DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Ejercicio obligatorio de la practica 3.1

Algoritmos y Estructuras de Datos II

Integrante	LU	Correo electrónico
Castro Russo, Matias Nahuel	203/19	castronahuel14@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. Cumple

`Cumple` **se representa con** `estr`

donde `estr` es `tupla`(*negocios*: `DiccAVL(Negocio, TriplaNegocio)` ,
regalos: `Conj(Regalo)` ,
negociosValidos: `Conj(Negocio)` ,
negocioConRegMasBarato: `itDiccAVL(Negocio, TriplaNegocio)`)

`TriplaNegocio` **se representa con**

donde es `tupla`(*iNegocio*: `itConj(Negocio)` ,
iregalosNegocio: `Conj( itConj(Regalo) )` ,
heap: `MinBinaryHeap(precio: Precio, iiregalo: ititRegalos)`)

Aclaraciones sobre la estructura

Todos los Conjuntos contenidos en mi estr, son conjuntos Lineales, denoto Conj(x) como tipo equivalente a ConjuntoLineal(x), y todas las incersciones que voy a hacer son agregadosRapidos, se que nunca voy a agregar un elemento, que ya este contenido previamente en ese conjunto, ahorrandome el costo de buscar si el elemento pertenece o no. Entonces siempre agrego en O(1)
Con MinBinaryHeap, me refiero a una cola de prioridad binaria, en la cual tiene mayor el elemento con menor precio.
Con DiccAVL me refiero a un Diccionario representado con un árbol AVL, osea un árbol balanceado en altura.

Explicaciones de algoritmos

publicarLista(in/out c: cumple, in n: negocio, in L: dicc(regalo, nat)):

Básicamente, defino(n,TriplaNegocio) en c.negocios, agrego los regalos de n a c.regalos y el negocio n a c.negociosValidos. El solo hecho de definir en un Diccionario AVL tiene complejidad O($logN$)
A continuación cuento un poco como es este proceso, que agrego y a donde, las complejidades, y los algoritmos que uso :

Creo un arreglo arr, con claves(L), voy a pedir que el diccionario se pueda iterar completo en O(L), siendo L la cantidad de claves del diccionario.
Hago un ciclo donde itero todo L, en cada iteración lo que voy a hacer es agregar la clave (el regalo) al conjunto en c.regalos, y me guardo un itConj(regalos) correspondiente al regalo recién agregado. Ese iterador lo agrego a mi conjunto iregalosNegocio, nuevamente me guardo un iterador que apunte al iterador recien agregado al conjunto. Este ultimo iterador va a ser el segundo elemento de una tupla (precio, este iterador).
Esta tupla la agrego al arr
Al finalizar el ciclo de iteración, voy a tener el arreglo arr con todas las tuplas (precio, regalo).
Usando el algortimo de Floyd, voy a representar un BinaryHeap con ese arreglo, en este caso usaré MinBinaryHeap, donde la mayor prioridad la tiene la tupla con precio menor a todas.
Básicamente con el algoritmo de Floyd represento un heap de la siguiente manera:
cada nodo del heap ocupa una posición determinada en el heap.

.si le llamo v a la raiz del heap, y con p(v), me refiero a la posición de v en dicho arreglo.Entonces $p(v) = 0$
.si v es el hijo izquierdo de u entonces $p(v) = 2p(u) + 1$
.si v es el hijo derecho de u entonces $p(v) = 2p(u) + 2$

De esta manera, usando Floyd, voy acomodando, cada subheap para que cumpla el invariante de un MinBinaryHeap(voy mirando y arreglando los subheap, empezando desde el más izquierdo y bajo. y voy iterando en orden).
Acomodar el heap para que sea uno valido, con este algoritmo, tiene una complejidad O(L), teniendo L nodos.

Entonces hasta tengo la TriplaNegocios, que lo unico que le falta(a TriplaNegocio) es el primer elemento, el cual es el itConj(Negocio). Entonces agrego el negocio n a c.negociosValidos (con agregadoRapido, ya que se, que n no pertenece previamente al conjunto). Y me guardo un iterador itConj(Negocio), apuntando a este negocio que estoy

agregando. Luego me guardo este iterador en π_1(TriplaNegocio) para la clave n por supuesto, en lo que estoy definiendo. AgregarRapido, y guardarme el iterador tiene complejidad O(1)

