Resolución 1 ejercicio:

El sistema de versionado se encarga de asignar una versión al entregable liberado al cliente con cierto criterio. Si el producto liberado posee grandes cambios en relación a la versión anterior, se inicia una versión mayor nueva. Si los cambios son moderados, se le asigna una nueva versión intermedia. Si los cambios son mínimos, se le asigna una versión menor. Si los cambios son muy reducidos, se le asigna una versión de parche. Dichas versiones son números consecutivos que arrancan en 0. Finalmente, cada liberación de un entregable puede ser de tipo alfa, beta o final. A continuación, se presenta el formato de versión según los criterios definidos. Formato de versión: mayor.intermedia.menor.parche{tipo} Por ejemplo, la versión de un producto X puede ser 1.2.1.0alfa, lo cual indica versión mayor 1, intermedia

1. Implementar la estructura del árbol TAD Versiones que represente al sistema de versionado, junto con los TAD que considere necesarios. (0,5 pts)

TAD Versiones

Tipo:

Versiones: registro

producto: cadena

versión: Versión

liberación: Liberación

fin registro

TAD Bugs

Tipo

Bugs: registro

Descripcion: cadena

Identificador: entero

Fin registro

Proced CrearBug(ref b:Bugs, descripcion: cadena, identificador: entero)

b.Descripcion🡨 descripcion

b.identificador🡨 identificador

fin proced

#funcion getDesBug (b: Bugs):cadena

#Funcion getIdenBug (b:Bugs): entero

TAD liberación

Tipo

Liberación= registro

nombre: cadena

bugs:Bugs

Finreg

Proced nuevaLib(lib: Liberacion, nombre: cadena)

Lib.nombre🡨 nombre

Fin proced

#Funcion getLib (lib: Liberación): cadena

TAD Version

Tipo

Version: registro

tipoVersion: cadena

numVersion: entero

fin reg

proced nuevaVersion(ver: Version, tipo: cadena, num: entero)

ver.tipoVersion🡨 tipo

Version.num🡨 num

Fin proced

Funcion getTipoVersion(ver:Version):cadena

Funcion getNumVersion(ver: Version): entero

c) Implementar la operación generar Versión que, dado un producto, un nivel de cambio (mayor, menor, etc) y un tipo de liberación (alfa, beta o final), valide si es posible realizar una versión nueva y la devuelva en texto.(1 pts)

funcion generarVersion (producto: cadena, liberacion: cadena, ver: lista(Versiones)):bool

vars: datoX: Versiones, xs:Lista(Versiones)

mientras not esListaVacia(ver) hacer

head(ver, datoX)

tail(ver)

si datoX.versiones.tipoVersion = producto entonces

si validarLiberacion(datoX, liberacion) entonces

devolver(verdadero)

sino

devolver(falso)

fin si

fin si

fin mientras

si esListaVacia(ver) entonces

devolver(falso)

fin si

fin funcion

proced validarLiberacion(versiones: Lista(Versiones), liberacion: cadena): bool

vars: x:versiones.liberacion

si no esListaVacia(versiones) entonces

si liberacion= ´beta´ entonces

getLib(versiones.liberaciones)

si validarLiberacion(versiones.liberacion, liberacion) entonces

devolver(verdadero)

sino

devolver(falso)

fin si

sino

si liberacion= ´alfa´ entonces

getLib(versiones.liberaciones)

si validarLiberacion(versiones.liberacion, liberacion) entonces

devolver(verdadero)

sino

devolver(falso)

fin si

sino

devolver(verdadero)

fin si

fin si

fin funcion

#funcion validarLiberacion(liberacion=Lista(liberacion), liberacion: cadena)): bool /.si no hay versiones mas nuevas (en caso de alfa) como final o beta, devolver verdadero sino falso.

