pila,*((siguiente(c.instruccion)).punteroAventana)[c.capacidadVentana -1]);
28:                        indiceInstrucción++;
29:                        avanzar(c.ITinstruccion)
30:                    else
31:                        if OP(*(c.ITinstruccion)) == OJUMP then
32:                            InstruccionOJUMP(&c);
33:                        else
34:                            if OP(*(c.ITinstruccion)) == OJUMPZ then
35:                                if el tamaño de la pila es 0 o el valor que esta al tope de la pila es 0 then
36:                                    InstruccionOJUMP(&c);
37:                                else
38:                                    indiceInstrucción++;
39:                                    avanzar(c.ITinstruccion)
40:                                end if
41:                            end if
42:                        end if
43:                    end if
44:                end if
45:            end if
46:        end if
47:    end if
48: end if
```

Complejidad: $\mathcal{O}(1)$

Justificación: La funcion realiza la instrucción. Desapilar, apilar, modificar ints,nats o ventanas, ver que se cumpla la guarda del if es en $\mathcal{O}(1)$.Aclaracion: usar la función InstruccionOX siendo X SUB, MUL, ADD o WRITE se pasa la posición de memoria de la calculadora para no aumentar la complejidad de las funciones.

IValorHistoricoVariable(**in** c : `Calc`, **in** v : `variable`, **in** t : `Nat`) $\rightarrow$ res : Int

```
1: if V esta definido en c.varVentana then
2:     L ← 0;
3:     R ← c.capacidadVentana -1;
4:     if t>((Tobtener(c.varVentana,v))[L].instante) then
5:         ITVEN ← CrearPuntero(Tobtener(c.varVentana,v))
6:         while ITVEN[L]<ITVEN[R] do
7:             M ← L+((R-L)/2);
8:             if (ITVER[M]).valor>T then
9:                 R ← M;
10:            else
11:                L ← M;
12:            end if
13:        end while
14:        res ← (*(ITVEN)[L]).valor;
15:    else
16:        nc ← nuevaCalculadora(π(1)(c.Inicio),π(2)(c.Inicio),c.CapacidadVentana);
17:        i ← 0;
18:        A ← 0;
19:        while i<=t do
20:            if c.cantidadAsignaciones es estrictamente mayor a A luego π(1)(c.asignaciones)[a]==i then
21:                AsignarVariable(nc,π(2)(c.asignaciones)[a],π(3)(c.asignaciones[A]))
22:                A++;
23:            end if
24:            EjecutarUnPaso(nc);
25:            i++;
26:        end while
27:        res ← TSignificado(nc.VarVentana,v)[L];
28:    end if
29: else
30:
31:    if si c.cantidadAsignaciones es igual a 0 then
32:        res ← 0;
33:    elsenc ← nuevaCalculadora(π(1)(c.Inicio),π(2)(c.Inicio),c.CapacidadVentana);
34:        i ← 0;
35:        res ← 0;
36:        while i<c.cantAsignaciones do
37:            if la variable de c.asignaciones[i]==v then
38:                if el Nat de c.asignaciones[i]¡= t then
39:                    res ← π(3)(c.asignaciones[A]);
40:                end if
41:            end if
42:            i++;
43:        end while
44:    end if
45: end if
```

Complejidad: $\mathcal{O}(\#p.(|V| + |R|) + W.\#V + T + |Asignaciones|)$ donde #p es la cantidad de instrucciones en el programa, $|V|$ es la longitud del nombre de la variable, $|R|$ es la longitud del nombre de la rutina, W es el tamaño de la ventana, #V es la cantidad de variables diferentes, T el tiempo actual de la calculadora y $|Asignaciones|$ la cantidad de asignaciones que se hizo en la calculadora.

Justificación: La funcion busca si esta definido en el dicctree VarVentana que tiene complejidad $\mathcal{O}(|V|)$,si esta revisa que el tiempo pasado por parametro esta en la ventana, revisando si el tiempo es mayor a lo ultimo que el tiempo que guardo la ventana. Si esta, lo busca en la ventana haciendo busqueda binaria que tiene complejidad $\mathcal{O}(logW)$ siendo W la longitud de la ventana, por lo tanto si la variable existio en el codigo fuente del programa y el tiempo esta en la ventana tiene complejidad $\mathcal{O}(|V| + logW)$. Si el tiempo no pertenece a la ventana, rehace la calculadora y ejecuta los pasos/asignaciones hasta estar en el tiempo del que se desea saber el valor de una variable y devuelve el valor de dicha variable lo cual tiene complejidad $\mathcal{O}(\#p.(|V| + |R|) + W.\#V + T)$. Si no esta en el dicctree VarVentana, devuelve 0 si la cantidad de asignaciones es igual a 0. Sino busca la tupla en asignaciones con el Nat mas alto, variable igual a v y devuelve el Int de la tupla lo cual tiene complejidad $\mathcal{O}(|Asignaciones|)$, siendo $|Asignaciones|$ la cantidad de asignaciones que se hizo en la calculadora

IAsignarVariable(**in/out** c: `Calc`, **in** v: `Variable`, **in** i: `Int`)

1: agregarAtras(c.Asignaciones,⟨c.InstanteActual,v,i⟩);
2: cantAsignaciones++;
3: **if** La variable v esta definido en c.varVentana **then**
4: registrar(significado(c.varVentana,v),i);
5: **else**
6: definir(c.varActual,x,i);
7: **end if**

Complejidad: $\mathcal{O}(|v|)$
Justificación: La funcion agrega en el vector asignaciones de la calculadora una tupla que indica en que momento se hizo la asignación,a que variable y que valor. Copiar el nombre de una variable es de complejidad $\mathcal{O}(|v|)$. Despues revisa si esta definido en el dicctrie varVentana y si lo esta registra el valor i en la ventana. Registrar es de complejidad $\mathcal{O}(|1|)$ y ver si esta definido y buscar el significado la variable en varVentana es de complejidad $\mathcal{O}(|v|)$. Si no esta definido en dicho diccionario lo define en el diccionario varActual, lo cual tambien es de complejidad $\mathcal{O}(|v|)$.