Lo único que me queda por ver es lo siguiente:
Si la lista que se esta publicando, es la primera, es decir, conjNegocios, tiene solo un elememento, especificamente n, el negocio que agregué recién. Entonces en c.negocioConRegMasBarato, guardo un iterador al negocio que estoy definiendo en el diccionario, ese iterador es itDiccAVL(Negocio,TriplaNegocio).
En caso contrario, comparo el precio minimo del heap del negocio agregado, con c.negocioConRegMasBarato.siguiente().heap.min(
En c.negocioConRegMasBarato, me quedo con el iterador al que contenga el menor. Todo este proceso de actualizar el iterador al más barato, tiene complejidad O(1)

∴ **Complejidad:** O((). L + log n)

---

regalos(in c: cumple,in n: negocio) → res: conj(regalo)

Busco en el DiccAVL de negocios, por la clave n, y yo se que el significado es una TriplaNegocio. La cual en el segundo elemento de la tripla (regalosNegocio) es un conjunto de iteradores del conjunto que contiene a todos los regalos del cumpleaños. Por ende el regalosNegocios solo va a contener los iteradores a los regalos correspondientes al negocio que corresponda. Entonces este es el conjunto que tiene lo que me pide:

c.negocios.obtener(n).regalosNegocio

creo un it de este conjunto c.negocios.obtener(n).regalosNegocio, y mi funcion devuelve ese iterador.

Obviamente al ser un iterador de un conjunto lineal, luego puedo recorrerlo tranquilamente, y tengo en cuenta que iteraria sobre un conjunto de iteradores al regalo.

Buscar en el diccionario AVL para llegar al significado de la clave n tiene una complejidad O(N), crear un iterador de un conjunto lineal O(1), y devolver el iterador O(1)

∴ **Complejidad:** O(1).

---

negociosConRegalos(in c: cumple) → res: conj(negocio)

Dado un cumpleaños, c.negociosValidos contiene a todos los negocios que aun tienen regalos. Como es parte de mi estructura, acceder a ella es O(1), entonces devuelvo un iterador a c.negociosValidos. Por lo tanto seria algo de este estilo:

it ← crearIt(c.negociosValidos) ▷ O(1)
return it ▷ O(1)

∴ **Complejidad:** O(1).

---

regaloMasBarato(in c: cumple) → res: regalo

En c.negocioConRegMasBarato tengo un iterador de negocios, especificamente el iterador es del negocio que contiene al regalo mas barato. Entonces acceder a la TriplaNegocio del negocio que lo contiene es O(1)
Luego para poder acceder al regalo mas barato, tengo que ver el mínimo del heap, nuevamente esto es O(1), pero el mínimo del heap me devuelve una tupla (Precio, ititRegalo), por lo cual para acceder realmente al regalo, necesito hacer ititRegalo.siguiente().siguiente(), que también es todo O(1)
Por lo tanto, seria algo de este estilo:

res ← c.negocioConRegMasBarato.heap.min().siguiente().siguiente() ▷ O(1)
return res ▷ O(1)

∴ **Complejidad:** O(1).

---

comprarRegaloMasBarato(in/out c: cumple)

En c.negocioConRegMasBarato, tengo un itDiccAVL, donde esta el regalo más barato, por ende, el regalo que va a ser comprado en esta función.
Por lo tanto, tengo acceso en (1), borro el regalo de c.regalos, y de iregalosNegocio.

Si al borrar ese regalo del conjunto de regalos del negocio que lo contenia, queda vacia, entonces su heap tambien lo esta. Eso implica que ese negocio no tiene más regalos por comprar. Entonces elimino al negocio de negociosValidos, este borrado es en O(1), ya que accedo a él mediante iNegocio.
Si el negocio esta vacio de regalos, entonces tambien lo elimino del diccionario, este eliminado es O(log N)