Implementar la operación próxima Versión que dado un producto y una versión, devuelva la siguiente versión de tipo final que resuelva todos los bugs detectados en la versión ingresada. (4 pts)

Funcion proxVersion (producto: cadena, version: cadena, ver: Versiones)): Versiones

Vars: x: Verisones, terminado; bool

Terminado🡨 falso

listaVacia(x)

proxVersion2 (producto, verison, ver, x, terminado)

devolver(x)

fin funcion

funcion proxVersion2 (producto, versión cadena, ver, x: Lista(Versiones), terminado: bool)): Versiones

si no esListaVacia(ver) entonces

head(ver, x)

tail(ver)

si x.producto = producto entonces

getTipoVersion(ver.Version)

si x. Version = Version y x. Version.tipoVersion= ‘final’ entonces

recorrerLiberaciones(ver.liberacion, terminado)

fin si

fin si

fin si

si not terminado entonces

proxVersion2(producto, versión: cadena, ver, x, terminado)

sino

devolver(x)

fin si

fin funcion

proced recorrerLiberaciones (lib: Lista(liberaciones), x: Lista(liberaciones), ref terminado: bool)

vars: x: Lista(Liberaciones)

listaVacia(x)

head(lib, x)

mientras no esListaVacia(lib) y x.bug= ‘ ’ hacer

head(lib, x)

tail(lib)

fin mientras

si no esListaVacia(lib) entonces

terminado🡨 verdadero

sino

terminado 🡨 falso

fin si

fin proced

Ejercicio 2

Una compañía de radio está planificando instalarse en una región para emitir su contenido radial, para lo cual desea realizar algunas estimaciones iniciales relacionadas a dónde ubicar las estaciones emisoras y las antenas repetidoras. Las antenas, sean emisoras o repetidoras, tienen una cobertura asociada que determina el radio de alcance en kilómetros. La región se compone de distintos asentamientos, que pueden ser ciudades o pueblos, los cuales tienen una cierta cantidad de habitantes. Si un asentamiento está dentro del radio de alcance de una antena, todos sus habitantes podrán escuchar los programas emitidos. El análisis se centra en simular la colocación de antenas en distintas ciudades para poder determinar a cuántos habitantes se podrá cubrir. Sólo se contempla la instalación de una antena en una ciudad, ya que los pueblos no poseen la infraestructura necesaria para hacerlo.

Se solicita realizar lo siguiente:

e) Diseñar e implementar la estructura del TAD Región, junto con los que considere necesarios para representar la situación descripta.

(0,5 pts)

(1,5 pts)

TAD Región

Tipo

Región= registro

antena: Antena

asentamientos: Asentamientos

fin reg

TAD Antena

Antena= registro

tipoAntena: cadena

cobertura: real

fin reg

proced newAntena(tipo: cadena, cobertura: real, ant: Antena)

Antena.tipoAntena🡨 tipo

Antena. Cobertura🡨 cobertura

Fin proced

# getTipoAnt(Ant: Antena):cadena

# getCober(Ant: Antena):real

TAD Asentamientos

Tipo:

Asentamientos: Registro

cuidad: cadena

cantCuidad: entero

pueblo:cadena

cantPueblo: entero

Radio: entero

fin reg

proced NewAsantamiento(asa: Asantamientos, habCui, habPueb: entero, Radio: entero)

asa.cantCuidad🡨 habCui

asa.cantPueb🡨habPueb

asa.radio🡨 radio

fin proced

getCantCui(asa: Asantamientos): entero

getCantPueb(asa: Asantamientos): entero

getRad(asa: Asantamientos): entero

f) Implementar la operación cantidadOyentes que permita calcular la cantidad de habitantes a los cuales les llega la señal de radio, dada una antena y la instalación de la misma en una ciudad. La cobertura de la antena se asume viene dada como una propiedad de la misma, por lo cual varía según la antena a instalar.

