Estructuras de Datos y Algoritmos – IIC2133

I2

19 mayo 2014

Ordenación

1) Explica cómo ordenar una baraja de naipes por pinta (en el orden espadas, corazones, diamantes y tréboles) y por rango dentro de cada pinta (as, rey, reina, *jack*, 10, 9, …, 2), con la restricción de que las cartas están dispuestas cara abajo en una fila, y las únicas operaciones permitidas son mirar los valores de dos cartas e intercambiar las posiciones de dos cartas (manteniéndolas cara abajo).

Respuesta:

insertionSort, pero con intercambios (también *selectionSort* con intercambios, que se convierte en algo así como *bubbleSort*).

2) *Mergesort natural* es una versión *bottom*-up de mergesort que aprovecha el orden que pueda haber en el arreglo original: encuentra un subarreglo ordenado (incrementa un puntero hasta encontrar un dato que sea menor que su predecesor en el arreglo), luego encuentra el próximo subarreglo ordenado, y luego los mezcla.

Escribe un algoritmo (en pseudo código) de mergesort natural para ordenar una *lista ligada*. (Este es el método preferido para ordenar listas ligadas porque no usa espacio adicional y su orden de crecimiento es de la forma $n\log n$.)

Respuesta:

El algoritmo tiene dos partes. En la primera, se busca dos sublistas (consecutivas) ordenadas, como se describe en el primer párrafo, y se las indentifica con punteros al inicio y al final de cada una. En la segunda parte, se hace la mezcla, usando la misma estrategia general de *mergesort*, pero mezclando listas ligadas en lugar de arreglos. Esto significa que no se van traspasando valores del arreglo original a un segundo arreglo, sino que se van revinculando los punteros de las listas.

3) Si ordenamos el archivo de empleados de la Universidad Católica por año de nacimiento, muy probablemente habrá muchas claves duplicadas. Si bien quicksort se desempeña bien en esta situación, es posible mejorar mucho su desempeño; p.ej., un subarreglo que consiste sólo de ítemes iguales (todos tienen la misma clave) no necesita seguir siendo procesado, pero quicksort sigue particionándolo en subarreglos más pequeños. Una posibilidad es particionar el arreglo en tres partes: una para los ítemes con claves menores que el pivote, otra para los ítemes con claves iguales al pivote, y otra para los ítemes con claves mayores que el pivote.

Escribe un algoritmo (en pseudo código) de *partición en 3* que haga una pasada de izquierda a derecha sobre el arreglo $a[e \ldots w]$, manteniendo tres punteros: un puntero *less* tal que $a[e \ldots less-1]$ son menores que el pivote v; un puntero *greater* tal que $a[greater+1 \ldots w]$ son mayores que v; y un puntero i tal que $a[less \ldots i-1]$ son iguales a v y $a[i \ldots greater]$ aún no han sido examinados.

Respuesta:

```
less = e
i = e+1
greater = w
while (i <= greater)
      if (a[i] < v)
            exchange(a[less], a[i])
            less = less+1
            i = i+1
      else
      if (a[i] > v)
            exchange(a[i], a[greater])
            greater = greater-1
      else
            i = i+1
return (less, greater)
```

Grafos.

4) Kevin Bacon es un actor que ha trabajado en muchas películas. En el mundo del cine, asignamos el *número Kevin Bacon* a un actor de la siguiente manera. El propio Kevin Bacon recibe un 0; un actor que ha trabajado en una película junto a Kevin Bacon recibe un 1; un actor que ha trabajado en una película junto a un actor cuyo número es 1, recibe un 2; etc. (Por supuesto, si un actor califica para dos o más números distintos, se le asigna el menor de esos números.) Describe y justifica un esquema eficiente basado en grafos no direccionales para determinar el número Kevin Bacon de un actor.

Respuesta:

Construimos un grafo en que los nodos son las películas y los actores (dos tipos distintos de nodos, pero todos nodos al fin), y las aristas unen las películas con los actores correspondientes; no hay aristas entre películas ni aristas entre actores.
A partir del nodo correspondiente al actor que nos interesa, ejecutamos BFS hasta llegar al nodo que corresponde a Kevin Bacon; el número Kevin Bacon del actor que nos interesa es un medio de la longitud del camino encontrado (en número de aristas).