IIndiceInstrucciónActual(**in** c: `Calc`) → res : Nat

1: $res \leftarrow$ (c.IndiceInstrucción);

Complejidad: $\mathcal{O}(1)$
Justificación: La funcion devuelve el Indice de la instruccion en el momento actual de la calculadora. Copiar un Nat es de complejidad $\mathcal{O}(1)$.

IInstanteActual(**in** c: `Calc`) → res : Nat

1: $res \leftarrow$ (c.InstanteActual);

Complejidad: $\mathcal{O}(1)$
Justificación: La funcion devuelve el Instante actual de la calculadora. Copiar un Nat es de complejidad $\mathcal{O}(1)$.

IRutinaActual(**in** c: `Calc`) → res : String

1: $res \leftarrow \pi(1)$(ProgCalc[indiceRutinaActual]);

Complejidad: $\mathcal{O}(1)$
Justificación: La funcion devuelve el nombre de la rutina que se esta ejecutando en la calculadora, buscandolo en progCalc con el indiceRutinaActual. Devolver directamente un valor es $\mathcal{O}(1)$.

IPila(**in** c: `Calc`) → res : String

1: $res \leftarrow$ c.pila);

Complejidad: $\mathcal{O}(1)$
Justificación: La funcion devuelve la pila de la calculadora.

1.5. Servicios usados

Se utilizan los Tads Programa,Instruccion, Diccionario, y Ventana tanto para explicar el modulo Calculadora como para su pseudocodigo.

Del modulo Instruccion se usan las operaciones: OP, constanteNumerica, nombreVariable y nombreRutina el cual todas estas funciones tienen complejidad $\mathcal{O}(1)$ excepto nombreVariable que tiene complejidad $\mathcal{O}(|V|)$. y nombreRutina que tiene complejidad $\mathcal{O}(|R|)$

Del modulo Programa se usa la operacion IParaCalculadora que devuelve todas las rutinas y sus instrucciones en $\mathcal{O}(1)$.

Del modulo DiccTrie se usa las operaciones: Tdefinir, Tdefinido?, y Tobtener el cual tienen complejidad $\mathcal{O}(|V|)$ siendo $|V|$ el largo del nombre de la variable.

Del modulo Ventana se usa la operación registrar cuya complejidad es en $\mathcal{O}(1)$

2. Módulo Programa

Interfaz

Parametros formales
Generos: Programa
Se explica con: Secu, String, Nat, Bool, Instrucción
Operaciones: NUEVOPROGRAMA() $\rightarrow res$: `Programa`
Pre $\equiv$ {true}
Post $\equiv$ {$\widehat{res} =_{\text{obs}}$ nuevoPrograma}
Complejidad: $\Theta(1)$
Descripción: Crea un nuevo programa

IAGINSTRUCCIÓN(**in/out** p: `programa`, **in** r: `string`, **in** i: `instrucción`)
Pre $\equiv$ {$p = p_0$}
Post $\equiv$ {$\widehat{p} =_{\text{obs}}$ agInstrucción($\widehat{p}$,$\widehat{r}$,$\widehat{instruccion}$)}
Complejidad: $\Theta(|p|.|r|)$
Descripción: Agrega una instruccion a una rutina especifica

IRUTINAS(**in** p: `programa`) $\rightarrow res$: `conj`$\langle$`string`$\rangle$
Pre $\equiv$ {true}
Post $\equiv$ {$\widehat{res} =_{\text{obs}}$ Rutinas($\widehat{p}$)}
Complejidad: $\Theta(|p|)$
Descripción: Devuelve el nombre de todas las rutinas del programa.