(2 pts)

Funcion cantidadOyentes(región: Lista(Region)): entero

Vars: xs: Lista(region),x, y: entero

Si es ListaVacia(región) entonces

Devolver(0)

Sino

x🡨 getCober(xs.antena)

y🡨 getRad(xs.asantamientos)

si x >= y entonces

devolver(getCantCui(xs.asantamiento) + cantidadOyentes(tail(Region))

fin si

fin funcion

g) Asumiendo que se han instalado algunas antenas en ciertas ciudades de la región, implementar una operación hayCoberturaMúltiple que, dadas dos ciudades con antenas instaladas, determine si existen asentamientos en la región en los cuales se recibe cobertura de radio por ambas antenas mencionadas.

Funcion hayCoberturaMultiple(Cuidad1, Cuidad2: cadena, región:Lista(Region))

Vars: xs: Lista(región)

si no esListaVacia(Region) entonces

getCuidad(xs.asentamiento)

Parcial 2021

1) Dada una lista de números enteros, implementar una operación rotaciones que permita generar una lista de listas, donde cada una de esas listas sea una rotación de la lista original. El resultado debe incluir la lista original como primer elemento, mientras que los elementos restantes deben estar ordenados según la rotación generada de mover la cabeza al final. (2 pts)

Ejemplo:

rotaciones( [1,2,3,4] ) = [ [1,2,3,4], [2,3,4,1], [3,4,1,2], [4,1,2,3] ]

proced Rotaciones (xs: Lista(entero), ref xss: Lista(Lista(entero)))

vars: ys: Lista(entero)

listaVacia(xss)

copiarLista(xs, ys)

Rotaciones2 (xs, ys, xss)

Fin proced

proced Rotaciones2 (xs, ys: Lista(entero),ref xss:Lista(lista(entero)))

vars: x: entero

si no esListaVacia(xs) entonces

tail(xs)

insertar(xss, ys)

head(ys, x)

tail(ys)

insertarUltimo (x, ys)

Rotaciones2(xs, ys)

Fin si

Fin proced

2) Un Sistema Operativo utiliza un clásico y simplificado sistema de archivos (Filesystem) con el cual permite organizar el contenido de los directorios y archivos almacenados en un disco de forma lógica. Sabemos que un sistema de archivos tiene un directorio raíz y cada directorio (también llamado carpeta) puede contener tanto otros directorios como también archivos. El sistema de archivos tiene también un nombre o etiqueta que lo caracteriza.

Los directorios tienen un nombre alfanumérico asociado. Los archivos pueden ser de tipo ASCII o binario (aunque pueden extenderse a más tipos eventualmente), también pueden ser que estén ocultos o visibles (esto aplica también para directorios), tienen un nombre (alfanumérico), una extensión (máximo 4 caracteres) y un espacio ocupado en disco (tamaño en cantidad de bytes utilizados). Todo elemento del sistema, sea directorio o archivo, debe tener asociado una dirección física donde se almacena el primer bloque del mismo en el disco (se simplifica esta dirección como una Cadena).

El acceso a cierto elemento del sistema de archivos se realiza a través de una ruta o path de navegación, la cual se compone, por ejemplo, de una Cadena con la siguiente forma:

/dir1/dir2/file1

● la primera ‘/’ siempre corresponde a la ubicación del directorio raíz

● el directorio dir1 (debería estar contenido dentro del directorio raíz)

● el directorio dir2 (debería estar contenido dentro del dir1)

● el archivo file1 (debería estar contenido dentro del dir2)

Se solicita realizar lo siguiente:

d) Implementar la operación moverDirectorio que, dado un sistema de archivos, una ruta origen y una ruta destino, realice el movimiento del directorio o archivo origen al directorio destino. El movimiento implica que el elemento apuntado en la ruta origen pasa a ubicarse dentro del directorio apuntado por la ruta destino. Se debe validar previamente que ambas rutas existan, que la ruta origen apunte a una ubicación distinta al directorio raíz (no se puede mover la raíz) y que la ruta destino apunte a un directorio (no puede apuntar a un archivo). (2 pts)

