Pauta I2 parte I

1) Considera un árbol rojo-negro en que el número de nodos negros en cada ruta de la raíz a una hoja es *k*;

a) [1 pt] ¿Cuál es la altura máxima posible del árbol? ¿Y la mínima? Justifica.
b) [1 pt] ¿Cuál es el máximo número de nodos que puede tener el árbol? ¿Y el mínimo? Justifica.

En este árbol hacemos una inserción:

c) [2 pts] ¿Cuál es la cantidad máxima de cambios de color que pueden ocurrir? Justifica.
d) [2 pts] ¿Cuál es la cantidad máxima de rotaciones (simples o dobles) que pueden ocurrir? Justifica.

Solución:

a) [0.5 pts] La altura máxima posible del árbol es $2k$, y esta se obtiene con un árbol que alterna nodos rojos y negros desde la raíz hasta las hojas partiendo con la raíz negra y terminando con las hojas rojas. No puede ser más alto porque no puede haber más nodos negros por enunciado y no puede haber más nodos rojos por las reglas del árbol rojo-negro.
[0.5 pts] La mínima altura es k, y es posible con un árbol binario completamente hecho de nodos negros. No rompe ninguna regla de los árboles rojo negro y no puede haber un árbol de menor altura por enunciado.
b) [0.5 pts] El máximo de nodos se obtiene con el árbol descrito en la primera parte de parte a), el cual tiene $2k$ niveles completos. Este árbol tiene $\sum_{i=0}^{2k-1} 2^i = 2^{2k} - 1$ nodos en total.
Opción 2:
[0.5 pts] El máximo de nodos se obtiene con el árbol 2-4 equivalente al árbol descrito en la primera parte de parte a), el cual tiene k niveles completos con nodos 4. Este árbol tiene $\sum_{i=0}^{k-1} 3 \cdot 4^i = 3 \cdot \sum_{i=0}^{k-1} 4^i = 3 \cdot \frac{4^k - 1}{4-1} = 4^k - 1$ nodos en total.

[0.5 pts] El mínimo se obtiene con el árbol descrito en la segunda parte de a), el cual tiene k niveles completos. Este árbol tiene $\sum_{i=0}^{k-1} 2^i = 2^k - 1$.
c) [1 pt] El máximo de cambios de color que pueden ocurrir es O(k).
[2 pt] Por ejemplo, en el árbol descrito en la primera parte de a), si insertamos un nodo rojo, tendríamos que realizar un cambio de color en cada nivel hasta llegar a la raíz. No puede haber más cambios de color ya que los cambios de color se realizan solo en la ruta de inserción del nuevo dato además de los tíos de los nodos de la ruta de inserción.
d) [1 pt] El máximo de rotaciones que se hacen en la inserción es 1.
[1 pt] Una rotación corresponde a un reordenamiento de los datos en un nodo del árbol 2-4 equivalente y solo ocurre en la hoja en la que se insertó. Una vez rotado, el árbol sigue cumpliendo las propiedades de árbol rojo negro por lo que no son necesarias más rotaciones.

2) La pizzería intergaláctica M87 sirve pizzas de todos los incontables sabores existentes en todos los multiversos, y necesita ayuda para hacer más eficiente la atención de los millones de pedidos que recibe por segundo. Los pedidos funcionan de la siguiente manera:

- Una persona solicita una pizza del sabor que haya escogido y da su nombre. Su pedido se agrega al sistema.
- Cuando la pizza está lista, se llama por el altavoz a la persona que haya pedido ese sabor hace más tiempo, y, una vez entregada la pizza, se borra ese pedido del sistema.

Explica cómo usar tablas de hash para llevar a cabo este proceso eficientemente. ¿Qué esquema de resolución de colisiones debería usarse y por qué? ¿Qué es lo que se guarda en la tabla?

Solución:

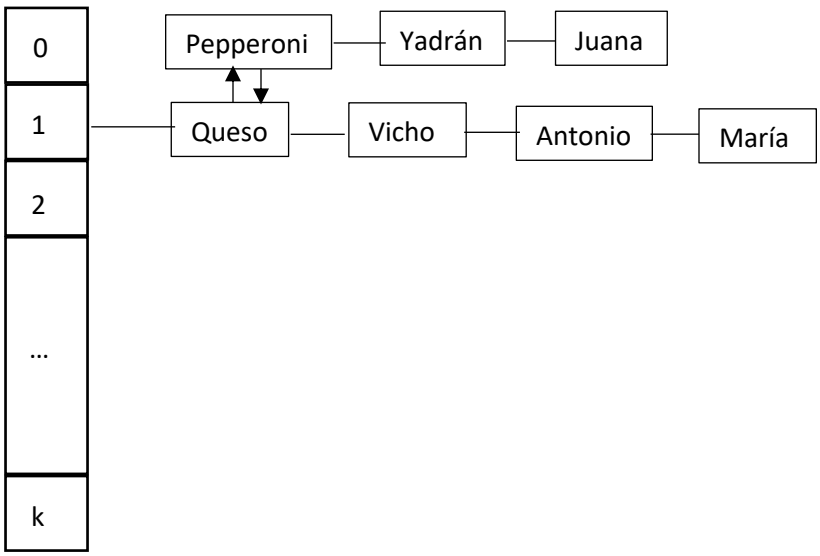
Para eso usamos una **tabla de hash** que nos permita guardar múltiples *values* para un mismo *key*. En este caso *key* corresponde al tipo de pizza y un *value* corresponde al nombre de la persona que lo pidió. **[1.5pts]**