5) Considera el algoritmo de Prim, estudiado en clase, para encontrar un árbol de cobertura mínimo para un grafo no direccional $G = (V, E)$, a partir del vértice r en E:

```
for ( cada vértice u en V ) u.key = ∞
r.key = 0
formar una cola Q con todos los vértices en V, priorizada según el campo key de cada vértice
π[r] = null
while ( !Q.empty( ) )
        u = Q.extractMin( ) —esta operación modifica la cola Q
        for ( cada vértice v en listadeAdyacencias[u] )
                if ( v ∈ Q ∧ costo(u,v) < v.key )
                        π[v] = u
                        v.key = costo(u,v) —esta operación modifica la cola Q
```

El desempeño de Prim depende de cómo se implementa la cola Q; si Q es un heap binario, entonces Prim toma tiempo $O(E \log V)$: el *for* dentro del *while* mira cada arista de G y para cada una realiza una operación *decreaseKey* sobre un vértice (cuando actualiza $v.key$, en la última línea).

Si los costos de todas las aristas de G son números enteros entre 0 y una constante W (tal que es factible declarar un arreglo de tamaño W+1), describe una forma de implementar la cola Q, tal que Prim corra en tiempo $O(E)$, es decir, que cada operación *descreaseKey* tome tiempo O(1). (Recuerda que si W es constante, es decir, no depende ni de E ni de V, entonces recorrer un arreglo de tamaño W+1 toma tiempo $O(1)$.)

Respuesta:

- Implementamos Q como un arreglo Q[0], Q[1], …, Q[W], Q[W+1]. Cada elemento del arreglo es una lista doblemente ligada de vértices, en que Q[k] contiene todos los vértices cuyo *key* vale k; Q[W+1] contiene los vértices cuyo *key* vale infinito (∞).
- Entonces, *extractMin* consiste simplemente en recorrer Q buscando el primer elemento no vacío; por lo tanto, toma $O(W) = O(1)$.
- También, *decreaseKey* toma tiempo $O(1)$: basta sacar el vértice de la lista en que está (la lista tiene que ser doblemente ligada para que sacar un vértice tome $O(1)$) y agregarlo al comienzo de la nueva lista que le corresponda según su nuevo valor de *key*.

6) En algunos lenguajes de programación es posible declarar (explícitamente) dos nombres de variables como equivalentes, es decir, son referencias al mismo objeto. Después de una secuencia de estas declaraciones, el compilador necesita determinar si dos nombres dados son equivalentes. En particular, supongamos que las declaraciones son de la forma `equivalent <id1> <id2>`. Cuando el compilador ve una de estas declaraciones, primero, tiene que determinar si, como consecuencia de declaraciones anteriores, `<id1>` e `<id2>` ya son equivalentes (p.ej., `equivalent <id1> <id3>` … `equivalent <id2> <id3>`), en cuyo caso no hace nada; si `<id1>` e `<id2>` no son equivalentes, entonces debe hacerlos equivalentes a partir de ese momento.

De acuerdo con las estructuras de datos y algoritmos estudiados en clase, ¿qué tan rápidamente puede el compilador procesar una secuencia de p declaraciones que en total involucran n nombres de variables distintos? Justifica.

Respuesta:

Se trata de unión de conjuntos disjuntos. Cada variable es inicialmente un conjunto por sí misma (*singleton*); y cada declaración `equivalent` une dos conjuntos en uno nuevo, si es que esas variables no están ya en el mismo conjunto. Así, hay n singletons y p declaraciones, cada una de las cuales implica dos operaciones *find* y posiblemente una operación *union*; en términos de la notación usada en los apuntes de clase, $m = n + 3p$.

Por lo tanto, si usamos la representación de conjuntos sugerida en la lámina 18 de los apuntes, y usamos unión por rango y compresión de ruta a medida que ejecutamos las operaciones sobre los conjuntos, el compilador puede hacer su trabajo en tiempo $O((n+3p)\alpha(n))$.