1. Implementar la estructura del TAD Filesystem que represente al sistema de archivos simplificado del enunciado, junto con los TAD que considere necesarios. Agregar las firmas de las operaciones de dichos TADs que utilice en los siguientes puntos. Implementar la operación crearFilesystem que permita inicializar un sistema de archivos con cierta etiqueta pasada por parámetro. La inicialización genera el directorio raíz vacío. (1 pt)

TAD

Tipo

FileSystem: registro

raíz: ElementoFL

etiqueta: cadena

fin reg

Tipo

ElementoFL: registro

Tipo: enum{directorio, archivo}

directorio: Directorio

archivo: Archivo

contenido: Lista(ElementoFL)

nombre: cadena

dirección: cadena

oculto: bool #verdadero si esta oculto, falso si no

visual: bool #verdadero si es visible, falso si no

espacio: entero

fin reg

Tipo

Archivo: registro

tipo: cadena

nombre: cadena

extensión: cad[4]

fin reg

Tipo

Directorio: registro

ruta: cadena

fin reg

Implementar la operación crearFilesystem que permita inicializar un sistema de archivos con cierta etiqueta pasada por parámetro. La inicialización genera el directorio raíz vacío.

Proced crearFlieSystem(etiqueta: cadena, ref Fl: FileSystem)

Vars: Elem: ElementoFL

Fl. Etiqueta🡨 etiqueta

listaVacia(elem.contenido)

Elem.nombre🡨 ‘ ’

Elem.Archivo.visual🡨 verdadero

Elem.Archivo.oculto🡨 falso

Elem.Archivo.tipo🡨 Directorio

Fin proced

b) Implementar la operación obtenerFileFísico que, dado un sistema de archivos y una ruta del sistema, devuelva la dirección física del archivo que corresponda. Si la ruta no existiera, es decir, el archivo específico no existe dentro del directorio de la ruta o la misma apunta a un directorio en lugar de archivo, debe devolver la dirección “0x0000” (dirección inexistente). (2 pts) Como ayuda pueden utilizar la siguiente operación que convierte una ruta del sistema (Cadena) en una lista de los elementos ordenados:

rutaLista(ruta: Cadena, ref elementos: Lista(Cadena))

ejemplo: /dir1/file1 => [“dir1”, “file1”]

Tip: Resolverlo a partir de una operación más abstracta que devuelva la referencia a un elemento dentro del sistema de archivos para poder aprovecharla en puntos siguientes.

Proced obtenerFileFisico (ruta: cadena, FL: FileSystem, ref dir: cadena)

Vars: path: Lista(cadena)

rutaLista(ruta, path)

OFF2(FL. Raíz, dir, path)

Fin proced

Proced OFF2 (Elem: ElemetoFL, ref dir: cadena, path: Lista(cadena))

Vars: El: cadena , encontrado: bool, elemento: Lista(ElementoFL), subElem: ElemtoFS

Si no esListaVacia(path) entonces

Head(path, El)

Tail(path)

Encontrado🡨 falso

Si esArchvio(El) entonces

Si Elem. Nombre = El y esListaVacia(path) entonces

Dir🡨 Elem.direccion

Encontrado 🡨 verdadero

Fin si

Sino

elemento🡨 Elem.contendio

Mientras no esListaVacia(path) y no encontrado hacer

Head(Elemento, x)

Tail(Elemento)

Si x = El entonces

Encontrado🡨 verdadero

OFF2(subElem, dir, path)

Fin si

Fin mientras

Finsi

Sino

Dir🡨 ‘0x0000’

Fin si

Fin proced

c) Implementar la operación espacioOculto que, dado un sistema de archivos y una ruta del sistema, devuelva el espacio ocupado por cada archivo oculto que se encuentre dentro del directorio apuntado por la ruta, incluyendo aquellos archivos ocultos que estén en subdirectorios del mismo. A su vez, también se debe devolver la proporción del espacio total ocupado de los ocultos respecto a todos: ocultos / (ocultos + visibles). (2 pts)

Es requisito que esta operación devuelva todo lo solicitado en un mismo recorrido.

