Algoritmos y Estructuras de Datos II

Trabajo Práctico 3

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

Diseño y elección de estructuras

Integrante	LU	Correo electrónico
Bruno, Patricio	62/19	pdbruno@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

1. Dar una estructura de representación del módulo Cumpleaños explicando detalladamente qué información se guarda en cada parte, las relaciones entre las partes, y las estructuras de datos subyacentes.

Servicios Usados:

- El diccionario que me entra como parámetro en PublicarLista debe pertenecer a un módulo que se explique con el TAD Diccionario y que sea posible iterarlo para recorrer todas sus definiciones. Avanzar() debe costar O(1).
  Un ejemplo concreto de una estructura que se comporte así es el diccionario lineal del apunte de módulos básicos de la materia
- Una Cola de Prioridad implementada con un min-heap (que a su vez puede puede implementado tanto con un array como con un árbol binario). En este caso particular, la cola que utilizo contiene tuplas donde su campo “menorPrecio” es quien determina el orden de la cola. En cuanto a complejidad, el min-heap me permite obtener el mínimo elemento en timepo constante, encolar en O(log n) y desencolar en O(log n).
- Un diccionario implementado con un árbol AVL, cuyos tiempos de búsqueda e inserción sean logarítmicos en función de la cantidad de claves. Me referiré a esta estructura como diccLog.
- El conjunto lineal del apunte de módulos básicos.

Cumpleaños se representa con estr
donde estr es tupla(

- negocios: diccLog(negocio, tupla(
  conjRegalos: conjLineal(regalo)
  colaRegalos: vector(tupla(precio: nat,
  regalo:itConjLineal(regalo))),
  itNegociosConRegalos: itConjLineal(negocio)
)
)
- negociosXPrecio: colaPrio(itDiccLog(negocio, tupla(...)))
- negociosConRegalos: conjLineal(negocio)

1. negocios es un diccionario logarítmico donde las claves son negocios (nat) y los significados son una tupla que contiene:
   a. un conjunto lineal de regalos aún no comprados en ese negocio (para devolverlo en Regalos() en O(1))
   b. un vector con tuplas que contienen un iterador a dicho conjunto (para borrar el regalo si lo compran) y el precio que corresponde al regalo. La idea es insertar todas las tuplas iterador-precio en el vector y luego aplicar el algoritmo de Floyd a dicho vector para que en tiempo lineal pase a cumplir con el invariante de min-heap. Naturalmente, el campo “precio” de cada tupla es quien determina la relación de orden. De esta forma soy capaz de obtener el mínimo precio en O(1) y desencolar en tiempo logarítmico.
   c. un iterador al conjunto de negocios con regalos (para borrar el elemento si corresponde).
2. negociosConRegalos es un conjunto lineal de todos los negocios que aún tienen regalos disponibles.
3. negociosXPrecio es una cola de prioridad implementada con un min-heap que me permita obtener el minimo en O(1), encolar en en O(log(n)) y desencolar en O(log(n)). Cada elemento es un iterador al diccLog de negocios. Haciendo siguiente() de cada iterador se puede acceder a la cola de regalos de cada negocio y luego a su menor elemento (próximo()) en tiempo constante. Allí encontraremos al regalo más barato y su **precio**. Éste dato es el que se usa para ordenar el min-heap de negociosXPrecio. Reitero que dado un elemento de la cola de prioridad, puedo acceder a su precio más barato en O(1), así que encolar y desencolar no empeoran su complejidad, manteniéndose logarítmicos. Por último, como obtener el menor elemento de un heap es a su vez O(1), se puede acceder al regalo con el menor precio de toda la estructura en tiempo constante.

Nótese que EliminarSiguiente y AgregarRápido de itConj es O(1), por eso se eligió un conjunto lineal.

2. Justificar de qué manera es posible implementar los algoritmos para cumplir con las complejidades pedidas. Escribir el algoritmo para la operación ComprarRegaloMásBarato.

a) publicarLista(in/out c: cumple, in n: negocio, in l: dicc(regalo, nat))
O(1) para crear un conjunto vacío
O(1) para crear un vector vacío
O(L) para iterar sobre todos los regalos del diccionario y
1- agregar el regalo al conjunto (agregarRapido es O(1)) (me quedo con un it)
2- agregar el precio y dicho iterador al final del vector (recordar que insertar L elementos seguidos en un vector tiene costo amortizado O(1) por inserción)
O(L) para aplicar el algoritmo "heapify" de Floyd al vector. De ahora en más, puede actuar como una cola de prioridad.
O(1) para agregar el negocio al conjunto de negocios con regalos (agregarRapido es O(1)) (me quedo con un it)
O(1) para crear un tupla con la lista, el conjunto y el iterador a conjunto
O(log n) para definir la tupla como significado y el negocio como clave en el diccLog

Todas las operaciones son O(1), excepto definir de diccLog (O(log n)), iterar l (O(L)) y "heapificar" el vector (O(L)). Complejidad final: O(log n + L)

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

```
res ← Significado(cumple.negocios, n)->conjRegalo
```

Significado es O(log n) por ser un diccionario logarítmico. Acceder al dato de un puntero y acceder al campo de una tupla es O(1)
Complejidad final: O(log n)

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

```
res ← cumple.negociosConRegalos //O(1)
```

Complejidad final: O(1)

d) regaloMásBarato(in c: cumple) → res: regalo

```
itNegocio ← proximo(cumple.negociosXPrecio) //O(1), por ser un minHeap
colaRegalos ← siguiente(itNegocio)->colaRegalos //O(1)
itRegalo ← proximo(colaRegalos).regalo //O(1), por ser un minHeap
res ← siguiente(itRegalo) //O(1)
```

Complejidad final: O(1)

e) comprarRegaloMásBarato(in/out c: cumple)

```
itNegocio ← proximo(cumple.negociosXPrecio) //O(1)
conjRegalos ← siguiente(itNegocio)->conjRegalos //O(1)
colaRegalos ← siguiente(itNegocio)->colaRegalos //O(1)
itRegalo ← proximo(colaRegalos).regalo //O(1)
eliminarSiguiente(itRegalo) //O(1)
desencolar(colaRegalos) //O(log R)
desencolar(cumple.negociosXPrecio) //O(log n)
if(esVacio?(conjRegalos)){ //O(1)
      itNegociosConRegalos ←
siguiente(itNegocio)->itNegociosConRegalos //O(1)
      eliminarSiguiente(itNegociosConRegalos) //O(1)
}else {
      encolar(cumple.negociosXPrecio, itNegocio) //O(log n)
}
```

Todas las operaciones son O(1), excepto desencolar y encolar de las colas de negocios y de regalos (O(log n) y O(log R) respectivamente). Una posible optimización es desencolar el negocio si ya no tiene regalos, y hacer un “shift-down” si todavía tiene regalos. Esto nos ahorra una operación de complejidad O(log n).

Complejidad final: O(log n + log R)