ILONGITUD(**in** p: `Programa`, **in** r: `String`) $\rightarrow res$: `Nat`
Pre $\equiv$ {$\widehat{r} \in$ rutinas($\widehat{p}$)}
Post $\equiv$ {$\widehat{res} =_{\text{obs}}$ longitud($\widehat{p}$,$\widehat{r}$))}
Complejidad: $\Theta(|p|)$
Descripción: Devuelve la cantidad de instrucciones que tiene la rutina indicada por parametro en el programa.

IINSTRUCCIÓN(**in/out** p: `programa`, **in** r: `string`, **in** n: `Nat`) $\rightarrow res$: `Instrucción`
Pre $\equiv$ {$\widehat{r} \in$ rutinas($\widehat{p}$) $\wedge$ n $\in$ longitud($\widehat{p}$,$\widehat{r}$)}
Post $\equiv$ {$\widehat{res} =_{\text{obs}}$ instrucción($\widehat{p}$,$\widehat{r}$,$\widehat{n}$)}
Complejidad: $\Theta(|p|)$
Descripción: Devuelve la instrucción cuyo indice es n de la rutina r indicado en los parametros.

IPARACALCULADORA(**in** p: `Programa`, **in** r: `String`, **in** n: `Nat`) $\rightarrow res$: `Lista`$\langle$`Tuple`$\langle$`String,secu(`$\langle$`Instruccion, Nat` $\rangle$`),Nat`$\rangle\rangle$
Pre $\equiv$ {true}
Post $\equiv$ {longitud(p)=longitud(res) $\wedge$ ($\forall$ r:rutina) (r $\in$ rutinas($\widehat{p}$) $\longrightarrow_L$ ($\exists$ t:Tuple$\langle$ String,vector($\langle$ Instruccion, Indice:Nat $\rangle$), Nat $\rangle$) ($\exists$ i:Nat) (0<=i<longitud(res) $\wedge$ t=res[i] $\wedge$L $\Pi(1)$(res[i]) =r $\wedge$ $\Pi(3)$(res[i])=longitud($\Pi(2)$(res)) $\wedge$L
($\forall$ j:Nat) (0<=j< $\Pi(3)$(res[i]) $\longrightarrow_L$ (($\Pi(2)$(res[i]))[j])=Instruccion(p,r,j) $\wedge$ (Instruccion(p,r,j)=OJUMP $\vee$ Instruccion(p,r,j)=OJUMPZ) $\longrightarrow_L$ $\Pi(1)$(res[($\Pi(2)$($\Pi(2)$(res[i]))[j])])=nombreRutina(Instruccion(p,r,j))}
Complejidad: $\Theta(1)$
Descripción: Devuelve el programa en un formato adecuado para poder generar la calculadora

Representación

2.1. Representacion de Programa

`Programa` **se representa con** `Prog`

donde Prog es Tupla:
$\langle$ listProg: vector(tuple$\langle$ Rut:Rutina,instRut: vector ($\langle$ instruccion: Instr, Indice: Int $\rangle$), tamanioRutina:nat $\rangle$), rutinasAunNoExistente: vector(tuple$\langle$nombreRut:string,lista:list(puntero(Instruccion))$\rangle$) $\rangle$

2.2. Invariante de Representación

1. Para toda tupla dentro de listProg, tamanioRutina es igual a instrRut.size() de la misma tupla.

2. Para toda tupla dentro de listProg, no existe 2 tuplas que tienen el mismo Rut.

3. Para todo indice en listProg, indice es mayor o igual a 0 y menor a la cantidad de rutinas que tiene listProg (listprog.size())

4. Si la instruccion es un OJUMP o OJUMPZ, los indices indican en que posicion de listPRog esta la rutina con el nombre que esta en la instruccion. Si la rutina aun no existe en listProg, entonces el indice es igual a -1. **5.** Para todo nombreRut en rutinasAunNoExistente, no existe tupla en listProg tal que Rut = nombreRut. **6.** Para toda instruccion en listProg tal que sea OJUMP O OJUMPZ y la rutina a la que salte no este en listProg, existe una tupla en rutinasAunNoExistente tal que nombreRut sea igual a la rutina a la que salte dicha instruccion y en lista este un puntero que apunta a dicha instruccion.

2.3. Función de Abstrsacción

Como precondición, el programa p debe cumplir el invariante de representación.

La abstracción de p es P.

1. Todas las rutinas(rut) en listProg deben pertenecer a Rutinas(p) y para toda rutina perteneciente a Rutinas(p), esta debe estar en listProg.

2. Para toda rutina r en listProg, tamanioRutina es igual a longitud(p,r).

3. Para toda rutina r en listProg que esta en el indice i de listProg y para todo 0<=j<tamanioRutina (tambien en el indice i), la instruccion en el indice j es igual a Instrucción(p,$\widehat{r}$,$\widehat{j}$).