Los *values* de un mismo *key* se deben guardar en una Cola (FIFO), de manera que agregar un nuevo *value* o extraer el siguiente sea $O(1)$ y se atienda en orden de llegada. **[1pt].** (Si deciden guardarlo en un Heap que ordena por orden de llegada, las operaciones son $O(log\ n)$, por lo que sólo obtienen **[0.5pts]**)

La eliminación del pedido sale automática con el heap o la cola ya que obtener el siguiente elemento lo extrae de la estructura. **[1pt]**

Pero como el dominio de las *key* es infinito, debemos ir despejando las celdas de la tabla cuando una *key* se queda sin *values*, ya que no tenemos memoria infinita. **[0.5pts]** Para esto es necesario usar **encadenamiento** ya que permite eliminar *key*s de la tabla sin perjudicar el rendimiento de esta. De esta manera, en caso de que varios sabores de pizza sean hasheados a la misma celda de la tabla (colisión de sabores), las colas FIFO de los nombres de las personas que pidieron esas pizzas deberán ser encadenadas en una lista doblemente ligada que las contenga **[2pts].**

Ej: tabla de hash con k celdas, con colisión de sabores Pepperoni y Queso en k = 1.



PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2133 — Estructuras de Datos y Algoritmos — 2/2019

Solución I2P2

Pregunta 1

Algoritmo para obtener el diámetro de un árbol usando DFS.
Partimos escribiendo un algoritmo que use DFS para obtener el nodo mas lejano a un nodo cualquiera v. (También es válido incluir DFS directamente en el algoritmo para obtener el diámetro, o implementar BFS).

DFS(v, parent, dist):

```
max_dist = dist
furthest = v
for vecino in α[v] do
  if vecino ≠ parent then
    dist_lejana, nodo_lejano = DFS(vecino, v, dist + 1)
    if max_dist < dist_lejana then
      max_dist = dist_lejana
      furthest = nodo_lejano
    end if
  end if
end for
return max_dist, furthest
```

Luego, usando esta función, podemos encontrar el nodo mas lejano al nodo elegido, el cual será un extremo del diámetro, el otro extremo (junto con el valor del diámetro) lo encontramos ejecutando la misma función, pero empezando por un extremo.

DIAMETRO(G(V, E))

```
v ← nodo extraído al azar
extremo, _ = DFS(v, NULL, 0)
_, diametro = DFS(extremo, NULL, 0)
return diametro
```

[2 pts] Implementar BFS o DFS.
[2 pts] Encontrar un extremo del diámetro a partir de un nodo al azar.
[2 pts] Encontrar el largo del diámetro a partir de un extremo.

Pregunta 2

Dado un grafo dirigido con costos $G(V, E)$ tal que el costo de cada arista puede ser 1 o 2.

a) Explica cómo encontrar la ruta de menor costo entre dos nodos dados usando BFS. Puedes modificar el grafo o el algoritmo para resolver este problema.

b) Demuestra que tu solución en a) es correcta.

c) Calcula la complejidad del algoritmo propuesto en a) con respecto a $|V|$ y $|E|$

Solución:
Existen varias soluciones diferentes y correctas, veremos 2 de ellas pero si lo hicieron de otra forma válida se considera correcto también:

1) La primera es modificar el gráfo:

a) Se reemplazan cada una de las aristas de costo 2 por 2 aristas de costo 1 y un nodo intermedio. Luego podemos utilizar BFS ya que tenemos un grafo con todas las aristas de costo 1 equivalente a el grafo anterior.

b) Para demostrar que esto es correcto es importante mencionar 2 cosas. Primero las rutas del grafo original tienen el mismo costo que las rutas del grafo modificado. Segundo, está demostrado que BFS encuentra la ruta más corta en un grafo sin costos (que es igual que un grafo con todas las aristas del mismo costo) por lo que la ruta obtenida es la óptima.

c) Sabemos que la complejidad de realizar BFS en un grafo de $|V|$ nodos y $|E|$ aristas es de $\mathcal{O}(|V| + |E|)$. Si el grafo inicial tenía $|V|$ nodos y $|E|$ aristas y reemplazamos todas las aristas de costo 2 por 2 aristas de costo 1 y un nuevo nodo intermedio, en el peor caso todas son de costo 2 por lo que se duplican las aristas y se agregan tantos nodos como aristas habían, quedando $|V| + |E|$ nodos y $2 \cdot |E|$ aristas. Entonces la complejidad de BFS quedaría $\mathcal{O}(|V| + 3 \cdot |E|)$ que es equivalente a $\mathcal{O}(|V| + |E|)$.