Luego tengo que actualizar c.negocioConRegMasBarato, itero entre todos los negocios en el DiccAVL, y me quedo el que tenga el más barato, esto es O(N)
Complejidad: $O(N + logN + logR) = O(N + N + logR) = O(2N + logR) = O(N + logR)$

```
comprarRegaloMasBarato (in/out c: cumple){

    #me guardo un iterador a c.negocioConRegMasBarato, para
    #hacer el pseudo más legible
    itDicc = c.negocioConRegMasBarato # O(log 1)

    #me guardo un it a iiregalo, del regalo más barato
    iiitReg = itDicc.siguiente().heap.min().iiregalo.obtenerIt() # O(log 1)

    #elimino el regalo de c.regalos
    iiitReg.siguiente().siguiente().eliminarSiguiente() # O(log 1)

    #elimino el iterador correspondiente en itDicc.siguiente().iregalosNegocio
    iiitReg.siguiente().eliminarSiguiente() # O(1)

    #desencolo el heap
    itDicc.siguiente().heap.desencolar() # O(log R)

    #si el conjunto que contiene itRegalos, es vacia, entonces:
    if(itDicc.siguiente().iregalosNegocio.vacia()): #O(1)

        #elimino el negocio en negociosValidos
        itDicc.siguiente().iNegocio.eliminarSiguiente() #O(1)

        #elimino ese negocio del DiccAVL, donde
        #itDicc.siguiente().iregalosNegocio es un conjunto vacio
        #itDicc.siguiente().heap es una cola de prioridad vacia
        itDicc.eliminarSiguiente() #O(log N)

    #creo un iterador del DiccAVL negocios con el cual voy a iterar
    it = CrearIt(c.negocios) # O(1)

    #creo otro iterador del DiccAVL negocios
    itMin = CrearIt(c.negocios) # O(1)
```

```
35
36
37      #recorro todo el DiccAVL negocios, y me quedo con un iterador al negocio
38      #que tenga el regalo más barato
39      while( it.haySiguiente() ):  # O(N)
40
41          #guardo el precio del regalo mas barato de cada negocio
42          precioActual = it.siguiente().heap.min().precio  # O(1)
43          precioMin = itMin.siguiente().heap.min().precio  # O(1)
44
45          if(precioActual < precioMin):  # O(1)
46
47              #el nuevo minimo pasa a ser el actual
48              itMin = it  # O(1)
49
50          avanzar(it)  # O(1)
51
52      #terminado el ciclo, tengo el itDiccAVL que contiene al regalo más barato
53      #actualizo en mi estr c.negocioConRegMasBarato
54      c.negocioConRegMasBarato = itMin  # O(1)
55
56      # Complejidad: O(N + log R)
57
58  }
```

Explicación de la estructura

regalos:

Es un Conj(Regalo), el cual contiene todos los regalos que están disponibles para comprar.

A medida que se publican las listas de regalos, se van agregando todos los regalos publicados en esta.
Cada regalo es agregado con agregadoRapido O(1), ya que por la precondición, al publicar listas, nunca puedo agregar un regalo que ya pertenezca al conjunto de todos los regalos.

Por otro lado, al comprarse un regalo 'r', entonces 'r' es eliminado de este conjunto.
Por las estructuras y las funciones, al eliminar 'r', me voy a ahorrar la busqueda de 'r', ya que voy a tener un iterador it, dado que it.siguiente() sea 'r'. Por lo tanto gracias a it.eliminarSiguiente(), puedo borrar con complejidad O(1)

---

negociosValidos:

Es un Conj(Negocio), conjunto el cual va a contener todos los negocios que aún tengan regalos pendientes a comprar.

Al publicar una Lista de regalos en un negocio n, ese n es insertado en este conjunto(agregadoRapido, ya que ese n no puede pertenecer previamente al conjunto, por lo tanto la incersión será O(1)).

Cuando la lista de regalos de un negocio n quede vacía, porque han sido comprados todos los regalos que contenia la lista. Entonces, el negocio n será eliminado de este conjunto negociosValidos.
Eliminar este negocio n, tendrá una complejidad O(1), ya que al momento previo del eliminado, estaré en la TriplaNegocio que le corresponda al negocio n, la cual tiene un iterador it tal que it.siguiente() sea este negocio n. Ahorrandome

la busqueda de n en el conjunto.
Este iterador it es π_1(triplaN), lo que es equivalente a decir triplaN.negocio, siendo(triplaN = e.negocios.obtener(n))

---

negocios:

Es un **DiccionarioAVL(Negocio, TriplaNegocio)**

`TriplaNegocio` **se representa con**

donde es `tupla`(*iNegocio*: `itConj(Negocio)` ,
iregalosNegocio: `Conj( itConj(Regalo) )` ,
heap: `MinBinaryHeap(precio: Precio, iiregalo: ititRegalos)`)

.iNegocio es un iteradorador de e.negociosValidos, al hacer c.negocios.obtener(n).iNegocio.siguiente() tengo acceso al negocio n, contenido dentro del conjunto e.negociosValidos. Este iterador me es útil para poder eliminar n del conjunto lineal de negocios validos, con una complejiad O(1). Por ejemplo borrar n vendria a ser algo asi:
c.negocios.obtener(n).iNegocio.EliminarSiguiente()
Por supuesto, luego de eliminar n, c.negocios.obtener(n).iNegocio.siguiente() va a ser basura, se invalida.
Pero en el único caso que tengo que eliminar n, es porque la lista en ese negocio es vacía, por ende no tiene sentido que el diccionario negocios siga teniendo la clave n tampoco, entonces tambien la eliminaría del diccionario.