2.4. Algoritmos

---

iNuevoPrograma() $\rightarrow$ *res* : Prog

1: *res* $\leftarrow$ ⟨ vector::Vacia(), vector::Vacia() ⟩

Complejidad: $\Theta(1)$

---

iRutinas(**in** p : `Prog`) $\rightarrow$ *res* : vector⟨rutina⟩

1: *res* $\leftarrow$ ⟨ vector::Vacia()⟩
2: **for** $i = 0$ to longitud(p.listprog) **do**
3: agregarAtras(res,longitudop.listProg[i].Rut)
4: i++;
5: **end for**

Complejidad: $\Theta(|listProg| * ((long(res)) + (|r|)))$
Justificación: Recorrer el el vector listProg es $\Theta(|listProg|)$ y mientras se recorre se guarda en un vector el nombre de la rutina, lo cual copiar el nombre de la rutina es $\Theta(|r|)$ y en caso de tener que copiar res a otra posicion de la memoria para poder seguir guardando valores uno al lado de otro es $\Theta(long(res))$.

---

iLongitud(**in** p : `Prog`, **in** r : `Rutina`) $\rightarrow$ *res* : Nat

1: *res* $\leftarrow$ 0
2: **for** $i = 0$ to p.cantRutina **do**
3: **if** p.listProg[i].Rut==r **then**
4: *res* $\leftarrow$ p.listProg[i].tamanioRutina;
5: **end if**
6: i++;
7: **end for**

Complejidad: $\Theta(|p|)$
Justificación: Recorrer listProg es complejidad $\Theta(|listProg|)$ y copiar un Nat es $\Theta(1)$.

---

iInstrucción(**in** p : `Prog`, **in** r : `Rutina`, **in** n : `Int`) $\rightarrow$ res : Instrucción

```
1: for i = 0 to p.cantRutina do
2:     if p.listProg[i].Rut==r then
3:         res ← p.listProg[i].instRut[n].instr;
4:     end if
5:     i++;
6: end for
```

Complejidad: $\Theta(|p|)$
Justificación: Recorrer listProg es complejidad $\Theta(|listProg|)$ y copiar la instrucción a res es complejidad $\Theta(|s|)$ donde s es la longitud del nombre de la rutina o variable porque copiar los NAT es $\Theta(1)$, copiar la operacion(que es ENUM) es $\Theta(1)$ y copiar la longitud del nombre de la rutina o variable es $\Theta(|s|)$.

IParaCalculadora(**in** p : `Prog`) $\rightarrow$ res : Lista$\langle$ Rut:Rutina,instRut: vector ($\langle$ instruccion: Instr, Indice: nat $\rangle$), tamanioRutina:nat $\rangle\rangle$

```
1: for i = 0 to p.cantRutina do
2:     AgregarAtras(res,⟨p[i].Rut,lista::Vacia(),p.tamanioRutina⟩)
3:     for j = 0 to p[i].tamanioRutina do
4:         AgregarAtras(Ultimo(res).instRut[j])
5:         j++;
6:     end for
7:     i++;
8: end for
```

Complejidad: $\Theta(\#Instrucciones)$
Justificación: La funcion recorre cada instruccion de la rutina.

iAgregarInstruccion(**in/out** p: `prog`, **in** r: `string`, **in** i: `instr`)

```
1: BOOL existe ← False
2: indiceRut ← 0
3: for i = 0 to p.Π(1) do
4:     if p.listProg[i].Π (0) ==r then
5:         indiceRut ← i
6:         existe ← True
7:     end if
8: end for
9: if existe then
10:    if i.op() == oJUMP ∨ i.op() == oJUMPZ then
11:        BOOL esta ← False
12:        indiceAux ← 0
13:        for i = 0 to p.Π(1) do
14:            if p.listProg (0)[i].Π (0) == i.nombreRutina() then
15:                esta ← True
16:                indiceAux ← i
17:            end if
18:        end for
19:        if esta then
20:            p.listProg[indiceRut].Π(1).AgregarAtras(⟨i, indicdeAux⟩)
21:            p.listProg[indiceRut].Π(2) + +
22:            iAgregarAunNoExiste(p,i,indiceRut)
23:        else
24:            p.listProg.AgregarAtras(⟨ i.nombreRutina(), vector.vacia() ,0⟩)
25:            p.listProg[IndiceRut].Π(1).AgregarAtras(⟨i, p.cantRutina − 1⟩)
26:        end if
27:    else
28:        p.listProg[indiceRut].Π(1).AgregarAtras(⟨i, indicdeRut⟩)
29:        p.listProg[indiceRut].Π(2) + +
30:    end if
31: else
32:    if i.op() == oJUMP ∨ i.op() == oJUMPZ then
33:        p.listProg.AgregarAtras(⟨ r, vector.vacia() ,0⟩)
34:        BOOL esta ← False
35:        indiceAux ← 0
36:        for i = 0 to p.cantRutina − 1 do
37:            if p.listProgi[i].Π (0) == i.nombreRutina() then
38:                esta ← True