Proced espacioOculto (FL: FileSytem, ruta: cadena, ref espacioOcup: real)

Vars: path:Lista(cadena)

Todos, ocultos: entero

Todos🡨 0

Ocultos🡨 0

rutaLista(ruta, path)

espacioOculto2(FL.raiz, todos, ocultos, path)

espacioOcup🡨 ocultos div (ocultos + todos)

devolver(espacioOcup)

fin proced

# rutaLista(ruta: Cadena, ref elementos: Lista(Cadena)) (convierte una ruta del sistema (Cadena) en una lista de los elementos ordenados)

proced espacioOculto2 (Elem: ElementoFL, ref todos, ocultos: entero, path: lista(cadena))

vars: x: cadena, elemento: lista(ElementoFL), subElem: ElementoFL

Si no esListaVacia(path) entonces

Head(path, x)

Tail(path)

Si Elem.nombre =x entonces

Si Elem.oculto entonces

oculto🡨 oculto + Elem.espacio

Fin si

Todos🡨 todos + Elem.espacio

espacioOculto2(Elem, todos, ocultos, path)

sino

elemento🡨 Elem.contendio

Mientras no esListaVacia(elemento) hacer

Head(Elemento, subElem)

Tail(Elemento)

Si subElem.nombre = x entonces

Si Elem.oculto entonces

oculto🡨 oculto + Elem.espacio

Fin si

Todos🡨 todos + Elem.espacio

espacioOculto2(subElem, oculto, todos, path)

Fin si

Fin mientras

Finsi

Fin proced

d) Implementar la operación moverDirectorio que, dado un sistema de archivos, una ruta origen y una ruta destino, realice el movimiento del directorio o archivo origen al directorio destino. El movimiento implica que el elemento apuntado en la ruta origen pasa a ubicarse dentro del directorio apuntado por la ruta destino. Se debe validar previamente que ambas rutas existan, que la ruta origen apunte a una ubicación distinta al directorio raíz (no se puede mover la raíz) y que la ruta destino apunte a un directorio (no puede apuntar a un archivo). (2 pts)

proced moverDirectorio (FL: FileSystem, Origen: cadena, Destino: cadena)

vars: Elem: ElementoFL, ElemOrigen, x: ElementoFL, ElemDestino: lista(ElementoFL), encontrado: bool, archivo: ElementoFL

Elem🡨 raíz(FL)

sí validar(origen, Elem) y validar(destino,Elem) entonces

buscarArch(origen, FL.raiz, archivo)

ElemDestino🡨 Elem.contenido

Mientras no esListaVacia(ElemDestino) y no encontrado hacer

Head(ElemDestino, x)

Tail(ElemDestino)

Si x = destino entonces

Encontrado🡨 verdadero

Insertar(archivo, destino)

Fin si

Fin mientras

Fin proced

Funcion validar (x: cadena, Elem: ElementoFL): bool

Var: elemt: Lista(ElementoFL), subElemt: ElementoFL, encontrado: bool

Encontrado<- falso

Elemt🡨 Elem.contenido

Mientras no esListaVacia(elemt) y no encontrado hacer

Head(elemt, subElemt)

Tail(elemt)

Si subElemt. nombre= x entonces

Encontrado🡨 verdadero

Devolver(encontrado)

Finsi

Fin mientras

Fin funcion

Proced buscarArch(origen: cadena, elem: ElementoFL, ref archivo: ElementFL)

Var: elemt: Lista(ElementoFL), subElemt: ElementoFL, encontrado: bool

Encontrado<- falso

Elemt🡨 Elem.contenido

Mientras no esListaVacia(elemt) y no encontrado hacer

Head(elemt, subElemt)

Tail(elemt)

Si subElemt, nombre = origen entonces

Encontrado🡨 verdadero

Archivo🡨 subElemt

Finsi

Fin mientras

Fin funcion