.iRegalosNegocio es un conjunto de itConj(Regalo), especificamente es un it del conjunto regalos, el cual contiene a todos los regalos publicados en todos los negocios.
Recorriendo iRegalosNegocio:

```
    ii := crearIt(c.negocios.obtener(n).iRegalosNegocio))

    while(haySiguiente(ii)){    //O(n)
        // no se lo asigno a nada, es solo para mostrar como obtendria cada regalo
        ii.siguiente().siguiente()  //O(1)
        avanzar(ii) //O(1)
}
```

Puedo obtener todos los regalos aún no comprados en el negocio n

Al publicar una lista nueva en negocio n', voy a agregar todos los regalos de esa lista a c.regalos, y guardarme todos los respectivos iteradores en c.negocios.obtener(n').iRegalosNegocios.

Al comprar un regalo r, en el negocio n, sucede lo siguiente:
r claramente es c.negocios.obtener(n).heap.min().iiregalo Entonces, elimino r de regalos, elimino el it contenido en iRegalosNegocio que apuntaba a r, y desencolo el heap.

.heap: es MinBinaryHeap(precio: Precio, iiregalo: ititRegalos).
Este heap representa a una cola de prioridad de tuplas(precio, iiregalo), el precio es el valor que tomo como prioridad, el elemento con mayor prioridad, será aquel tupla que contenga al precio menor que todo el resto de elementos.
Y donde iiregalo es un iterador de un conjunto de iteradores del conjunto que contiene todos los regalos del cumpleaños.
Esto quiere decir que se accede al regalo más barato de de un heap h, de la siguiente manera:

```
h.min().siguiente().siguiente()
```

El heap va a ser modificado únicamente al comprarse el regalo que contenga en su raiz, el mismo se quita haciendo h.desencolar() y tiene un costo O(log R)

El heap lo creo usando el algoritmo de Floyd,pudiendoo crearlo y acomodarlo para que sea un heap valido en O(L), siendo L la cantidad de nodos con la que quiero crear el heap (para mas detalle de como creo este heap, y lo relaciono con las demas estructuras, ver la explicación del algoritmo PublicarLista en la página 1)

---

negocioConRegMasBarato:

Es itDiccAVL(Negocio, TriplaNegocio), osea es un iterador de e.negocios.
e.negocioConRegMasBarato.siguiente() me lleva al negocio que contiene el regalo más barato.

Este iterador cambia, se actualiza al momento de comprar un regalo, ya que siempre el regalo a comprar va a ser el mas barato de todos, por ende, el regalo que obtengo desde este iterador, seria algo así:

c.negocioConRegMasBarato.siguiente().heap.min().iiregalo.siguiente().siguiente() ▷ O(1)

Luego de comprarlo, obtener el nuevo más barato requiere de iterar entre todos los negocios, mirando el regalo más barato de cada uno, guardandome el itDiccAVL(Negocio, TriplaNegocio) del que lo contenga. Esta búsqueda del iterador para encontrar, crear los iteradores y guardarme el mínimo de todos ellos, tiene una complejidad O(N)

El otro momento donde puede cambiar c.negocioConRegMasBarato es al publicar una Lista nueva en negocio n, ya que esa lista puede tener un regalo más barato al que consigo con esta estructura. Por lo cual tengo que comparar el precio del regalo mas barato hasta el momento, con el mínimo del heap recien creado, y quedarme con el menor. En caso de empate, no modifico negocioConRegMasBarato. Esta comparación es O(1), al igual que crear el itnegocios del nuevo contenedor del más barato, en caso de necesitarlo.
En el caso de que al agregar esa lista, sea la primer lista, entonces: a c.negocioConRegMasBarato le asigno el iterador correspondiente al negocio que esta siendo agregado. Esto tambien tiene complejidad O(1)

Aclaración sobre la iteración en un DiccAVL:

Cuando recorrí diccionarios AVL, asumí que recorrer todo el diccionario tiene una complejidad O(N).
Dicha complejidad se consigue con un algoritmo del estilo de Inorder, visto en el taller. Hay varias maneras de implementarlo, recursivamente o no.
Por ejemplo, una de las maneras podría ser algo de este estilo:

```
(1) Crear una pila S.
(2) Inicializar el nodo actual a la raiz.
(3) Apilar el nodo actual y mover actual a la izquierda(actual = actual->izq) hasta que actual sea null.
(4) Si actual no es null y S no esta vacio
    .(a) Imprimir la cima(c) de S.
    .(b) actual = c->der.
    .(b) volver al paso 3
(5) si actual es null y S esta vacio, terminamos.
```

Este algoritmo itera por todo el AVL, y lo imprime. En caso querer recorrerlo y no imprimir, hago las operaciones que necesite en su lugar.