39:                indiceAux ← i
40:            end if
41:        end for
42:        if esta then
43:            p.listProg[p.cantRutina − d1].Π(1).AgregarAtras(⟨i, indicdeAux⟩)
44:            p.listProg[p.cantRutina − 1].Π(2) + +
45:        else
46:            p.listProg[p.cantRutina − 1].Π(1).AgregarAtras(⟨i, −1⟩)
47:            iAgregarAunNoExiste(p,i,longitud(p.listProg)-1)
48:        end if
49:    else
50:        p.listProg[p.cantRutina − 1].Π(1).AgregarAtras(⟨i, indiceRut⟩)
51:        p.listProg[p.cantRutina − 1].Π(2) + +
52:    end if
53:    iBuscarEnAunNoExiste(p,r,i)
54: end if
```

Complejidad: $\Theta(|P|.|R| + \sum_{r \in rutinas(p)} longitud(p, r))$
Justificacion:

iBuscarEnAunNoExiste(**in/out** p : `prog`, **in** r : `Rutina`, **in** *indiceRut* : `Nat`)

```
1: i ← 0;
2: while i < longitud(p.rutinasAunNoExistente) Y encontrado == False do
3:     if p.rutinasAunNoExistente[i].nombreRut == r then
4:         encontrado = true
5:         iterador ← CrearIT(p.rutinasAunNoExistente[i].lista)
6:         while iterator != CrearITUlt(p.rutinasAunNoExistente[i].lista) do
7:             *(siguiente(iterator)).Indice = longitud(p.listProg)-1;
8:             avanzar(iterator)
9:         end while
10:    end if
11:    i++;
12: end while
```

Complejidad: $\Theta(|p| + |R|)$
Justificacion: La funcion recorre p.rutinasAunNoExistente

---

iAgregarAunNoExiste(**in/out** p : `prog`, **in** i : `instr`, **in** *indiceRut* : `Nat`)

```
1: i ← 0;
2: encontrado ← False
3: while i ¡p.rutinasAunNoExistente Y encontrado == False do
4:     if p.rutinasAunNoExistente[i].nombreRut == nombreRutina(i) then
5:         encontrado = true
6:         AgregarAtras(p.rutinasAunNoExistente[i].lista,punteroA(p.listProg[indiceRut][tamanioRut-1]);
7:     end if
8:     i++;
9: end while
10: if encontrado == false then
11:     AgregarAtras(p.rutinasAunNoExistente,¡nombreRutina(i),lista::Vacia¿)
12:     AgregarAtras(p.rutinasAunNoExistente[rutinasAunNoExistente.size()-1].lista,punteroA(p.listProg[indiceRut][tamanioRu
    1]);
13: end if
```

Complejidad: $\Theta(|p| + |R|)$

3. Modulo Instrucción

Interfaz

Parametros formales
Generos: instruccion
Se explica con: VARIABLE, RUTINA, OPERACION, INT

*I*PUSH(**in** n : : nat)→res: instrucción
Pre{True}
Post{Op($\widehat{res}$) $=_{obs}$ OPush $\wedge_L$ constanteNumerica($\widehat{res}$) $=_{obs}$ $\widehat{n}$ }
Complejidad: $O(1)$
Descripcion: Genera la instruccion con la operacion Push.

*I*ADD()→res: instruccion
Pre{True}
Post{Op($\widehat{res}$) $=_{obs}$ OAdd}
Complejidad: $O(1)$
Descripcion: Genera la instruccion con la operacion Add.

*I*SUB()→res: instruccion
Pre{True}
Post{Op($\widehat{res}$) $=_{obs}$ OSub}
Complejidad: $O(1)$
Descripcion: Genera la instruccion con la operacion Sub.

*I*ADD()→res: instruccion
Pre{True}
Post{Op($\widehat{res}$) $=_{obs}$ OAdd}
Complejidad: $O(1)$
Descripcion: Genera la instruccion con la operacion Add.

*I*MULL()→res: instruccion
Pre{True}
Post{Op($\widehat{res}$) $=_{obs}$ OMull}
Complejidad: $O(1)$
Descripcion: Genera la instruccion con la operacion Mull.

*I*READ(**in** v : : variable)→res: instruccion
Pre{True}
Post{Op($\widehat{res}$) $=_{obs}$ ORead $\wedge_L$ nombreVariable($\widehat{res}$) $=_{obs}$ $\widehat{v}$}
Complejidad: $O(|v|)$
Descripcion: Genera la instruccion con la operacion Read.

*I*WRITE(**in** v : : variable)→res: instruccion
Pre{True}
Post{Op($\widehat{res}$) $=_{obs}$ OWrite $\wedge_L$ nombreVariable($\widehat{res}$) $=_{obs}$ $\widehat{v}$}
Complejidad: $O(|v|)$
Descripcion: Genera la instruccion con la operacion Write.

*I*JUMP(**in** r : : rutina)→res: instruccion
Pre{True}
Post{Op($\widehat{res}$) $=_{obs}$ OJump $\wedge_L$ nombreRutina($\widehat{res}$) $=_{obs}$ $\widehat{r}$}
Complejidad: $O(|r|)$
Descripcion: Genera la instruccion con la operacion Jump.