2) La segunda es modificar el algoritmo:

a) La modificación que se hace el algoritmo es al orden en que se revisan los nodos en la cola. BFS revisa los nodos en orden de profundidad pero entre nodos en la misma profundidad no hace una diferencia que en este caso, con aristas con costos mayores, si afecta el orden en que se revisan los nodos para llegar a la optimalidad. Por lo tanto, al revisar un nodo se almacenará el costo acumulado en llegar a ese nodo. Luego, en la cola de los nodos por revisar, se revisará primero el nodo cuyo costo acumulado para alcanzar es el más bajo. Si al expandirlo encontramos una forma menos costosa de llegar a uno de los nodos que queda por expandir, le cambiamos el costo por el menor de los dos. Se encontrará una solución cuando el nodo que se expandirá es el nodo final.
Esto es equivalente al algoritmo de Dijkstra.

b) Para demostrar que el algoritmo es correcto hay que demostrar que se expande cada nodo con la ruta a el de costo mínimo. Para demostrar esto realizaremos una inducción.

Caso Base: En la primera iteración del algoritmo entra a la cola el nodo inicio y se expande. Es trivial que al expandir el nodo inicio, se tiene la ruta óptima hacia él de costo 0.

Hipotesis de Inducción: En la iteración n del algoritmo, (en que se han sacado n nodos de la cola) asumimos que cada nodo fue sacado con su costo mínimo.

PD: Para el nodo que se expande en la siguiente iteración se llega con el costo mínimo posible.
A partir de que en la iteración anterior se había llegado al nodo expandido de forma óptima tenemos los siguientes casos en la cola:

1) Nodos que tienen el mismo costo de descubrimiento que el que se acaba de expandir.
2) Nodos con costo de descubrimiento +1 que el que se acaba de expandir
3) Nodos con costo de descubrimiento +2 que el que se acaba de expandir

Actualmente estamos sacando los nodos de 1) y agregando sus vecinos a 2) y a 3) según corresponda el costo de la arista para llegar a ellos. Todos los nodos de 1) fueron descubiertos con su costo mínimo ya que fueron descubiertos por algún nodo que ya salió de la cola y nuestra hipótesis de inducción nos dice que los que salieron de la cola fueron descubiertos con el costo mínimo.

Los nodos en 2) se separan en 2 casos: Los nodos que fueron descubiertos con una arista de costo 1 por un nodo a la profundidad actual y los nodos que fueron descubiertos por una arista de costo 2 por un nodo de la iteración anterior. En ambos casos el costo con el que saldrán de la cola es el mínimo para ese nodo.

Finalmente los nodos de 3) están en esa cola porque fueron descubiertos por un nodo a la profudidad actual por una arista de costo 2. Es posible que sacando nodos de 1) encontremos una arista de costo 1 que redescubra uno de esos nodos a menor profundidad. En ese caso el nodo se cambia su costo de descubrimiento y cambia al caso 2). En el caso en que no se redescubra de esta manera entonces la manera más barata de llegar al nodo es con esa arista final de costo 2.

Los nodos del caso 1) se expanden primero, luego del caso 2) y luego del caso 3). Por lo que queda demostrado que se expandiran con costo mínimo por lo que el algoritmo funciona.

Otra forma de demostrarlo es hablando de ordenar la cola segun el costo de descubrimiento y demostrar por que con eso se llega con costo mínimo a cualquier nodo.

No es correcto revisar todas las rutas posibles al nodo destino utilizando BFS por que BFS para funcionar asume que cuando encuentra la ruta a un nodo y lo expande esa ruta es la más corta. (Con costos 1 y 2 esto no se puede asegurar).

c) La complejidad de la adaptación del algoritmo depende de la implementación de la cola (o colas) para llevar a cabo el algoritmo. Si se utiliza un Heap para ordenar la cola entonces la complejidad del algoritmo sería de $\mathcal{O}(|V| + |E| + |V| \cdot log(V)$ por ordenar el heap para cada uno de los nodos que se revisan. Si se utiliza una cola para almacenar los nodos de cada uno de los casos 1), 2) y 3) entonces la complejidad es de $\mathcal{O}(|V| + |E|)$