*I*JUMPZ(**in** r : : rutina)→res: instruccion
Pre{True}

Post{Op($\widehat{res}$) $=_{obs}$ OJumpZ $\wedge_L$ nombreRutina($\widehat{res}$) $=_{obs}$ $\widehat{r}$}
Complejidad: $O(|r|)$
Descripcion: Genera la instruccion con la operacion JumpZ.

OP(**in** i : : instruccion)→res: ENUM
Pre{True}
Post{($\widehat{res}$) $=_{obs}$ Op($\widehat{i}$)}
Complejidad: $O(1)$
Descripcion: Devuelve la operacion de la instruccion.

constanteNumerica(**in** i : : instruccion)→res: int
Pre{Op($\widehat{i}$) $=_{obs}$ OPush}
Post{constanteNumerica($\widehat{i}$ $=_{obs}$ $\widehat{res}$)}
Complejidad: $O(1)$
Descripcion: Devuelve el parametro de la operacion Push.

nombreVariable(**in** i : : instruccion)→res: string
Pre{Op($\widehat{i}$) $=_{obs}$ ORead $\vee$ Op($\widehat{i}$) $=_{obs}$ OWrite}
Post{nombreVariable($\widehat{i}$) $=_{obs}$ $\widehat{res}$}
Complejidad: $O(1)$
Descripcion: Devuelve el parametro de la operacion Read o Write.

nombreRutina(**in** i : : instruccion)→res: string
Pre{Op($\widehat{i}$) $=_{obs}$ OJump $\vee$ Op($\widehat{i}$) $=_{obs}$ OJumpZ}
Post{nombreRutina($\widehat{i}$) $=_{obs}$ $\widehat{res}$}
Complejidad: $O(1)$
Descripcion: Devuelve el parametro de la operacion OJump o OJumpZ.

Representación

3.1. Representación de instrucción

instruccion es *instr* donde *instr* se representa con
Tupla⟨op: ENUM, s: STRNG, n: NAT⟩

3.2. Invariante de representación

Rep: $\widehat{instr}$ i →bool
Rep(i)≡
(i.Op $=_{obs}$ OAdd $\vee$ i.Op $=_{obs}$ OSub $\vee$ i.Op $=_{obs}$ OMul) →(i.s $=_{obs}$ $\langle\rangle$ $\wedge$ i.n $=_{obs}$ 0) $\wedge$ (i.Op $=_{obs}$ OPush →i.s $=_{obs}$ $\langle\rangle$)
$\wedge$ (i.Op $=_{obs}$ ORead $\vee$ i.Op $=_{obs}$ OWrite $\vee$ i.Op $=_{obs}$ OJump $\vee$ i.Op $=_{obs}$ OJumpZ) →i.n = 0

3.3. Función de Abstracción

Abs: $\widehat{instr}$ i→instruccion {Rep(i)}
Abs(i)≡ I/ Op(I) $=_{obs}$ i.Op $\wedge_L$ ((Op(I) $=_{obs}$ OPush $\longrightarrow_L$ constanteNumerica(I) $=_{obs}$ i.n) $\vee$ (Op(I)ORead $\vee$ Op(I) $=_{obs}$ OWrite $\longrightarrow_L$ nombreVariable(I) $=_{obs}$ i.s) $\vee$ (Op(I) $=_{obs}$ OJump $\vee$ Op(I) $=_{obs}$ OJumpZ $\longrightarrow_L$ nombreRutina(I) $=_{obs}$ i.s))

3.4. Algoritmos

I*I*PUSH(**in** n : : nat)→res: instr
string s ← vacia
res ← ⟨OPush, s, n⟩

I*I*ADD()→res: instr
string s ← vacia
res ← ⟨OAdd, s, 0⟩

I*I*SUB()→res: instr
string s ← vacia
res ← ⟨OSub, s, 0⟩

I*I*MUL()→res: instr
string s ← vacia
res ← ⟨OMul, s, 0⟩

I*I*READ(**in** s : : string)→res: instr
res ← ⟨ORead, s, 0⟩

I*I*WRITE(**in** s : : string)→res: instr
res ← ⟨OWrite, s, 0⟩

I*I*JUMP(**in** s : : string)→res: instr
res ← ⟨OJump, s, 0⟩

I*I*JUMPZ(**in** s : : string)→res: instr
res ← ⟨OJumpZ, s, 0⟩

OP(in i **:** `instr`**)** → res ENUM

1: res ← i.op

Complejidad: $\mathcal{O}(1)$
Justificación: Devolver un valor del tipo ENUM que esta dentro la instruccion i es $\mathcal{O}(1)$

---

constanteNumerica(in i **:** `instr`**)** → res int

1: res ← i.n

Complejidad: $\mathcal{O}(1)$
Justificación: Devolver un valor del tipo int que esta dentro la instruccion i es $\mathcal{O}(1)$

---

nombreVariable(in i **:** `instr`**)** → res string

1: res ← i.s

Complejidad: $\mathcal{O}(1)$
Justificación: Devolver el string dentro de la instruccion i es $\mathcal{O}(1)$

---

nombreRutina(in i **:** `instr`**)** → res string

1: res ← i.s

Complejidad: $\mathcal{O}(1)$
Justificación: Devolver el string que esta dentro de la instruccion i es $\mathcal{O}(1)$

---

4. Módulo Dicc Trie

Interfaz

Se explica con: DICCIONARIO(STRING,α)
Géneros: `DiccString(`α`)`.

Operaciones:

VACIO() $\rightarrow$ res : `diccString(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ Vacio} Genera un dicc vacio.

DEFINIR(**in/out** d: `diccstring(`α`)`, **in** p: `string`, **in** a: α) $\rightarrow$ res : `diccstring`
Pre $\equiv$ {$d = d_0$}
Post $\equiv$ {$\widehat{d} =_{\text{obs}}$ definir($\widehat{p}$, $\widehat{a}$, $\widehat{d}$)}
Complejidad: $O(|p| + copiar(a))$
Descripción: Define a en d con la clave p. El elemento a se define por copia

DEF?(**in** p: `string`, **in** d: `dicctring(`α`)`) $\rightarrow$ res : `Bool`
Pre $\equiv$ {true}
Post $\equiv$ {$res =_{\text{obs}}$ def?($\widehat{p}$, $\widehat{d}$)}
Complejidad: $O(|p|)$
Descripción: Devuelve true sii la p tiene una definicion en d

OBTENER(**in** p: `string`, **in** d: `dicctring(`α`)`) $\rightarrow$ res : α
Pre $\equiv$ {def?($\widehat{p}$, $\widehat{d}$)}
Post $\equiv$ {alias ($res =_{\text{obs}}$ obtener($\widehat{p}$, $\widehat{d}$))}
Complejidad: $O(|p|)$
Descripción: La complejidad toma como complejidad la entrada p con largo mas largo
Aliasing: Devuelve el siguiente de la clave p en d. Res se modifica sii el d se modifica

Representación

En este modulo usamos un Trie para definir un diccionario en el cual las claves son strings

La idea es que la complejidad de definir y obtener no dependa de la cantidad de claves, si no de la longitud de la clave que definimos.

`Estructura` **se representa con** `estr`

donde `estr` es `tupla`(*raiz*: `puntero(Nodo)`
)
donde `Nodo` es `tupla`(*definidos*: `puntero(`α`)`
, *siguientes*: `Array(puntero(Nodo))`
)

4.1. Invariante de representacion

.Dado un nodo N, ningun elemento dentro de siguientes de N puede apuntar a un Nodo que se uso para llegar a N ni a un nodo que apunta algunos de estos.// **.**Definidos no apunta a un valor que otro nodo esta apuntando al mismo tiempo.

4.2. Función de abstracción

Como precondición, el programa p debe cumplir el invariante de representación.

La abstracción de d es D. **.**Dado una clave c, si c esta definido en d, el nodo N que se obtiene al recorrer D segun la clave tiene a definidos no apuntando a NULL.
.Dado una clave c, si c esta definido en d, entonces la definicion de c es igual a lo que apunta el nodo N que se obtiene al recorrer D segun la clave.

4.3. Algoritmos

iVacio() $\rightarrow res$ **dicctring(**α**)**

1: Puntero(nodo) $raiz \leftarrow$ NULL
2: $res \leftarrow$ raiz

Complejidad: $\mathcal{O}(1)$
Justificación: Generar un puntero que apunta a NULL es $\mathcal{O}(1)$

---

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

1: Puntero(Nodo) $actual \leftarrow$ raiz
2: $i \leftarrow 0$
3: **while** i < long(p) **do**
4: $actual \leftarrow$ actual.siguientes[ord(p[i])]
5: $i \leftarrow$ i+1
6: **end while**
7: $res \leftarrow$ actual.definicion

Complejidad: $\mathcal{O}(|p|)$
Justificación: Siendo la complejidad, la longitud de la palabra mas larga sea cual sea p

---

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

1: Puntero(Nodo) $actual \leftarrow$ raiz
2: $i \leftarrow 0$
3: $nueva \leftarrow$ false
4: **while** i $\langle$ long(p) **do**
5: **if** actual.siguientes[ord(p[i])] == NULL **then**
6:
7: $nuevoNodo \leftarrow$ new Nodo;
8: actual.siguientes[ord(p[i])] $\leftarrow$ nuevoNodo;
9: $nueva \leftarrow$ true;
10: **end if**
11: $i \leftarrow$ i+1
12: **end while**
13: **if** actual.definicion $\neq$ NULL **then**
14: actual.definicion $\leftarrow$ NULL
15: **end if**
16: actual.definicion $\leftarrow$ copiar(a)
17: **if** nueva **then**
18: actual.it.claves $\leftarrow$ claves.AgregarRapido(p)
19: **end if**

Complejidad: $\mathcal{O}(|p| + copiar(a))$
Justificación: El while hace $|p|$ iteraciones siendo p la longitud de la palabra lo cual esto es $\mathcal{O}(|p|)$. Generar el nodo es $\mathcal{O}(1)$ ya que el arreglo esta acotado por 256. Copiar el valor a depende de que tipo de dato es, con lo cual esto es $\mathcal{O}(copiar(a))$.

---

iDef?(**in** p: `string` **in/out** d: `diccTrie`(α) $\rightarrow$ res Bool

```
1:  i ← 0
2:  if raiz ==NULL then
3:      res ← false;
4:  else
5:      esta ← true
6:      puntero(nodo)actual ← raiz
7:      while i ⟨ long(p) and esta do
8:          if actual.siguientes(ord(p[i]))== NULL then
9:              esta ← false
10:         else
11:             actual ← actual.siguientes(ord(p[i]))
12:             i ← i+1
13:         end if
14:     end while
15:     res ← esta
16: end if
```

Complejidad: $\mathcal{O}(|p|)$
Justificación: El while hace $|p|$ iteraciones siendo p la longitud de la palabra lo cual esto es $\mathcal{O}(|p|)$.

5. Módulo Iterador Trie

Interfaz

Se explica con: ItUnidireccional(α)
Géneros: `ItDiccString(`α`)`.

Operaciones:

CrearIt(**in** d: `DiccString(`α`)`) $\rightarrow$ res : `ItDiccString(`α`)`
Pre $\equiv$ {true}
Post $\equiv$ {alias(esPermuta(SecuSub(res),claves(d)))}
Complejidad: $O(1)$
Descripción: El iterador se invalida si se modifican claves del diccionario

HayMas?(**in** it: `ItDiccString(`α`)`) $\rightarrow$ res : `res: Bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{\text{obs}}$ True $\iff$ HayMas?(it)}
Complejidad: $O(1)$
Descripción: Devuelve true si hay mas claves para recorrer

Actual(**in** it: `ItDiccString(`α`)`) $\rightarrow$ res : `tupla`$\langle$`String,`α $\rangle$
Pre $\equiv$ {HayMas?(it)}
Post $\equiv$ {res $=_{\text{obs}}$ Actual(it) }
Complejidad: $O(|p|)$
Descripción: Devuelve una tupla con el elemento actual y su significado

Avanzar(**in/out** it: `ItDiccString(`α`)`) $\rightarrow$ res : `res`
Pre $\equiv$ {it = it0 $\wedge$L HayMas?(it)}
Post $\equiv$ {it = avanzar(it0)}
Complejidad: $O(1)$
Descripción: avanza a la posicion siguiente del iterador

Representación Para recorrer el DiccTrie, aprovecho que tengo el conjunto de claves y uso el iterador de conjunto lineal. Para obtener el elemento, busco la 'clave actual' en el diccionario

`ItDicctrie` **se representa con** `estr`

donde `estr` es `tupla`(*itClave*: `ItConj(String)`
, *dicc*: `DiccTrie(`α`)`)

5.1. Invariante de representacion

.Si raiz es NULL entonces Claves es vacio
.Si raiz no es NULL entonces claves no es vacio
.Todos los elementos dentro de claves estan definidos en el diccstring
.Ningun Nodo puede apuntar dentro de siguientes a un nodo que se encuentra en una posicion anteriora el

5.2. Algoritmos

iCrearIt(in d **:** `DiccTrie(`α`)`**)** $\rightarrow$ *res* **ItDiccTr(**α**)**

1: $res \leftarrow \langle$d, CrearIt(claves(d))$\rangle$

Complejidad: $\mathcal{O}(1)$
Justificación: crear el iterador de conjunto $\mathcal{O}(1)$, obtener las clavesd deld dicc es $\mathcal{O}(1)$, crear la tupla $\mathcal{O}(1)$

iHayMas(in *iter* **:** `ItDiccTrie(`α`)`**)** $\rightarrow$ *res* **bool**

1: $res \leftarrow$ HaySiguiente(iter.ItClave)

Complejidad: $\mathcal{O}(1)$
Justificación:

iActual(in *iter* **:** `ItDiccTrie(`α`)`**)** $\rightarrow$ *res* **tupla** $\langle$ **string,** α $\rangle$

1: $res \leftarrow \langle$ siguiente(iter.ItClave), obtener(siguiente(iter.ItClave), iter.DiccTrie) $\rangle$

Complejidad: $\mathcal{O}(|p|)$
Justificación: Buscar en un dicctrie es $\mathcal{O}(longituddelaclavemaslarga)$ en el peor caso y acceder al siguiente del conjunto es $\mathcal{O}(1)$

iAvanzar(in/out *iter* **:** `ItDiccTrie(`α`)`**)**

1: Avanzar(iter.ItClave)

Complejidad: $\mathcal{O}(1)$
Justificación: avanzar un iterador de conjunto lineal $\mathcal{O}(1)$